09J 09H

CPU CACHE DUAL PROCESSOR DISTRIBUTED COMPUTATION SYSTEM WITHIN BROAD·CAST TYPE LOCAL AREA NETWORKS

BY

PETER H. KAMAU

B. Eng. (Hons.) Sheffield University

M. Sc. Essex University

A Doctoral Thesis

submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of the Loughborough University of Technology.

SEPTEMBER 1985

Supervisors:

Dr. M.E. Woodward, B.Sc., Ph.D.

Mr. J.E. Cooling, B.Sc.

of the

Department of Electronic & Electrical Engineering

Dec 85

010040|01

In memory of my mother Njeri, and to my wife Alice Mwithaga and daughters Njeri and Wambui for their patience, sacrifice and emotional support throughout the duration of my studies at Loughborough University.

ACKNOWLEDGEMENTS

I would like to thank Professor I.R. Smith, my director of research and Head of the Department of Electronic and Electrical Engineering, for providing the research facilities.

I would also like to express my deepest gratitude to my supervisors, Dr. M.E. Woodward and Mr. J.E. Cooling for their guidance, support and encouragement. Many thanks too must go to the staff of the Computer Centre of Loughborough University of Technology.

I am very grateful to my sponsor, The Association of Commonwealth Universities for the financial support and to my employer, The University of Nairobi for granting me study leave to enable me to pursue this research programme.

Finally, I would like to thank Mrs. Ashwell for the trouble and care she took in typing this thesis.

SYNOPSIS

Over the last few years computer hardware has continued to become smaller, cheaper, faster and more numerous. Computer software too has continued to become more efficient and powerful. The result has therefore been an availability of increasingly versatile microcomputers whose power rival that of minicomputers and many of the earlier generations of mainframe computers. At the same time, computers and communications have merged, with the result that computing power has become cheaper than communication. As the computation becomes cheaper and the machines become faster, the desire to solve larger and more complex problems will continue to increase.

This research programme is set up with the above as background. The aim of this research is to investigate some aspects of distributed computation and how this can be achieved in wideband broadcast-type communication channels, such as the ethernet, within a Local Area Network (LAN). In such a type of LAN efficient channel protocols based on Carrier Sense Multiple Access with Collision Detection may be employed since the round-trip propagation delay is small.

The specific method of distributing a computation employed in this research is that of the CPU cache dual processor distributed computation. In such a CPU cache dual processor distributed computation system a smaller computer may decide to share some of its computational workload with a larger and more powerful computer existing within the same LAN. Furthermore, several such small computers with a CPU cache may exist in the same LAN. Hence, whether the smaller computers decide to share or not to share their computational workload with the large and more powerful

computer will also depend on the workload at the large computer.

In this research, both theoretical and experimental (simulation) methods of analysing the CPU cache dual processor distributed computation system to determine some of the important performance measures have been employed. Some of these performance measures that are relevant to the CPU cache system are scheduling time, CPU utilization, CPU throughput, CPU queueing time, input-output handling and the average channel delay. These performance measures are then used to characterise the computational workload at the large computer in order to determine the system capability for distributed computation within the LAN.

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 COMPUTER TECHNOLOGY TREND

Ever since 1945 when the first vacuum tube computer was built there has been a series of radical technological breakthroughs in electronics and computer hardware. With each major technological breakthrough a new and more powerful generation of computers based on the new electronic devices replace the previous generation of computers that were built using the older generation of electronic devices.

The first generation computers of the 1940s used tens of thousands of electronic valves, cost millions of pounds and were bulky enough to fill a large room (ref. 1.1). Such computers included EDSAC, EDVAC, LEO and UNIVAC 1. The second generation computers of the 1950s soon replaced the first generation computers and these used transistors in their hardware. These were still expensive but were less bulky and included such computers as the ATLAS and the IBM 7000 series of computers. They were soon to be replaced in the 1960s and early 1970s by yet more powerful and more compact third generation computers which were built using the then technology of small scale integration (SSI) and medium scale integration (MSI). Examples of these include the ICL1900 series and the IBM360 series computers. The computers of the late 1970s and early 1980s are essentially the fourth generation computers and these are built using the sophisticated microelectronic technology of Large Scale Integration (LSI) and the very large scale integration (VLSI) in which hundreds of thousands of electronic components can be packed on one single integrated circuit (IC) measuring only a few millimetres square. Because of this VLSI technology and mass production, relatively small but quite powerful computers can be produced abundantly and cheaply.

Despite their small size, power and the sophisticated microelectronic technology on which they are based, today's fourth generation computers are essentially the same model computer as the first generation computers in that they are all based on the John Von Neumann model of computation (ref. 1.2). In the Von Neumann model of a computer the computer program instructions are executed strictly sequentially and hence there is little opportunity to employ the large numbers of processors, which can be achieved with VLSI technology, to gain great processing speed if the computer program instructions are inherently parallel. The fifth generation computers of the future will most probably be essentially non Von Neumann and hence very powerful and very fast (ref. 1.3, 1.4).

1.2 I.C. TECHNOLOGY TREND

A computer is a complex system incorporating diverse technologies. Typically, electronic technology is used for the computation, magnetic for long-term storage, and electromechanical for input and output. The evolution of computer structures usually correlates with that of the available technology. The electronic technology has been the most dominant factor in the evolution of computers. Among the technology dimensions are the generation, component complexity and date.

The transistor and the integrated circuit (I.C.) have had a profound impact on the structure of computers. Further, the proliferation of the computer structures built from these technologies has provided enough data points that several interesting digital IC generations and trends can be discerned as shown in fig. 1.1 and fig. 1.2 (ref. 1.5, 1.6).

One of the dominant features in the design of digital ICs has been the collection of basic logic primitives (AND, OR, NAND, etc.) and sequential circuit components (flip-flops, registers, etc.) to build a single IC.

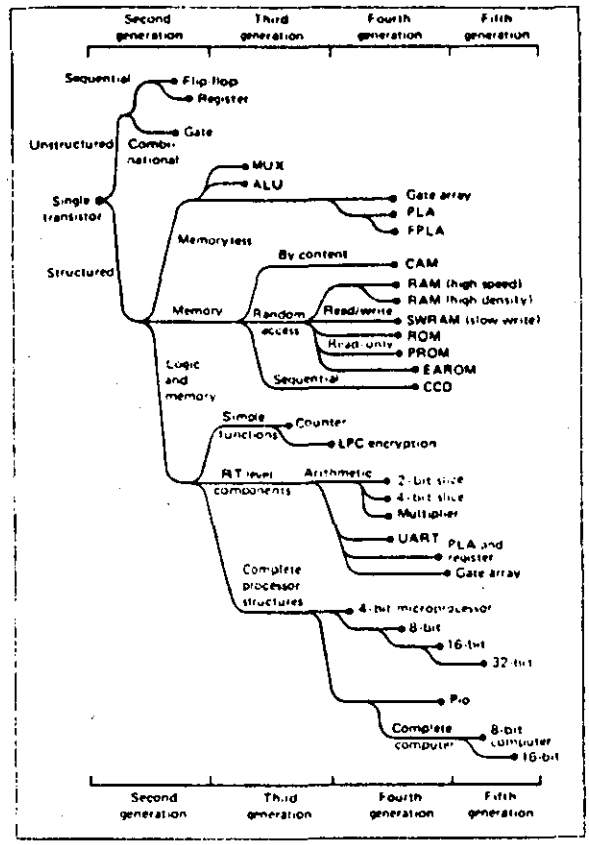


Fig. 1.1 Generations of Digital I.C. Functions

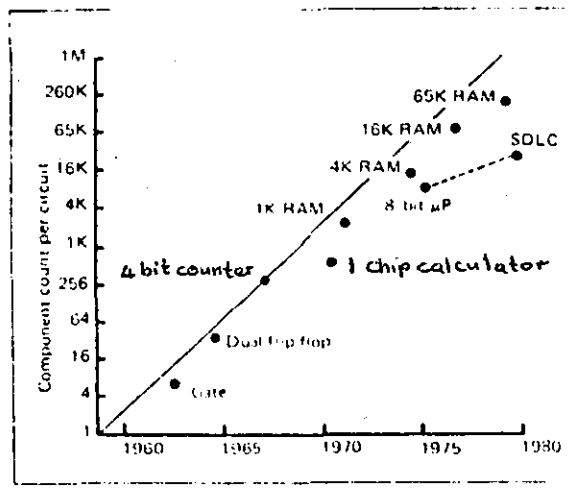


Fig. 1.2 Rate of Increase of the Component Count per I.C.

Further levels of integration led to the emergence of the SSI, MSI, LSI and the VLSI. In this way great reductions in size of component modules was possible. But this had the inherent drawback that the component modules contained a wide variety of functions and were thus specialized. Without well defined functions such as addition, multiplication, etc., the semiconductor technology cannot provide high density products in high volume because there are few large-scale, general-purpose universal functions. These problems have so far been reduced by the two methods of customizing LSI logic: programmable logic arrays (PLA) and the gate density.

Gate density has increased from about 12 in the SSI to hundreds of thousands in the VLSI. As the densities began to approach 100 gates, the construction of complete arithmetic units on a single chip IC became possible (e.g. the TTL SN 74181 ALU unit) (ref. 1.5, 1.6, 1.7). In the 1980s ICs may be expected to reach a packing density of one million gates on a single chip and perhaps ten million gates by the 1990s. A gate is built with from 2 to 6 or more transistors. Z8000 microprocessor chip has been estimated to have 5833 gates and 17,500 transistors (ref. 1.8). The four major packing densities that have been expected to characterise the design and production of IC chips is as shown below, (ref. 1.9, 1.10)

IC chips	Number of electronic devices (roughly)	Period (approx.)
LSI	50,000	1977-1984
VLSI	100,000	1979-1987
VVLSI	1,000,000	1985-1995
VVVLSI	10,000,000	199X-20X1

These values assume that the device count will continue to grow, roughly doubling every 12 to 18 months as has been the case for over 20 years.

A typical small computer might be built with from 1 to 100 or 1000 IC chips, a medium-size computer from 2000 to 7000 and a large computer from 10,000 to 100,000 or even more.

1.3 DISTRIBUTED COMPUTATION

A Multicomputer is a system with more than one computer. Multicomputer networks can be configured so that:

a) Each computer works on a separate problem

b) Each can multiprogram several problems, possibly including interactive terminal-based users and some background programs

c) All can work on the same problem, but essentially by first subdividing that problem into smaller problem pieces so that each can work separately.

d) All can work on the same problem, with increasingly closer coordination and interconnection.

In a distributed computation environment a single user problem (or task) is considered to be composed of many smaller subproblem pieces which are then distributed to and processed by two or more different computers or processors configured in a network. Such a mode of distribution of a computation can be seen to cover a wide spectrum ranging from the loosely coupled load and resource sharing networks such as the ARPANET (ref. 1.11, 1.12, 1.13) in which interaction is infrequent to the very tightly coupled multiprocessors and data flow computers in which interaction is on an instruction by instruction basis. Thus, there is a continuum from processors sharing a common memory (the tightly coupled

multiprocessors) to processors communicating via pre-established message-passing protocols but cooperating on one task. One of the major disadvantages of such message-passing protocol systems is that they are very slow and it may take them thousands of times longer to send a piece of data to a processor than it takes that processor to execute an instruction or to process that data.

The general-purpose single-CPU serial computers on which the loosely coupled computer networks are based execute a program one instruction after the other, in strict serial fashion. With today's technology it takes them about 1μs to fetch and then execute a single instruction. The fastest such computers can take 10 ns. It can be expected that even faster cryogenic computers will easily execute instructions within 10 to 100 ps range (ref. 1.14). But then there will no longer be any possibility of still further speeds since the absolute limit of the speed-of-light barrier will have been reached (light travels at about one foot per nanosecond). But even at these instruction execution speeds, there are still many problems where even the fastest single-CPU serial computers are hopelessly slow. Such problems include:-

a) Image processing

b) The perceptual recognition of and interaction with objects in motion

c) Handling and accessing of very large bodies of information

d) The modelling of 3-D masses of matter in order to predict weather, earthquake, or other large scale phenomena.

e) Modelling and development of intelligent thinking systems

Hence, the trend for computer design will be increasingly toward large configurable arrays and networks of many computers, all working tightly together in a parallel-serial fashion (ref. 1.15, 1.16).

Distributed Computation can be considered to be a form of mapping of the problem structure to the computer network structure. In this way it may be characterised into two main categories: reconfigurable and non-reconfigurable computation.

1.3.1 Reconfigurable Distributed Computation

Reconfigurable distributed computation includes the tightly coupled multiprocessors and data flow computer architectures in which the mode of computation can be described as fine grain. In general, the multiprocessors and the data flow networks can be considered to be distinct subcategories within the tightly coupled computation.

1.3.1.1 Multiprocessor Systems

A multiprocessor is a system with more than one processor. In a multiprocessor environment multiple processes are resident in the primary memory, all in stages of computation, and also intercommunicate via the shared memory. The main purpose of the multiprocessor configuration is to improve not only the individual program performance but also the system throughput by exploiting the parallelism inherent in problems and their algorithms. Four major configurations of processors can be identified (ref. 1.17, 1.18, 1.19, 1.20, 1.21):

a) The single-instruction single-data stream (SISD). This is the traditional single-CPU serial computer arrangement which has only one processor working on one set of data and executing instructions strictly according to the Von Neumann model.

b) The multiple-instruction single-data stream (MISD) system in which the processors are pipelined. In such a pipeline, each of a number of processors continually executes the same instruction as data flows through the pipeline, each processor executing a different step in the longer sequence of instructions.

c) The single-instruction multiple-data stream (SIMD) system in which the processors can be arranged in an array. In such an array, all the processors execute the same instruction, each on a different set of data.

d) The multiple-instruction multiple-data stream (MIMD) system in which each of the different processors executes a different sequence of instructions on a different set of data.

A single instruction means that the SIMD systems are highly synchronised and hence virtually all contention among the processors is eliminated. On the other hand, a single data stream means that the MISD systems load instructions only once for very efficient pipelining. The efficient operation of these systems is under the supervision of the control unit. The SISD and SIMD systems have a single control unit while the MIMD systems usually have a control unit for each processor. In this way the parallelism inherent in the problem and its algorithm can be mapped to the configuration of the processors.

1.3.1.2 Data Flow Systems

The concept of the data flow computation systems has come about because of the evolution of the VVLSI and VVVLSI technology in which many IC chips, each with hundreds of thousands of processors, can be reconfigured as demanded by the algorithm that defines the problem to be solved, (ref. 1.15, 1.16). The main aim here is to develop such multicomputer architectures that put large numbers of processors into appropriately coordinated interaction with one another, in such a way that they all cooperate and work efficiently together to solve a single problem. In this way these reconfigurable multicomputer networks will mirror the algorithms' information flow, much like the way the raw materials flow through assembly lines to

facilitate the production of a single finished product. Different algorithms will call for different reconfigurations of the processors. Thus, the aim is to design the algorithm, program, language and computer in a single integrated architectural exercise so that:-

a) The algorithm will mirror the flow of information through the network of processors

b) This flow will in turn determine the structure that the set of processors should take to efficiently effect the algorithm

c) This structure in turn will determine the architecture of the network of processors that executes the program.

d) The operating system will then either choose the appropriate architecture or it will actually form and reconfigure that architecture out of the large general-purpose network of the hundreds of thousands of processors or individual devices at its disposal.

In this way the flow graph that represents the algorithm's flow of information through the structure of processors (ref. 1.22, 1.23, 1.24) that transform the input information into the solution to be output is mapped into the physical processor-memory graph structure that will execute those processes (i.e. isomorphic hardware networks).

Hence the data flow model for reconfigurable distributed computation is a paradigm for highly distributed computation in which the interaction during the computation is very fine grain. This mode of interaction is not based on the conventional Von Neumann model of computation in which the speed of the instruction processing is dependent on the operation of the program counter and thus not limited by the basic instruction cycle. In a fine grain model of computation each node of the data flow graph may represent one machine instruction or a small group of such instructions.

If each node corresponds to a single machine instruction then it can be seen that the collection of the processors in fact are configured to run a single program and the interaction of the processors is on an instruction by instruction basis.

A major characteristic of the data flow model of distributed computation is that there are no variables and no memory locations in which to store the results (ref. 1.15). Instead values and partial results are represented by packets that are transmitted between the processing elements. Each processing unit carries out some function on the values at its input and produces an output result. Thus each function depends strictly on its inputs, and not on any global variables. As soon as the input packets have arrived, each processing unit may begin its computation. Hence there is no program counter and no explicit artificially forced sequencing of computation, other than that implicit in one calculation depending on the result of a previous one. As a simple example, consider the evaluation of the expression (X+Y)/(A.(B+1)). A tree algorithmic representation might be as shown in fig. 1.3. In this way the parallelism inherent in a problem can be automatically mapped onto the configuration of the processors, and great speed of computation can be achieved. In large and complex problems even greater speeds of computation can be achieved because of the possibility of the presence of many unrelated expressions which can be done in parallel.

1.3.2 Non-Reconfigurable Distributed Computation

Non-reconfigurable computation is based on the processing speed and power of the traditional general-purpose single-CPU serial computers. The organisation of such computers is as shown in fig. 1.4. The computer's single processor accesses a single memory, and inputs and outputs to and

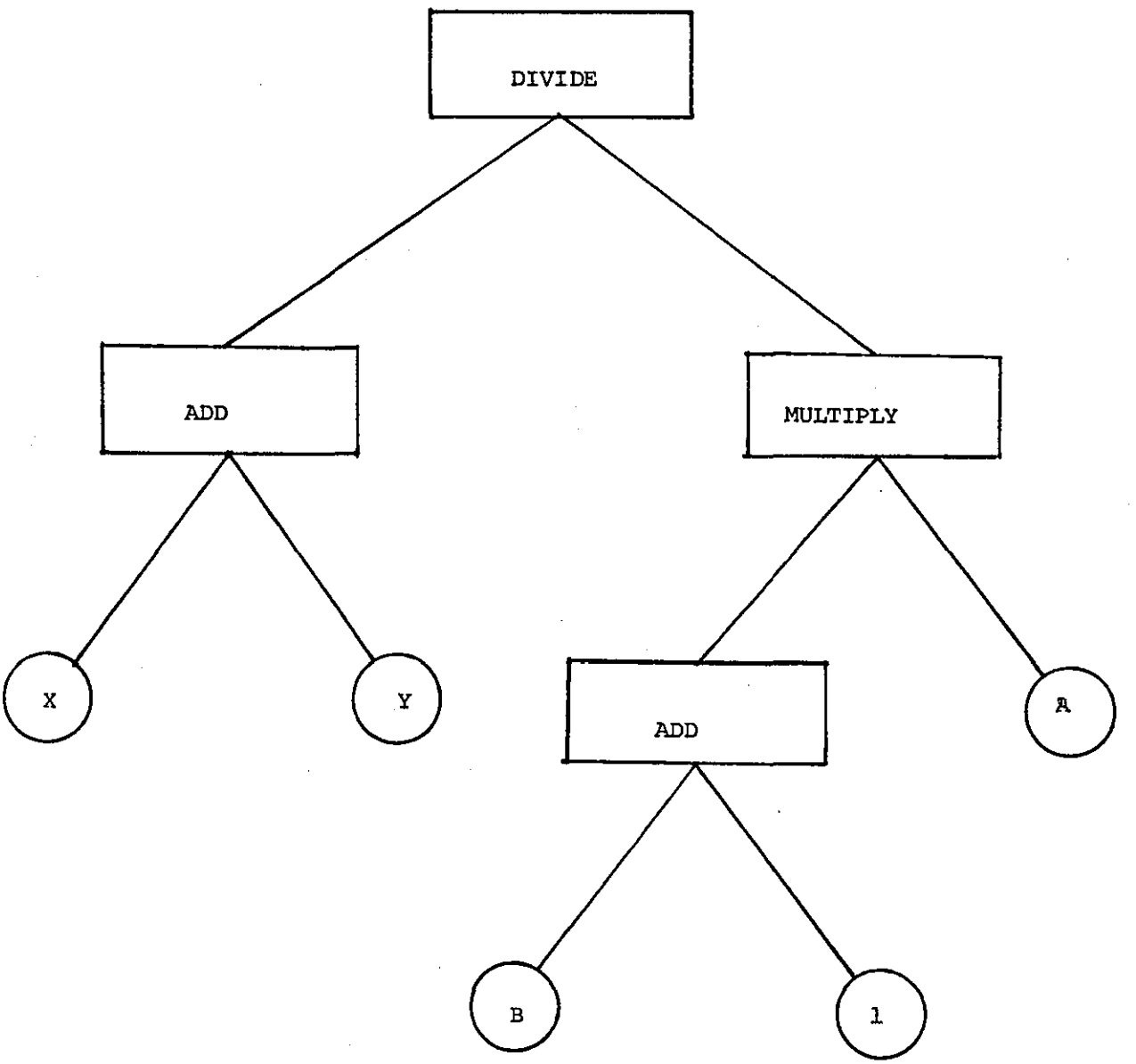
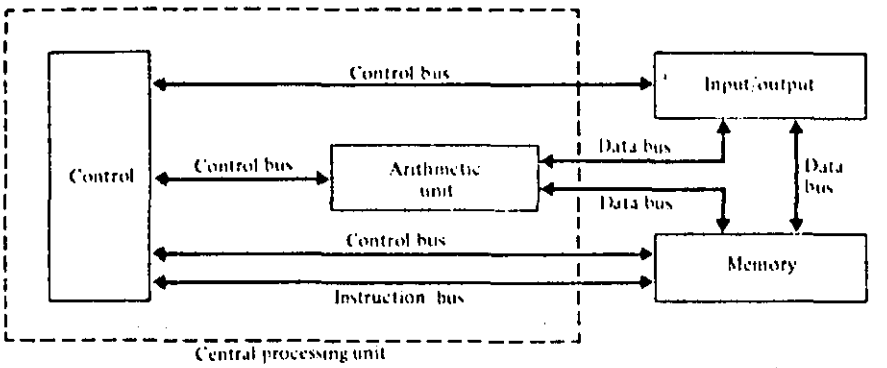
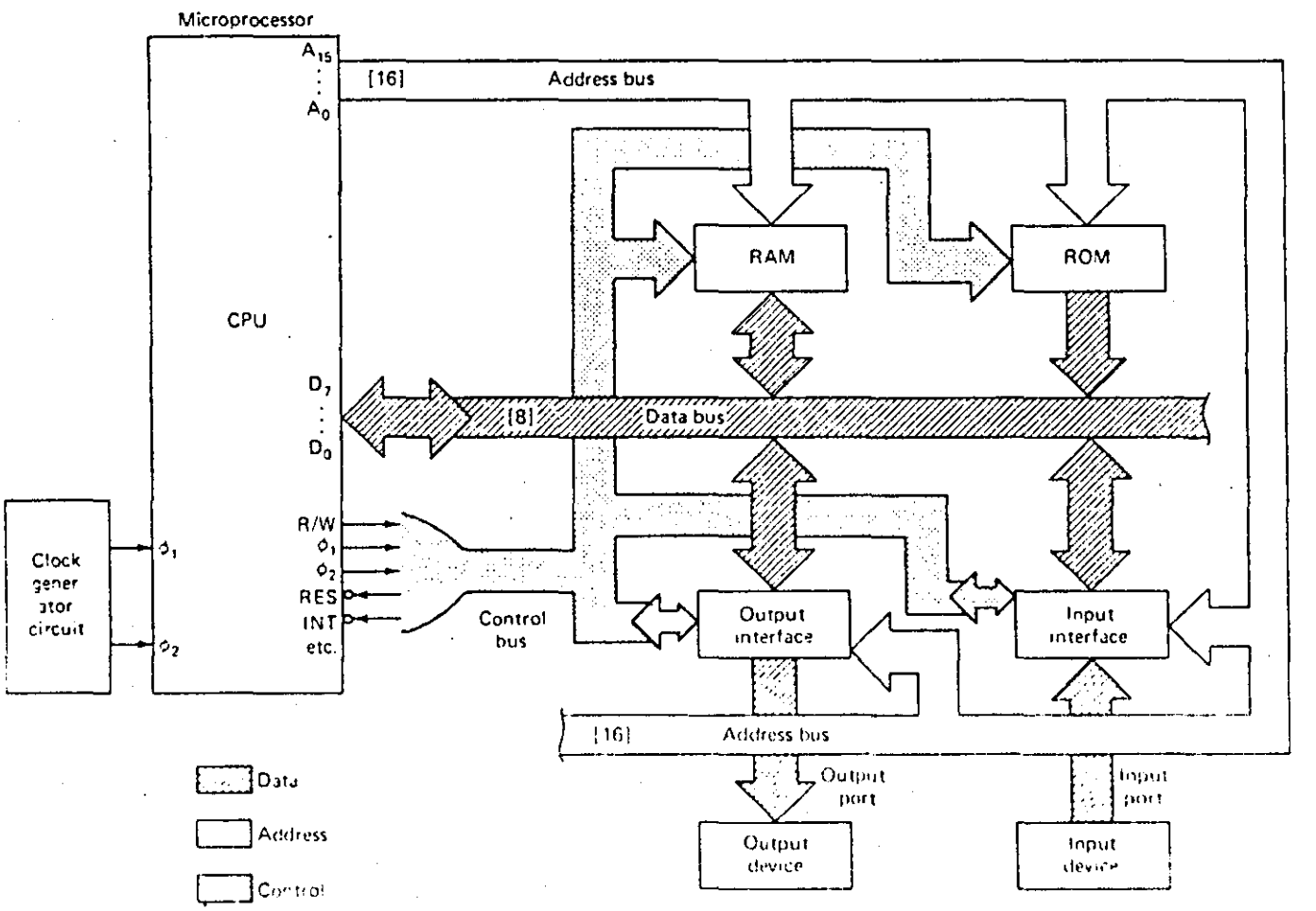


Fig. 1.3 A Tree Representation



(a) Organisation of a small computer



(b) A Typical 8-bit Microcomputer Structure

Fig. 1.4 Characteristics of Small Computers

from the outside world. The CPU comprises the arithmetic and logic unit, ALU, which is often made up of a whole set of simple special purpose processors, the control unit, CU, together with its high-speed registers. More specialized hardware such as for floating point arithmetic capability are sometimes added to improve the speed performance of the computer. Often there are several input/output (IO) devices, and a hierarchy of successively slower and larger memories. But the computation is done by the single CPU, using data and programs stored in its main memory.

The consideration of such autonomons and interconnected computers sharing a single computation as described above imply the existence of a computer network. Computer networks can be considered to have emerged from the convergence of computer and telecommunication technologies, fig.1.5 and 1.6 : two technologies with quite distinct histories and traditions.

As a consequence of the computers becoming smaller, cheaper and more numerous the need to interconnect them together into networks (ref. 1.25) has continued to increase. Today's microcomputers have processing speeds, instruction sets and memory management capabilities comparable to many medium power minicomputers. The small computers tend to have less complex software while the larger and more powerful computers are usually multi-accessed, time-shared, multiprogrammed and have large and complex software. The software of these large mainframe-like computers is often written in an exotic language, is machine dependent or is embedded in a complex web of libraries and thus requiring special system calls and other non-portable environmental features. They also tend to have very specialised hardware. Hence part of the reason for the need for the computers to internetwork arises because of the prospects for resource

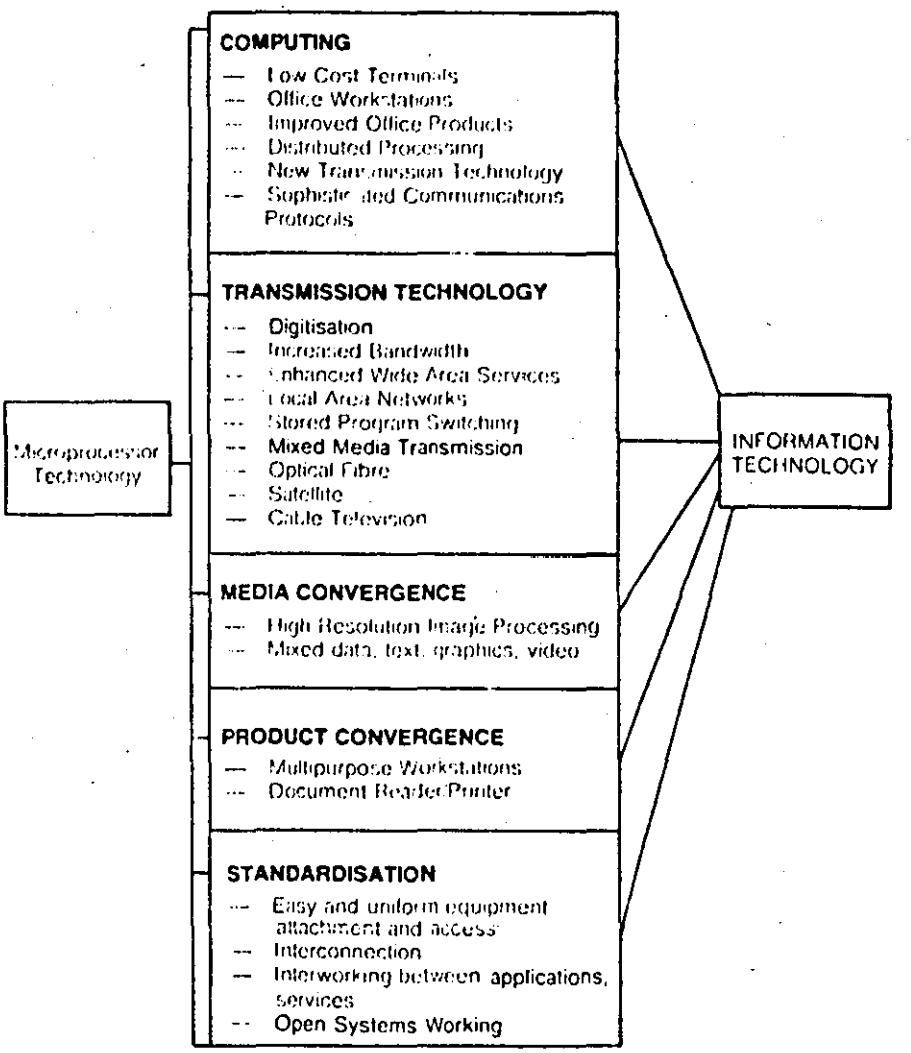


Fig. 1.5 Convergence of Computing and Telecommunications

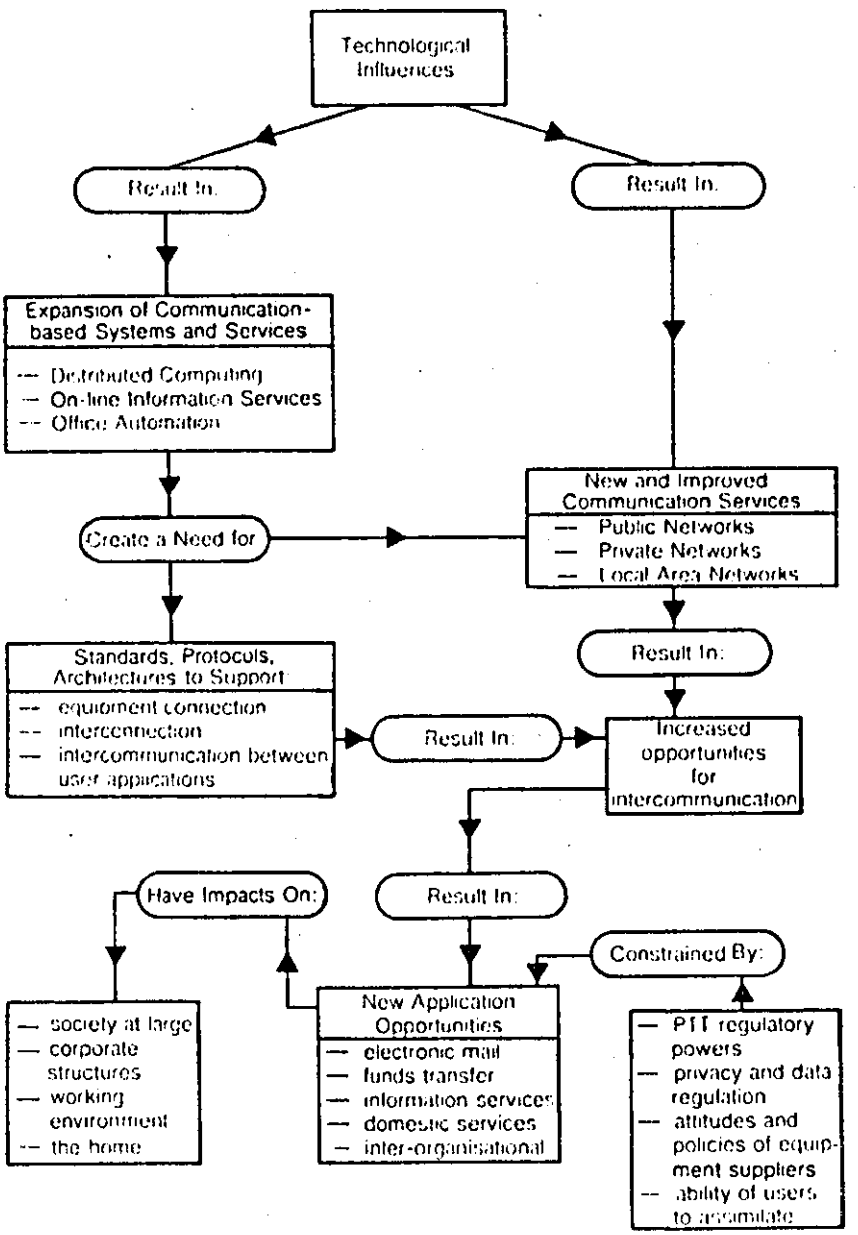


Fig. 1.6 Computing and Communications Interdependencies

sharing, load sharing and to distribute the computing power existing within the network. The ARPA network (ref. 1.11, 1.12, 1.13) is the largest long-haul computer network in existence and it interconnects hundreds of computers geographically distributed across several continents. Another major factor which has contributed to the idea of computer networking is that the cost of the computation has progressively continued to fall over the last decade and has fallen to the point where the cost of the computation is cheaper than the cost of the communication facilities. From this it can be seen that the effect of computer networks is to reduce the effect of geography and distance. However, in a distributed computation environment it is necessary that the total delay in the communication network be very small. Fortunately, in a wideband broadcast-type LAN considered in this thesis very small delay can be achieved.

Several non-reconfigurable models of distributed computation are now examined.

1.3.2.1 The Hierarchical Model

In the hierarchical distributed computation model the computers in the network are functionally arranged hierarchically as shown in fig. 1.7. This is a logical arrangement in siatuations where the distribution of the entire computation is structured hierarchically. At the lowest level small microcomputers execute certain low level functions locally, such as computations and transactions, and then pass the results up to the next level of more powerful computers such as minicomputers. Some or even all the computations and transactions eventually reach the highest level of large mainframe-like computers which in turn may have access to on-line files or data-bases. This top level computer performs its own type of processing on its own transactions using the data and results from the lower computation levels. In this way each level processes a different level of detail.

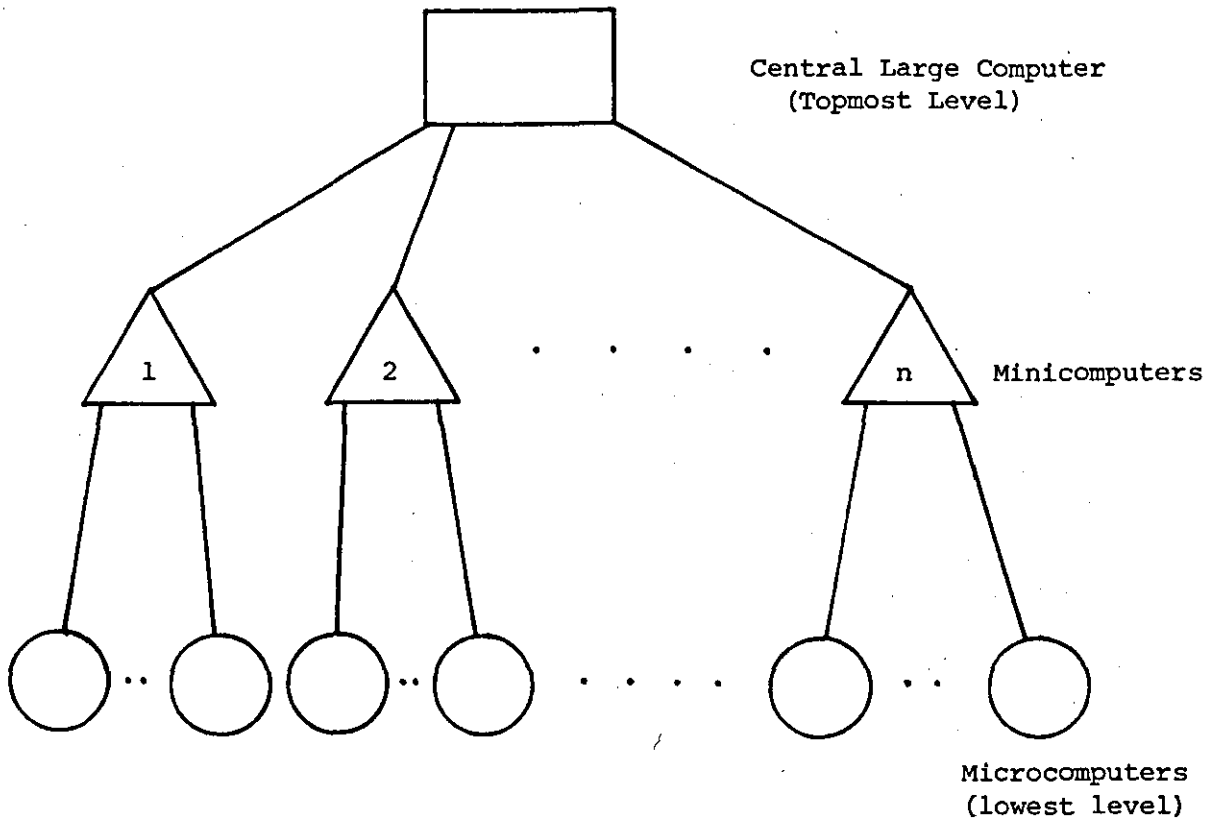


Fig. 1.7 Hiearchical Computation Model

A factory complex such as for production or process control may support such a hierarchical structure. The many monitoring instruments and sensors taking readings in an industrial or chemical process may be under the control of microcomputers. Minicomputers in turn analyse data and supervise the performance of the microcomputer by setting switches, operating the relays, adjusting the valves and regulating temperature and speed. Thus, the minicomputers control the process and also provide very fast response to critical changes. The highest level mainframe computer uses data and computation results from the minicomputers to enable it to perform high level functions such as process optimization, quality control, planning, management control and general data processing.

1.3.2.2 The User-Server Model

In the user-server computation model each individual computer in the network has enough computing power to serve the local site. This adequate computing power may be provided by a personal minicomputer with limited local disk storage. In such a set-up (ref. 1.26) specialized high quality or mass storage facilities may be located elsewhere on the network as shown in fig. 1.8. In this way it is possible to achieve resource sharing of large disk servers, file servers, data-base servers, expensive or unique high resolution phototypesetter, high quality printers or graphics facilities which may be centrally located due to economy of scale.

1.3.2.3 The Pool Processor Model

Computing requirements tend to be very bursty in nature. In such a bursty computation environment a relatively short but intensive period during which the computer is in use is usually followed by a relatively long period when the computing power is idle. In order to reduce this expensive CPU idle time, it may be found necessary to pool the processors in a central area and provide the various geographically distributed

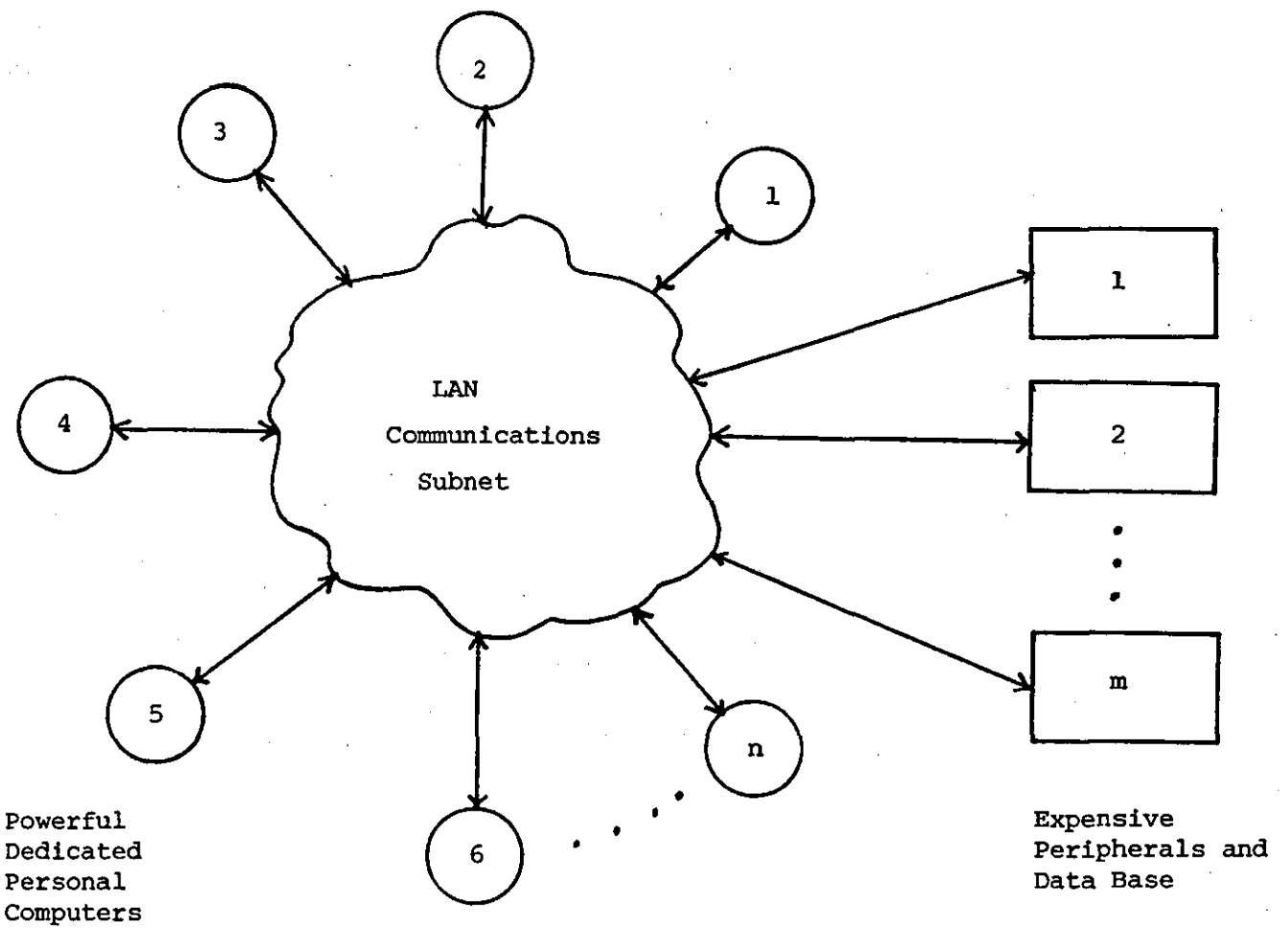


Fig. 1.8 User-Server Computation Model

users with non-intelligent terminals (ref. 1.27). Thus instead of providing each user with enough computing power, the computing power is centralized and accessed by any user who requests it as shown in fig. 1.9. In this way diverse and specialized CPU can be shared. In addition to pooling just the processors, other expensive or special high quality peripherals and data base could also be pooled (ref. 1.28). But the major disadvantage encountered in the pooled resource models such as the pooled processor and the user-server models is the complexity of the scheduling algorithm needed (ref. 1.29), to determine the properties of the computation such as floating point or memory requirements, and also the possibility of deadlock.

1.3.2.4 The CPU Cache Model

In the CPU cache model of distributed computation a smaller computer decides to share its workload with an existing more powerful and centrally located computer in the network. Such a situation can easily obtain in an organisation whose workload exceeds the capacity of the largest existing CPU. CPU cache in this context is used analogously with the memory cache in which there are two levels of the main memory: a small fast memory and a large slow memory. In such a set up the cache algorithm attempts to keep the most heavily used data in the fast memory, to reduce the memory access time. Many such users with a CPU cache problem and existing within a geographically small area may be connected to several existing powerful computers in the same network, as shown in fig. 1.10.

The CPU cache model of distributed computation is about halfway that of the user-server model in which each user has enough local computing power capability provided by a personal minicomputer and the pool processor model in which each user has limited local processing capability (ref. 1.30).

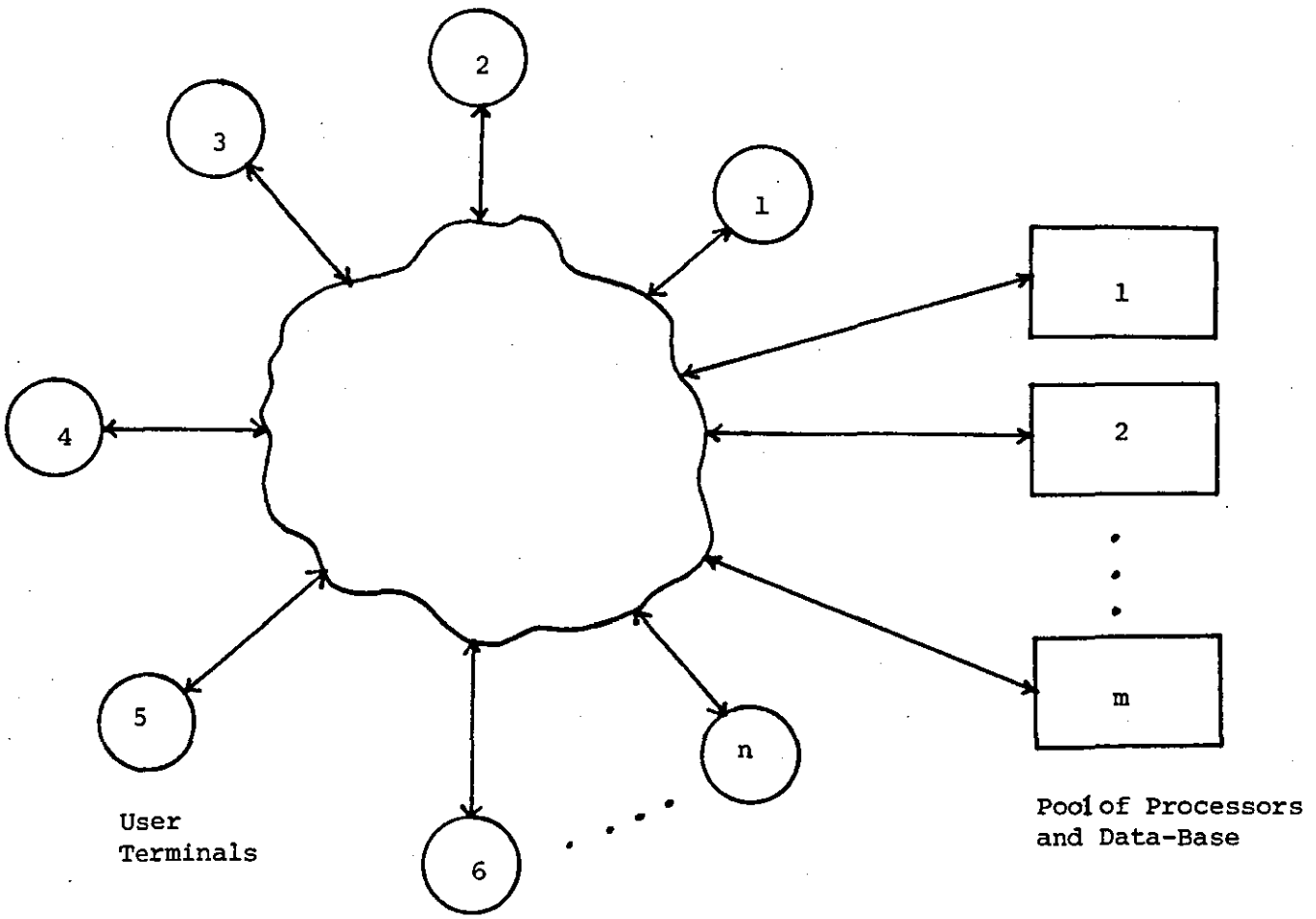


Fig. 1.9 Pool Processor Computation Model

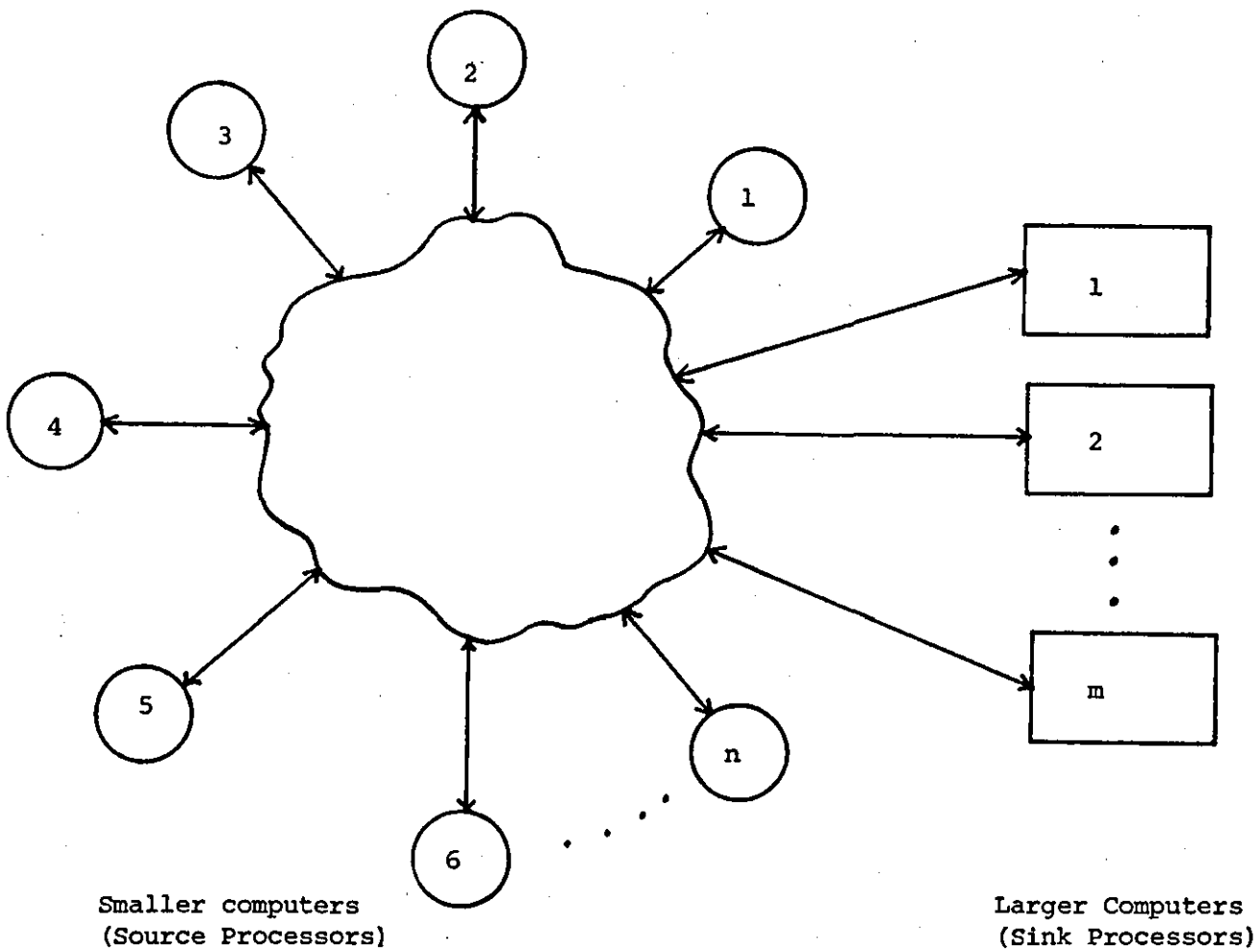


Fig. 1.10 CPU Cache Computation Model

In the CPU cache model each user has a small general-purpose computer which is capable of running a wide variety of user programs completely in a stand-alone mode. Such a definition of these smaller computers include the 8-bit and 16-bit microcomputers and some minicomputers. Even though they are relatively small, these smaller computers may still have versatile architecture and powerful instruction sets to support massive computational power such as floating-point capability, input/output devices such as high-speed printers, plotters, microfilm recorders and magnetic tapes as shown in fig. 1.11.

Hence in the CPU cache model the workload of the smaller computer is effectively partitioned into two portions and processed thus by the two computers. The decision to run a particular portion of the computation on one computer or the other is based on the relative suitability of the two machines and may depend on such factors as:

a) The relative processing costs of the two machines

b) The communication bandwidth between the two machines

c) The current workload

The current workload at the large computer is a very dominant factor in the decision where to run a particular computation. When the workload at the large computer is light then it can be expected that a large portion of the partitioned workload is sent to run there because the machine there is faster.

This thesis is based on the CPU cache model of distributed computation. The CPU cache model is based on the relative performance of the computers sharing the workload. This relative performance has so far been distinguished by the terms small and large computers. The terms small and large are now further clarified.

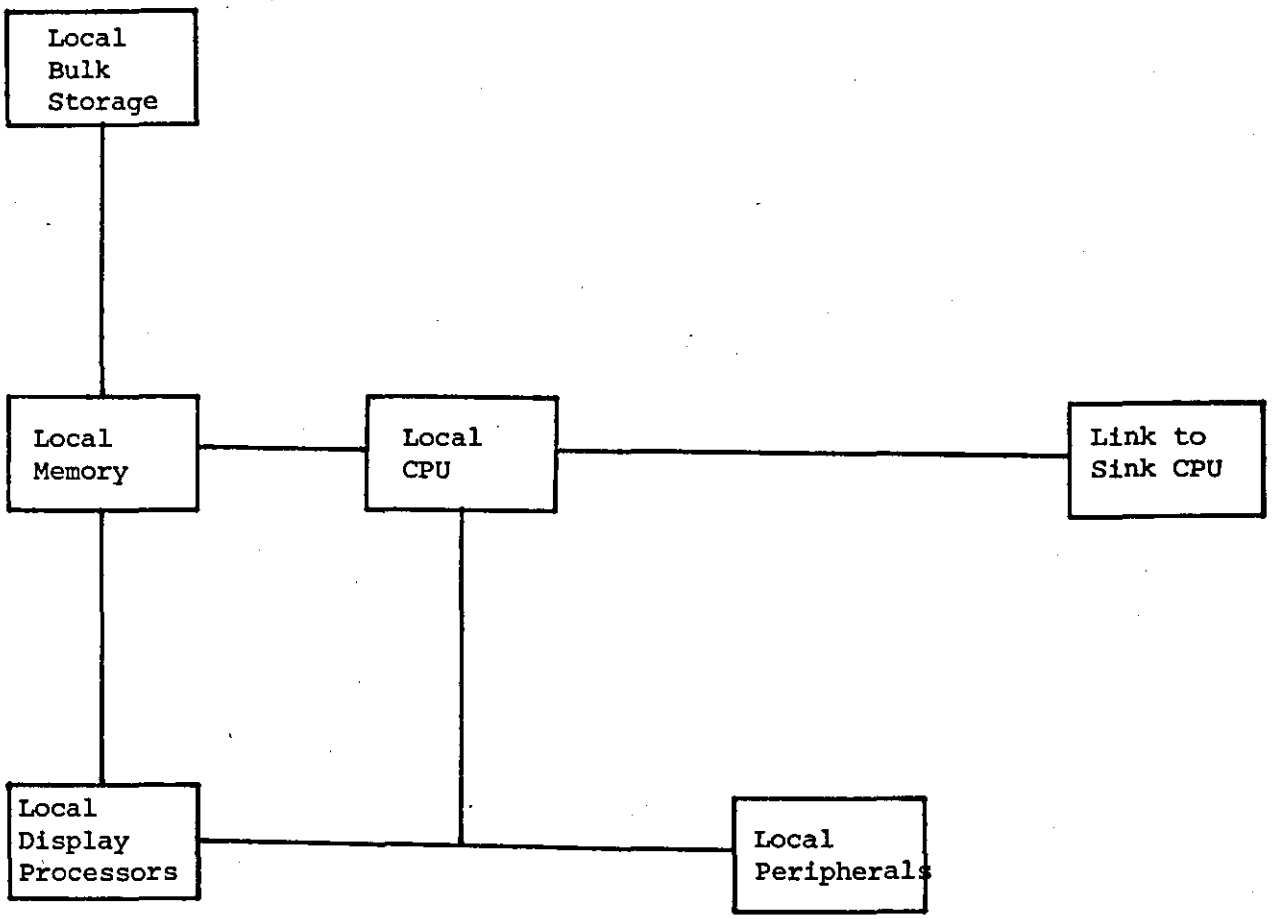


Fig. 1.11 Small Computer and Peripherals

1.4 SMALL AND LARGE COMPUTERS

On the one hand the distinction between the small and large computers to explain the existence of a CPU cache is not obvious. The adjectives small and large are relative terms. A computer can be small physically but quite powerful in terms of its instruction set capability. Micro-computers may be described as small computers but their software may enable them to outperform many older generation computers which are physically large. Hence the level of technology is an important factor in this classification. But many factors need to be taken into account to facilitate a valid distinction between the small and large computers. One can classify computers into the four main categories shown in table 1.1.

On the other hand the level of technology achieved makes this classification even more difficult (ref. 1.5). When an improved basic technology becomes available to the computer designers, there are four paths that the designs can take to incorporate the technology:

a) Use the newer technology to build a lower cost system with the same performance and thereby attract new applications

b) Hold the cost constant and use the technological improvement to get an increase in performance. This approach provides a growth in performance and quality at a constant price.

c) Push the design to the limits of the new technology, thereby increasing both performance and cost. In this approach the new technology is used to build the most powerful machines possible and thus enabling previously unsolved problems to be solved and in so doing advance the state of the art.

d) Find a completely new structure using the computer as a basic architype (e.g. the calculator) so that the design can be considered to be off the evolutionary path.

Characteristic	Programmable calculator	Micro-computer	Mini-computer	Midi-computer
Typical number of bits per word	32–64	4–16	12–32	16–64
Function	Dedicated	Dedicated or general purpose	General purpose	Dedicated or general purpose
Speed	Very slow	Slow to fast	Fast	Fast
Required user understanding of machine	Very limited	Extensive	Fair to substantial	Limited
Typical high-level language	BASIC implemented in hardware	PL/M PASCAL	BASIC FORTRAN ALGOL	ALGOL BASIC FORTRAN COBOL
Typical programming methods	Manually from keyboard	Assembly language	Assembly or high-level language	High-level language
Typical applications	Calculations	Device control Accounting Replacement of digital logic	Problem solving Process control Device control	Solving large problems Systems control
Cost	Low	Very low to low	Low to medium	Medium to high

Table 1.1 Comparison of Typical Characteristics of Computer Classes

	8008	*8080*	*8085*	*8086*
Number of instructions	66	111	113	133
Number of flags	4	5	5	9
Maximum memory size	16K bytes	64K bytes	64K bytes	1M bytes
I/O ports	8 input 24 output	256 input 256 output	256 input 256 output	64K input 64K output
Number of pins	18	40	40	40
Address bus width	8†	16	16	16†
Data bus width	8†	8	8	16†
Data types	8-bit unsigned	8-bit unsigned 16-bit unsigned (limited) Packed BCD (limited)	8-bit unsigned 16-bit unsigned (limited) Packed BCD (limited)	8-bit unsigned 8-bit signed 16-bit unsigned 16-bit signed Packed BCD Unpacked BCD
Addressing modes	Register‡ Immediate	Memory direct (limited) Memory indirect (limited) Register‡ Immediate	Memory direct (limited) Memory indirect (limited) Register‡ Immediate	Memory direct Memory indirect Register Immediate Indexing
Introduction date	1972	1974	1976	1978

† Address and data bus multiplexed

‡ Memory can be addressed as a special case by using register M

Table 1.2 Comparison of Typical Features of Microcomputers

An examination of the use of new technology for constant cost and constant performance over a period of time leads to an economic view that computer classes can be distinguished by cost and grouped into the following four main categories:

a) Programmable calculators (or monolithic microcomputers)

b) Microcomputers

c) Minicomputers

d) Maxicomputers (or mainframe computers)

as shown in fig. 1.12. Hence, the measure used to define a new class is cost, whereas the measure used to define an established class is performance. This is primarily because once a new class has become established in the market, the users become familiar with what computers and what class can be used for their application, and hence tend to characterise that class on a performance basis. The characterisation of existing classes on a performance basis is important because at each new technology time, performance increases by one category, and the mini-computer performance becomes available on a microcomputer, for example. Hence by considering the effect of technology upon the computer classes using new technology for constant cost and constant performance the following conclusions may be drawn:

a) The cost declines and this creates new classes of computers

b) The new classes become the established classes

c) The established classes become encroached upon.

Computer types can also be classified on the basis of their bytes of virtual address space. Several computer space dimensions are roughly correlated with the number of bytes in the virtual address. A larger virtual address usually means:

a) Wider instruction words to hold larger virtual addresses. These wider words imply wider memories and data paths, higher CPU-Memory bandwidth, and larger instruction sets

b) Usually, more functionality of the instruction set processor, represented by an ability to support more data-type in hardware

c) Higher costs due to (a)

d) Higher performance to gain economies of scale.

Fig. 1.13 shows this relationship of bytes in the virtual address space over a period of time.

From this discussion it may be seen that the CPU cache model of distributed computation can be used in the sharing of a computation between a variety of computers, such as:

a) Microcomputers and minicomputers

b) Microcomputers and mainframe computers

c) Minicomputers and mainframe computers

d) 8-bit microcomputers and 16-bit microcomputers

From (d) it may be expected that a mild form of a CPU cache relationship may exist between computers in the same class such as one mainframe computer and another.

A few characteristics of the various computers classes, with the exception of the monolithic microcomputers, that may exhibit a CPU cache are examined below. Monolithic microcomputers (e.g. TMS1000) are primarily single-chip systems incorporating the processor, program ROM, variable RAM, and sometimes dedicated input/output.

1.4.1 The Microcomputer

Very many different microcomputers have been manufactured. Their processing power is primarily determined by the microprocessor on which they are based. They have external RAM and ROM chips and are usually faster

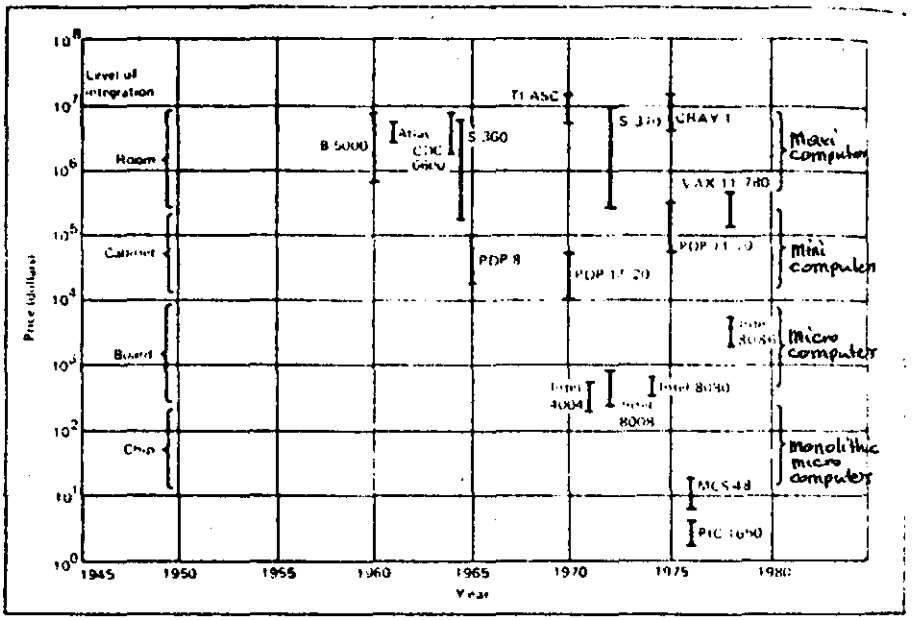


Fig. 1.12 Computer Class as a Function of Price

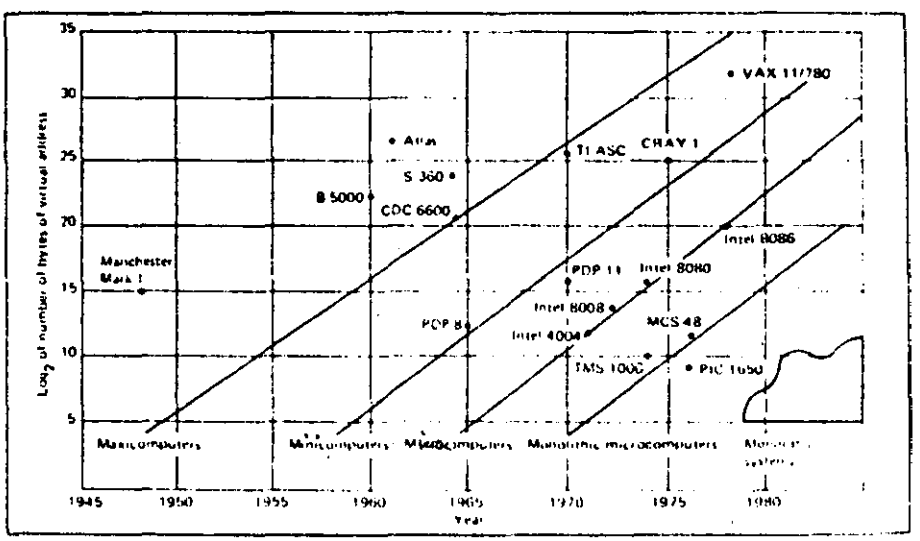


Fig. 1.13 Computer Class as a Function of Virtual Address Space

than the monolithic microcomputers, since the off-chip placement of memory and input/output (IO) frees gates for more complex instructions and wider data paths. Some of the important features of these microprocessors around which the microcomputers are built are briefly examined.

1.4.1.1 The 4040 4-bit microprocessor

The important features of the 4040 processors include

a) 24 registers

b) 4-bit words with 12-bit addresses

c) Use of 2's complement and BCD arithmetic

d) Acceptance of programs written in machine and assembly language

1.4.1.2 The 8080/8085 8-bit microprocessor

The 8080 and 8085 are N-channel MOS, Fig. 1.14, and share the same machine and assembly language. Their other important features include

a) 10 principal registers, including one accumulator

b) 8-bit words with 16-bit addresses

c) 5 flags to show the CPU status

d) Use of 2's complement and BCD arithmetic

e) Acceptance of programs written in machine language, assembly language, BASIC, PASCAL, and PL/M (a subset of PL/1)

f) 111 and 113 instructions respectively

g) Clock frequency is between 1 and 4 MHz

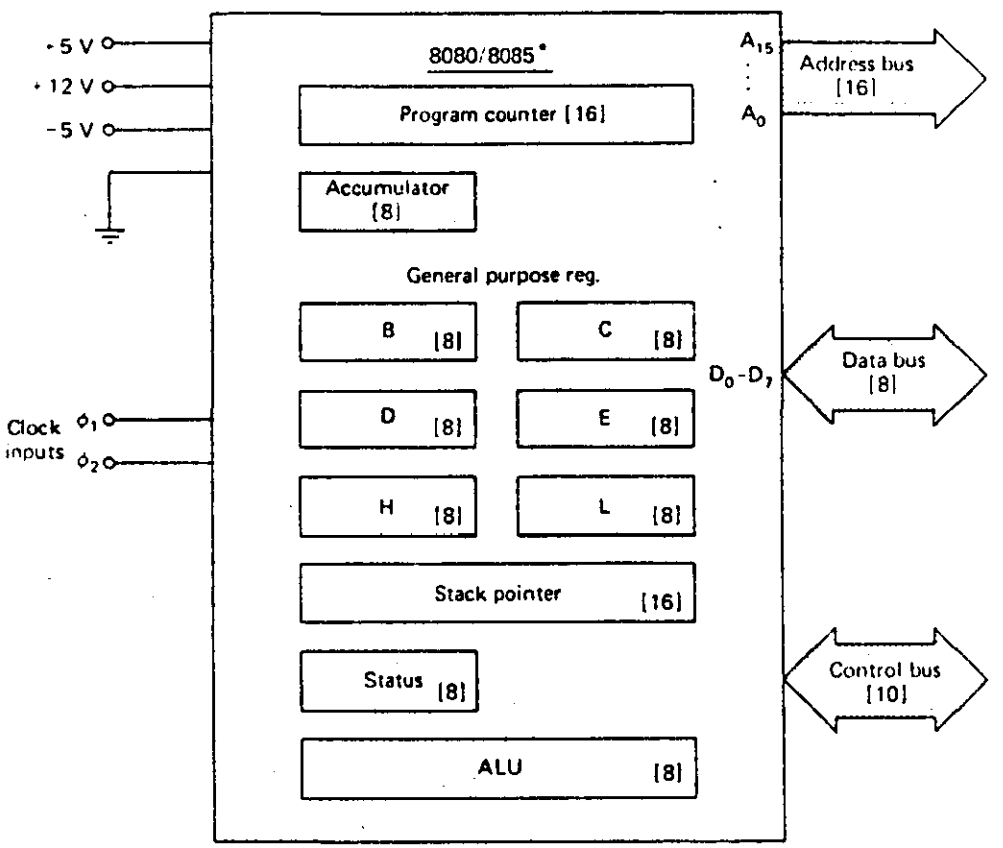
1.4.1.3 The Z-80 8-bit microprocessor

The Z-80 is an N-channel MOS microprocessor, fig. 1.15. The Z-80 includes all the 8080 instructions as a subset and has:-

a) 22 principal registers including two accumulators

b) 8-bit words with 16-bit addresses

c) 6 flags to show CPU status



*The 8085 has on-chip system controller and clock-generating capabilities, uses a single +5-V supply and has more bus control functions and interrupt capabilities.

Fig. 1.14 The 8080/8085 Microprocessor

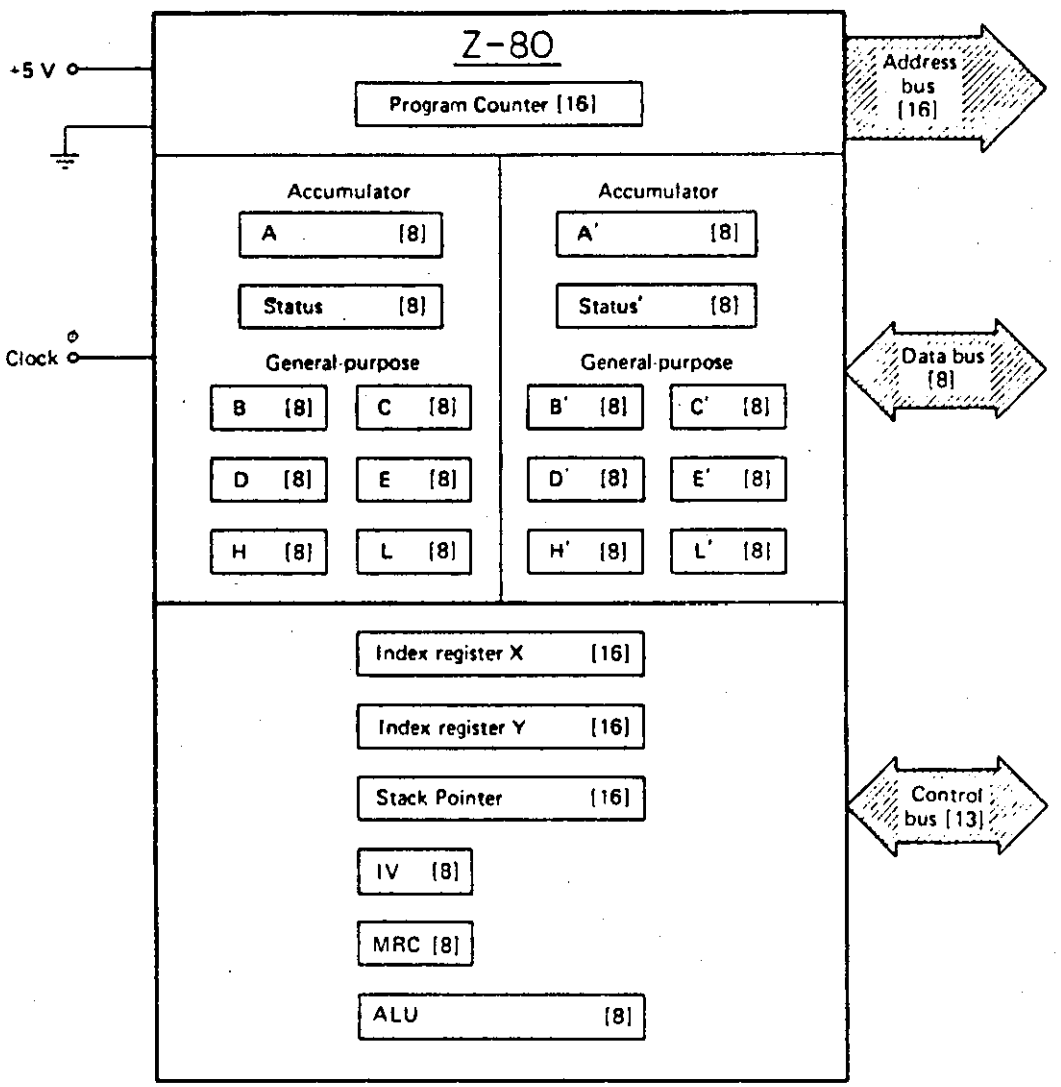


Fig. 1.15 The Z-80 Microprocessor

d) Use of 2's complement and BCD arithmetic

e) Acceptance of programs written in machine language, assembly language, BASIC, PASCAL, PL/M

f) 158 different instructions

g) Clock frequency of 2.5 MHz

1.4.1.4 The M6800 8-bit microprocessor

The Motorolla 6800 microprocessor resembles the 8080/8085 in many respects, fig. 1.16. Its important architectural features include

a) 6 principal registers including 2 accumulators

b) 8-bit words with 16-bit addresses

c) 6 flags to show CPU status

d) Use of 2's complement and BCD arithmetic

e) Acceptance of programs written in machine language, assembly language, BASIC, PASCAL, MPL (a subset of PL/1)

f) 72 types of instructions

g) Clock frequency of 1 MHz

1.4.1.5 The 16-bit microprocessors

The 16-bit microprocessors have evolved in direct competition with the well-established minicomputer classes. Some typical 16-bit microprocessors are the Intel 8086, Z-8000, M68000 and the Texas Instruments TMS9900. The main advantages of the 16-bit microprocessors come from the fact that with 16-bit word size, more powerful instructions can be written which allow for much efficient way to perform powerful computational tasks. Most of the present 16-bit microprocessors have instruction sets which are built around specific microcomputer instruction sets, thus allowing the user to take advantage of all the existing software available for a specific minicomputer. The main disadvantage with the 16-bit microprocessors is that more than 40 pins are used on the microprocessor

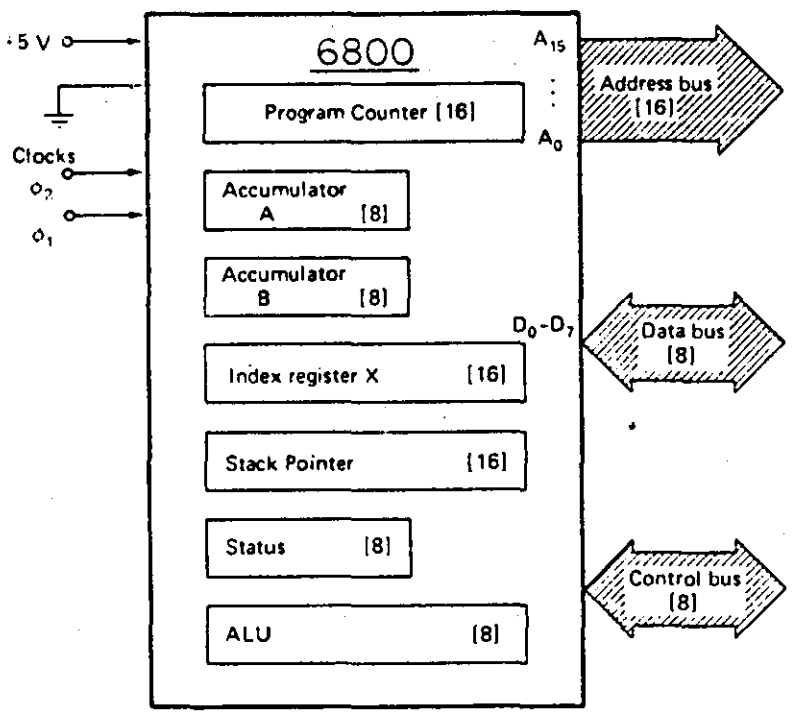


Fig. 1.16 The 6800 Microprocessor

package unless bus multiplexing is used. However multiplexing of bus information increases the system complexity, as far as parts counts is concerned, and this may slow down the speed of operation of the overall system. Tables 1.2, 1.3, 1.4 and 1.5 show a comparison of some of the important features of Intel microprocessors (ref. 1.5).

1.4.2 The Minicomputer

Many different types of minicomputers have been manufactured. Two of the more popular minicomputers are briefly examined.

1.4.2.1 The PDP-8 Minicomputer

PDP-8 is one of the earliest minicomputers and is widely available. The most important features include

a) One accumulator

b) 12-bit words

c) Separate buses for memory and I/O

d) Use of 2's complement arithmetic

e) Acceptance of programs written in machine language, assembly language, FORTRAN, BASIC, PASCAL

1.4.2.2 The PDP-11 Minicomputer

PDP-11 is one of the most popular computers. Some of its most important features include:

a) 8 or 16 registers

b) 16-bit words divided into two 8-bit bytes that can be individually addressed and manipulated. Some models have 32-bit words.

c) A single bus (Unibus) for operations with both memory and I/O devices

d) A processor status register that keeps track of four types of conditions.

e) Use of 2's complement arithmetic

f) Acceptance of programs written in machine language, assembly language, BASIC, FORTRAN, PASCAL

	8008	*8080 (2 MHz)*	*8086 (8 MHz)*
register-register transfer	12.5	2	0.25
jump	25	5	0.875
register-immediate operation	20	3.5	0.5
subroutine call	28	9	2.5
increment (16-bit)	50	2.5	0.25
addition (16-bit)	75	5	0.375
transfer (16-bit)	25	2	0.25

All times are given in microseconds.

Table 1.3 Comparison of Typical Microcomputer Performance

	8008	*8080*	*8085*	*8086*
Silicon gate technology	P-channel enhancement load device	N-channel enhancement load device	N-channel depletion load device	Scaled N-channel (HMOS) depletion load device
Clock rate	0.5–0.8 MHz	2–3 MHz	3–5 MHz	5–8 MHz
Min gate delay† F0 = FI = 1	30 ns‡	15 ns‡	5 ns	3 ns
Typical speed-power product	100 pj	40 pj	10 pj	2 pj
Approximate number of transistors¶	2,000	4,500	6,500	20,000§
Average transistor density (mil² per transistor)	8.4	7.5	5.7	2.5

† Fastest inverter function available with worst-case processing
‡ Linear-mode enhancement load
§ This is 29,000 transistors if all ROM and PLA available placement sites are counted
¶ Gate equivalent can be estimates by dividing by 3.

Table 1.4 Comparison of Typical Microcomputer Technology

	Intel 8008	*Intel 8080*	*National IMP-8*	*Motorola 6800*	*Intel 8085*	*RCA COSMAC*	*Fairchild F-8*	*Zilog Z-80*	*Intel 8086*	*Motorola MC 68000*	*Zilog Z-8000*
Technology	PMOS	PMOS	PMOS	NMOS	NMOS	CMOS	NMOS	NMOS	HMOS	HMOS	NMOS
Number of pins per package	18	18	24	40	40	28, 40	40	40	40	64	48
Instruction time (μs)	12.5	7.5	4.2	2	2	6	2	1	0.25	0.25	0.75
Data-path width (bits)	4	8	2 × 4	8	8	8	8	8	8	16	16
Maximum memory size (bytes)	4K	16K	64K	64K	64K	64K	64K	64K	1M	16M	8M
Register file size	16	7		3	7	16	72	7 (2 sets)	16	16	16
Stack size	3	7	16	In RAM	In RAM			In RAM	In RAM	In RAM	In RAM
Instruction size (bytes)	1-2	1-3	1-2	1-3	1-3	1	1	1-3	1-4	2-6	2-10
Basic instruction-set size	45	66	38	72	111	48	101	89	133	59	110
Number of addressing modes	4	4	Many	7	5	2	3	10	24	10	8
Data-types	Integer, decimal	Integer	Integer	Integer, decimal	Integer, decimal	Integer	Integer, decimal	Integer, decimal	Integer, decimal	Integer, decimal	Integer, decimal, string
Interrupt	None	8 levels	1 level	1 level	8 levels	1 level	2 levels	128 vectored	256 levels	8 levels, 256 vec	3 levels, vectored
Year introduced	1971	1972	1973	1974	1974	1975	1975	1975	1978	1980	1979

Table 1.5 Characteristics of Various Microprocessors

			Memory size		*Memory performance*		
Memory module	*Function*	*Access method*	*Module size (bits)*	*Modules/ computer*	*Access time (s)*	*Data rate (bits/s)*	*Cost/bit ($)*[1]
Punched paper card	Permanent, archival	Random + linear	(500 ~ 1,000)/ card; ~ 1,000 card/unit	1 ~ 2	$10^0 \sim 10^3$	10^4	2×10^{-6} + 2×10^{-7}
Magnetic card	Secondary, archival	Linear + constant + cyclic	3×10^6	1 ~ 4	$10^{-1} \sim 10^0$	0.4×10^4	1.5×10^{-6} + 5×10^{-7}
Magnetic tape	Secondary, archival	Linear	7×10^8	1 ~ 16	$10^0 \sim 10^2$	$0.4 - 4 \times 10^6$	2×10^{-9} + 4×10^{-5}
Moving-head, floppy disk	Secondary	Cyclic	$10^6 \sim 10^7$	1 ~ 2	$10^{-1} \sim 10^{-1}$	10^5	$10^{-5} + 10^{-7}$
Moving-head disk pack	Secondary, files swapping	Linear + cyclic	4×10^8	1 ~ 16	$10^{-2} \sim 10^0$	$10^6 \sim 10^7$	3×10^{-8} + 3×10^{-5}
Fixed-head disk	Secondary, files swapping	Cyclic	$10^7 \sim 10^8$	1 ~ 40	$10^{-3} \sim 10^{-2}$	$10^6 \sim 10^7$	$10^{-7} \quad 10^{-6}$
Drum	Secondary, swapping	Cyclic	$(1 - 5) \times 10^7$	1 - 10	$10^{-1} \sim 10^{-2}$	$10^6 \sim 10^7$	10^{-3}
Magnetic bubbles	Secondary, swapping	Cyclic	$(1 - 5) \times 10^6$	1 ~ 10	$10^{-2} \sim 10^{-3}$	$10^5 \sim 10^6$	$10^{-2} \sim 10^{-4}$
Charge-coupled devices	Secondary, swapping	Cyclic	$10^6 \sim 10^7$	1 - 10	$10^{-3} \sim 10^{-4}$	$10^6 \quad 10^7$	$10^{-2} \sim 10^{-4}$
Video disk (write once)	Secondary	Cyclic	$10^{10} \sim 10^{12}$	1 - 10	$10^{-1} \sim 10^0$	$10^6 \quad 10^7$	5×10^{-8} + 5×10^{-7}
Bulk core memory	Primary and or secondary, swapping	Random	10^7	1 - 8	$(2 \quad 10) \sim 10^{-6}$	$10^6 \quad 10^8$	0.02 0.05
High speed core or thin-film memory	Primary	Random	$10^4 \quad 10^6$	1 16	$(0.2 \quad 2) \times 10^{-6}$	$10^7 \quad 10^8$	$10^{-2} \quad 10^{-3}$
Integrated circuit (MOS memory)	Primary	Random	$10^4 \quad 10^6$	1 20	$10^{-7} \sim 10^{-6}$	$10^6 \quad 10^7$	$10 \quad 10^{-3}$
Integrated circuit (bipolar memory)	Primary, processor state	Random	$10^4 \quad 10^6$	1 20	$10^{-8} \sim 10^{-7}$	$10^{-9} \quad 10^8$	$10^{-3} \sim 10^{-2}$
Integrated circuit (content addressable)	Primary, cache	Content, random	2×10^5	1 2	$\sim 10^{-7}$	10^8	$1 \sim 3$
Read only	Processor instruction-set definition	Random	$(1 \sim 5) \times 10^5$	1	10^{-7}	10^8	10^{-2}

[1] The first component is the memory medium (e.g., a disk pack), and the second component is the transducer (e.g., a disk drive).

Table 1.6 Some Memory Characteristics

1.4.3 The Mainframe Computers

There are many different types of computers which fall under this category, some of which are powerful dedicated computers. All of these large and very powerful general purpose computers are very expensive and they also tend to be the largest machines that can be built in a given technology at a given time. Their major characteristic is the possession of a large virtual-address space: in excess of 16 Mb. They also have a rich set of data-types. Over the years the scientific data-types have progressed from short-word to long-word fixed point scalars, to floating-point scalars and finally to vectors and arrays. They have high-performance CPU and some of them support instruction pipelining and instruction prefetch capability. Examples of these types of computers include the IBM System/360, VAX-11, CRAY-1, CDC 6600, STARAN and Illiac IV.

Since early 1960's, a number of parallel developments in computer architecture and software evolved, all seeking to make more efficient use of these expensive hardware installations. These developments sought not only to increase the number of tasks completed per unit of time but also to increase the efficiency of the hardware usage on single tasks. The four major system-level concepts that served as focal point for these developments were:

a) Multiprogramming

b) Timesharing

c) Virtual memory

d) Virtual machine

Another important consideration for these mainframe computers is the size of their memory and their memory hierarchy management. Usually the fastest, and most expensive, technology is used in the registers in

the CPU. Ideally one would like to execute programs as if all data existed in the CPU registers (mainly semiconductor). When more data are required, slower, larger, and lower-cost storage such as the primary memory (mainly semiconductor, magnetic core) is added. Larger program and data storage and medium-term storage can be provided by the secondary memory (mainly magnetic: drum, disk, tape). Finally tertiary memory (magnetic tape) provides archival and long-term storage. An important performance measure for memory is the memory access time, table 1.6. It has also been estimated that the CPU performance is very closely matched to the size of the computer memory (ref. 1.31, 1.32).

1.5 THIS THESIS

This thesis is primarily an analytical investigation into some aspects of measures of performance in a distributed computation environment and how this can be achieved in a wideband broadcast-type Local Area Computer Network (LAN). Hence the thesis can logically be considered to consist of two main parts: the pure communication aspects and the pure computation aspects.

By distributing a computation in this respect is meant the use of two or more autonomous computers which are interconnected by communication links to solve a single problem. Hence this thesis is based primarily on the processing performance of the traditional general-purpose single-CPU serial computer. As was explained previously, this traditional single-processor computer is only one of a potentially infinite number of possible computers: those with 1, 2, 3 ... n processors configured in all possible ways. The thesis specifically addresses itself to that mode of distributed computation in which two autonomous computers and interconnected by communication links are used to solve a single problem by

partitioning the problem into two portions and sharing it between themselves. Hence the CPU cache model of distributed computation is used. In a CPU cache environment (ref. 1.30, 1.33, 1.34) a smaller computer shares its computational workload with a larger computer existing in its neighbourhood with the express aim of speeding up the computation to reduce the total time that the smaller computer would have taken to solve the same problem on its own, as was explained previously. Furthermore, an arbitrary number of such smaller computers are assumed to exist in the same LAN as the larger computer as illustrated in Fig. 1.17. But, despite their co-existence in the same LAN the smaller computers do not partition and share any computation among themselves. The only permitted mode of partitioning and sharing of the computational workload is that between the smaller computers and the larger computer in the LAN. Moreover, it is up to the smaller computers to gauge and to decide the size of the portion of their computational workload to process themselves and how much portion to schedule to the larger computer. Furthermore it is left to the smaller computers to decide for themselves whether or not it is worth apportioning for scheduling and assigning any of their computational workload to the larger computer. In order to facilitate the smaller computers to reach a reasonable decision as to how much of their work-load to share with the larger computer the latter, at intervals, reports to the former an estimate of the computational workload backlogged at the larger computer. In this way the smaller computer will be encouraged to go ahead and schedule some of its computational workload to the larger computer if the backlog workload there is small and vice versa if the workload is large. It is possible that in the majority of cases the smaller computer has little or no workload at all to share with the larger computer in

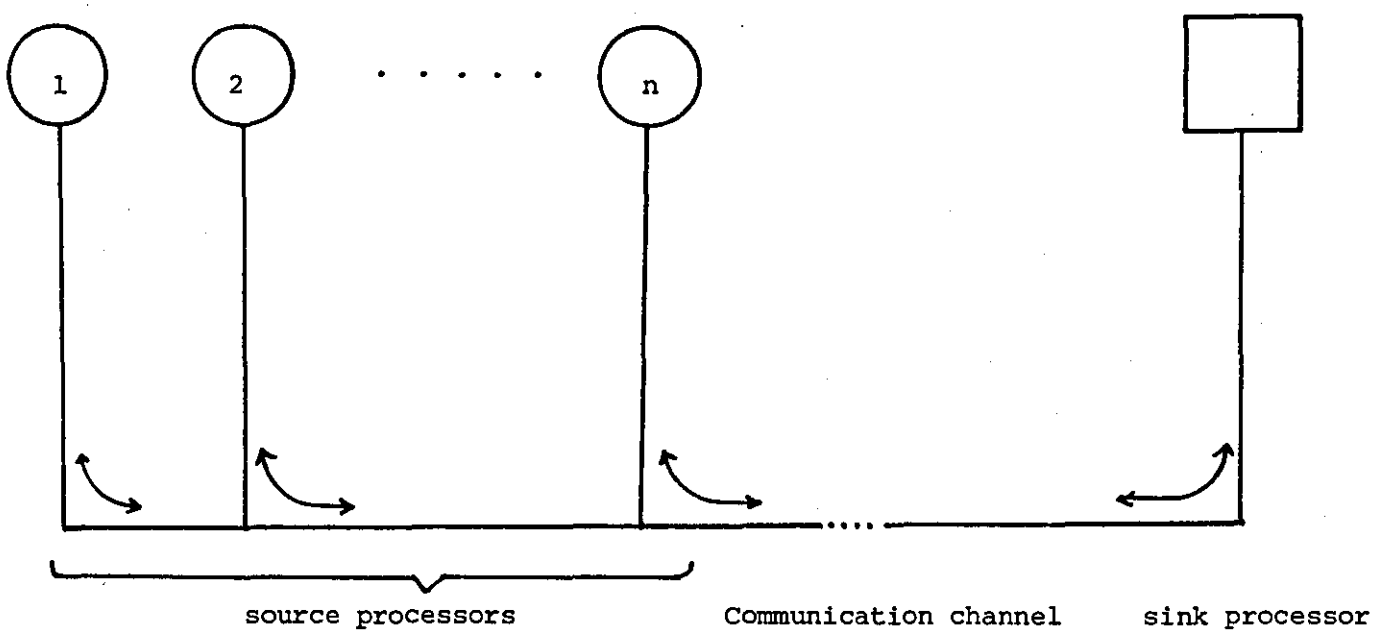


Fig. 1.17 Computer Interconnection and Layout

the LAN. In this case the workload at the larger computer will be low for most of the time, in which case any smaller computer apportioning its workload to the larger computer can expect to get it run there reasonably fast. On the other hand it may be expected that at certain other times the smaller computers have appreciable workloads and need to share some of it with the larger computer and thus increasing the backlog computational workload there too. In this case the smaller computers can expect that shared workload at the larger computer to run considerably slower. Hence the backlog computational workload at the larger computer is a critical factor in this mode of distributed computation.

It can be expected that as the smaller computers get faster, cheaper and more abundant programmers are going to continue to want to solve problems of increasingly larger size. The simultaneous existence of some form of a "computer centre" in the neighbourhood will then set the situation right for the CPU cache mode of distributed computation within the LAN. Such a situation can easily obtain in such environments as

a) The University

b) Factory complex

c) Research Laboratories

1.6 ORGANISATION OF THE THESIS

Chapter two will attempt to present the overall picture concerning the topology of the network of computers in the LAN. It will also examine how the various computers in the LAN communicate with one another in general and the various protocols involved. It will also define the main characteristics of the type of LAN on which this thesis is based.

Chapter three will examine the characteristics and organisation of a problem for distributed computation. It will also examine how the

problem can be partitioned as well as the partitioning algorithms used.

Chapter four will examine the characteristics and principles of computation. It will look at what constitutes a computation and the expression of a computation in terms of time. It will also examine computation probabilistically and will look at how the computation can be characterised by probability distribution or density functions.

Chapter five will address itself to the characteristics of the LAN with a view to examining channel delay performance and will determine the mean channel delay experienced by the channel packets.

Chapter six will examine and present the computational model used to characterise distributed computation. Its main aim is to characterise the workload at the large and more powerful computer in the LAN as this is a critical factor in distributed computation. It will also present the theoretical and experimental (simulation) results obtained.

Finally, chapter seven will briefly review the results and make suggestions for possible future developments.

CHAPTER 2

NETWORK ORGANISATION

2.1 INTRODUCTION

The design and organisational issues of computer networks are very broad and cover many diverse and interrelated areas such as the computer hardware organisation, computer software organisation, communications processor hardware and software, network topology and network protocols, (ref. 2.1, 2.2). Other major operational issues such as the routing procedures, flow control, congestion control and communication switching procedures also have to be resolved in order to facilitate a smoothly operating network. This chapter examines some of these main organisational features of a computer network with the aim of presenting the basic format of this thesis.

2.2 THE NETWORK STRUCTURE

Modern computer networks are designed in a highly structured way. A structured design facilitates the network to be flexible and to grow in size or in quality of the service it provides simply by adding on another facility or a software sophistication without the need to disrupt the existing layout. In a computer network this layout can be either the network hardware or the network software which together determine the kinds of services or applications the network may support.

The two main issues of the network structure are its topology and architecture.

2.2.1 The Network Topology

In a computer communication network there exists a collection of machines (the computers) (ref. 2.8), which are capable of running user

programs. These machines are sometimes referred to as the network-users or the network-stations or just users. These network-users are connected together by means of the communication subnet whose job it is to carry data from one network-user to another. The communications subjet itself can be considered to consist of two basic components:

a) the communication switches

b) the communication channels

The communication switches may themselves be small processors of varying degrees of complexity. All traffic to and from a network-user must go through its communication switch. All the communication switches are in turn interconnected by means of the transmission channels. The way in which the communication channels interconnect the communication switches define the topology of the network.

Communication within the communication subnet can take on two basic forms: point-to-point or broadcast (ref. 2.11, 2.12, 2.13, 2.14).

2.2.1.1 Point-to-point channels

In a point-to-point communications subnet (ref. 2.7), the network contains many communication links each one of which interconnects a pair of network-users. If two network-users not sharing a link wish to communicate, they must do so indirectly via other intermediate network-users. When a message is sent from one network-user to another via one or more intermediate users, the message is received at each intermediate user in its entirety, stored there temporarily until the required output line is free, and then forwarded. Hence point-to-point channels are also variously known as store-and-forward channels.

Some of the network topologies that support point-to-point communications subnet are (ref. 2.14):

a) star network

b) loop network

c) hierarchical tree network

d) mesh network

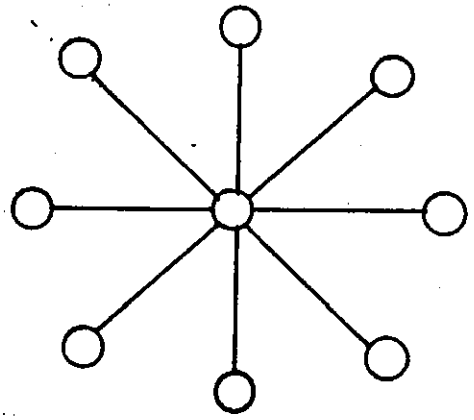
These are illustrated in fig. 2.1.

Point-to-point communications subnet is generally not used for local networks but has been used for long-haul computer communication networks such as ARPA (ref. 2.3, 2.10), SITA (ref. 2.4, 2.5) and TYMNET (ref. 2.6).

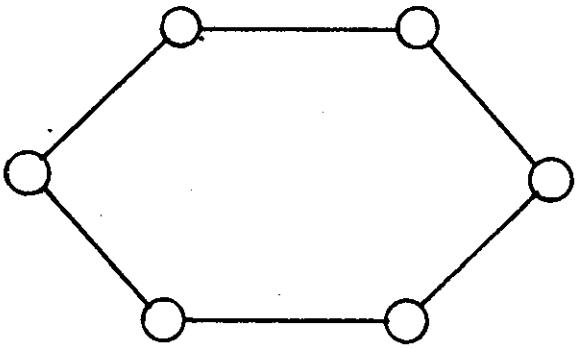
2.2.1.2 Broadcast channels

In a broadcast-type communications subnet (ref. 2.10), a single communication channel is shared by all the network-users. Inherent in such a topology is the fact that messages transmitted by any network-user will be received by all the other users in the network. Messages must therefore carry some identification indicating to which network-user the message is intended for. Network-users receiving messages not intended for them must ignore and not interfere with them. Because all users share a common transmission link, one and only one network-user is allowed to gain access and to transmit into the channel at any one time. Hence some form of a channel access control must be exercised to determine which network-user may transmit next. Centralized or distributed channel access control schemes can be used. In the event of a simultaneous channel access some collision arbitration mechanism must be employed to resolve the conflict.

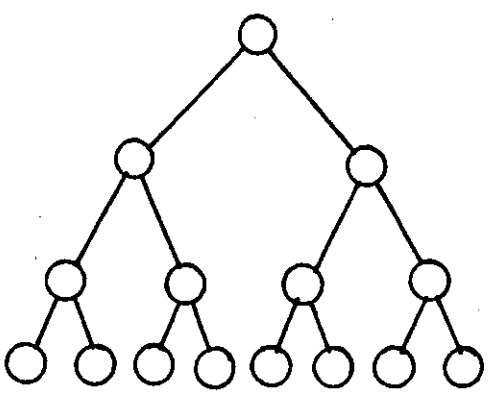
Some of the network topologies that support broadcast-type communications subnets are shown in fig. 2.2, and described below.



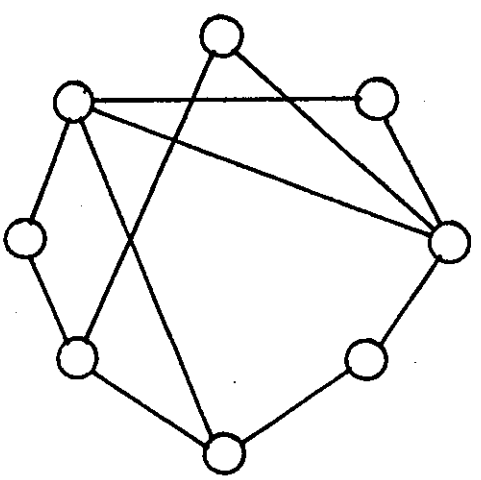
(a) star topology



(b) loop topology

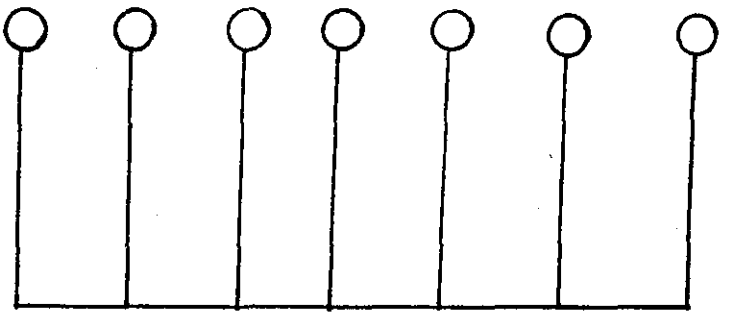


(c) hierarchical tree topology

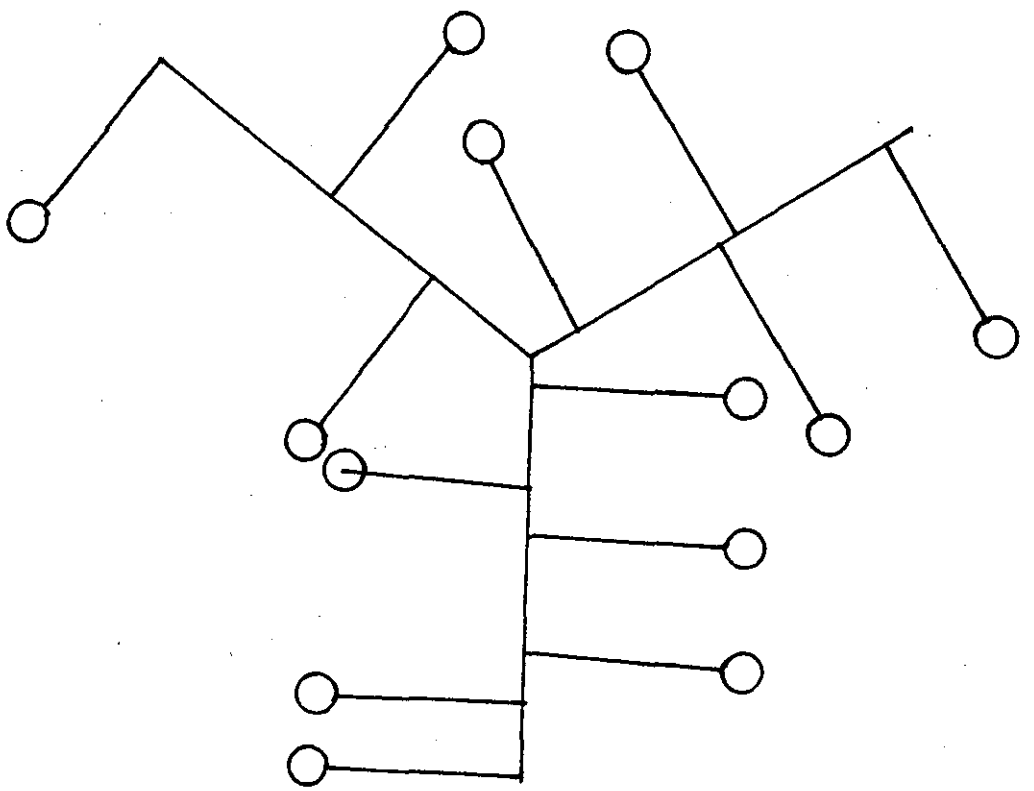


(d) mesh topology

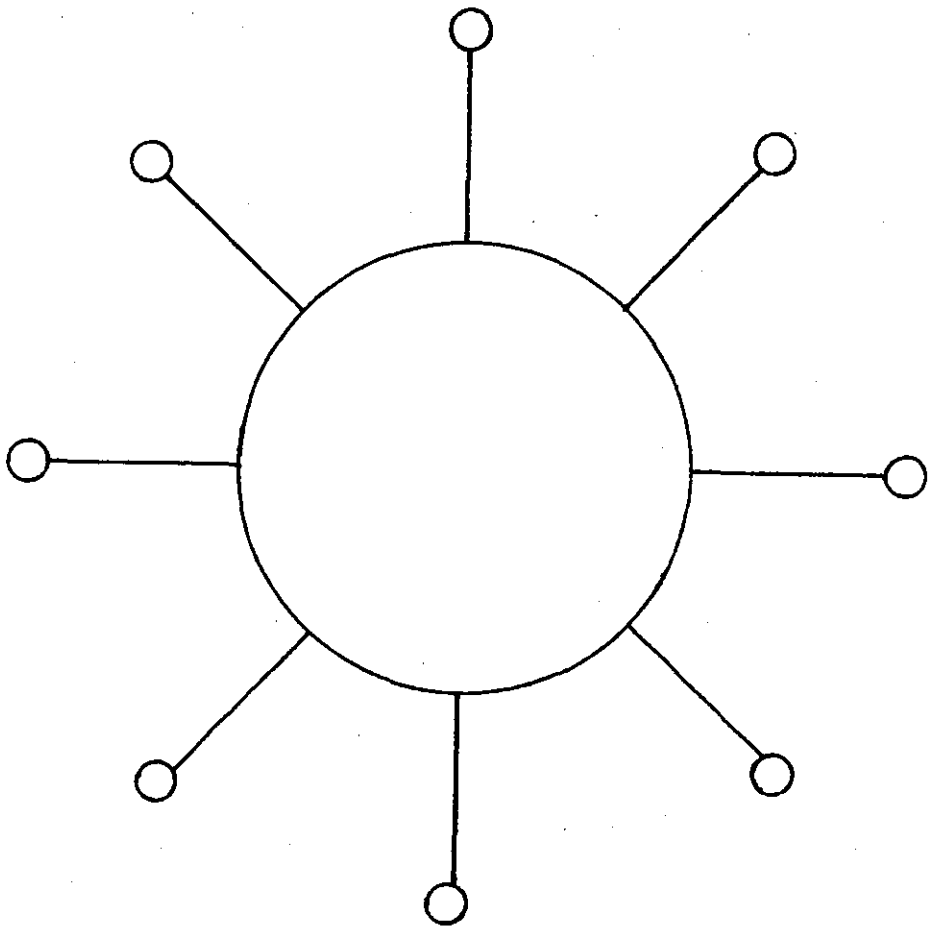
Fig. 2.1 Point-to-point network topologies



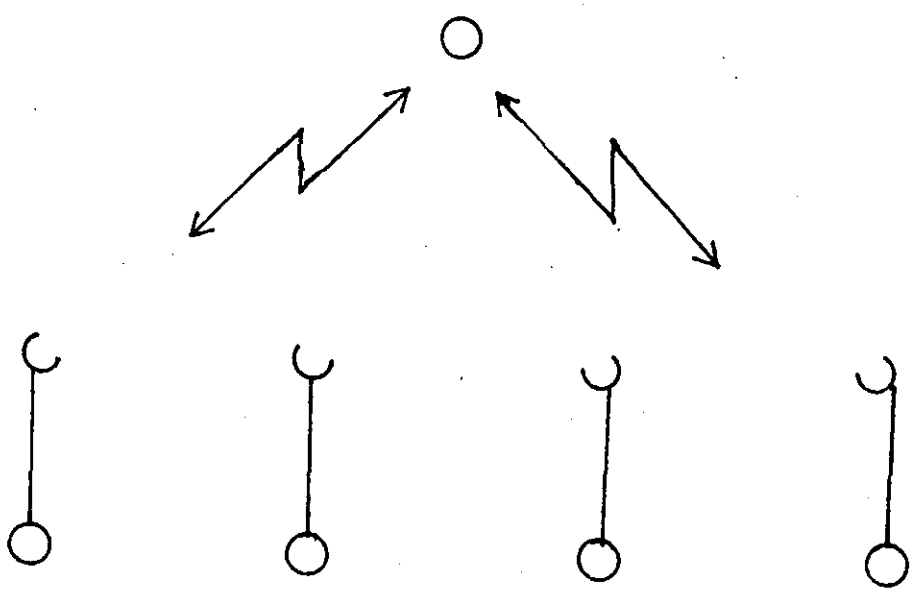
(a) The bus topology



(b) The broadcast tree topology



(c) The ring topology



(d) Satellite or radio topology

Fig. 2.2 Multi-point Network Topology

2.2.1.2.1 The Bus Topology

In the bus network (ref. 2.16) all network-users attach, through the appropriate hardware interfacing, directly to a linear (passive) transmission channel. At any one time just one network-user is allowed to be the bus-master and can transmit its messages (ref. 2.15). During this time all the other users are prohibited from sending their messages but must listen to the transmissions in progress. The twisted-pair and the coaxial cable are two of the most common transmission channels employed in the bus communications subnet. Hence the bus communications subnet can support wideband communication. The coaxial cable can support baseband or broadband communication (ref. 2.17, 2.18, 2.21). In the baseband bus communications subnet digital signalling and bidirectional transmission are used. In this way the entire channel bandwidth is utilized by the signal. The bus communications subnet can support a signal transmission speed of up to about 50 Mbps, a maximum distance of about 25 km, and can also support several hundred network-users depending on the size of the network and the transmission speed. The broadband bus communications subnet (ref. 2.19, 2.20) uses analog signalling and unidirectional transmission so that frequency division multiplexing (FDM) can be used and considerably longer distances can be covered. If the size of the network is small, the broadcast bus topology is appropriate.

2.2.1.2.2 The Tree Topology

The tree topology is a slight generalization of the bus topology. The transmission channel is a branching passive cable without any closed loops. Just like the bus topology, baseband and broadband communication can be supported by the tree communications subnet topology (ref. 2.19, 2.20, 2.21). Most of the characteristics cited above for the bus topology

also apply to the tree topology.

2.2.1.2.3 The Ring Topology

In the ring topology (ref. 2.22, 2.23, 2.24, 2.25), network-users are attached to the transmission channel via repeaters, each of which is connected to two others by unidirectional transmission links to form a single closed path. Since multiple users share the ring, control is needed to determine at what time each user may insert its message. Like in the bus and the tree topologies this control can be achieved in a distributed rather than in a centralized way. Each user has enough channel access logic that controls transmission and reception of messages in the subnet.

Transmission in the ring subnet is unidirectional so that the message bits are transmitted sequentially, bit by bit, around the ring from one repeater to the next. Each repeater in turn regenerates and retransmits each bit. In this way each individual bit of the message propagates around the ring separately, not waiting for the rest of the message to which it belongs. Thus each bit may travel round the entire ring within a few bit times, usually before the complete message has been transmitted. This characteristic behaviour of the broadcast ring topology differs from that of the loop point-to-point topology in that in the loop topology each user message is not retransmitted until the entire message has been received. In the ring network each link between the network users may therefore have a different message on it at any one time.

Twisted-pair, baseband coaxial cable and the optical fibre can be used as the links of the transmission channel in the ring topology, and either analog or digital signalling can be used. A transmission speed of

up to 10 Mbps, a maximum distance of a few kilometers and a few tens of users can be supported on the ring communications subnet.

2.2.1.2.4 The Satellite and Radio Topology

In the satellite (ref. 2.45) or ground radio (ref. 2.26, 2.27), topology each network user has an aerial by means of which it can transmit and receive. The satellite network consists of a set of ground stations and a communications satellite in a synchronous orbit. The ground stations transmit data to the satellite, which then broadcasts the transmission back down to all the ground stations. In this way all the network users can hear the output of the satellite and also possibly some of the transmissions of the other network users to the satellite.

Because of the nature of the satellite network communications subnet, the transmission frequency ranges are high and hence there is a potential for high transmission data rates (ref. 2.28). However, the satellite subnet has a long round-trip propagation delay of about half a second (ref. 2.13) and hence is more suitable for long-haul networks. The long-haul ARPA network (ref. 2.27) has satellite links in its communications subnet.

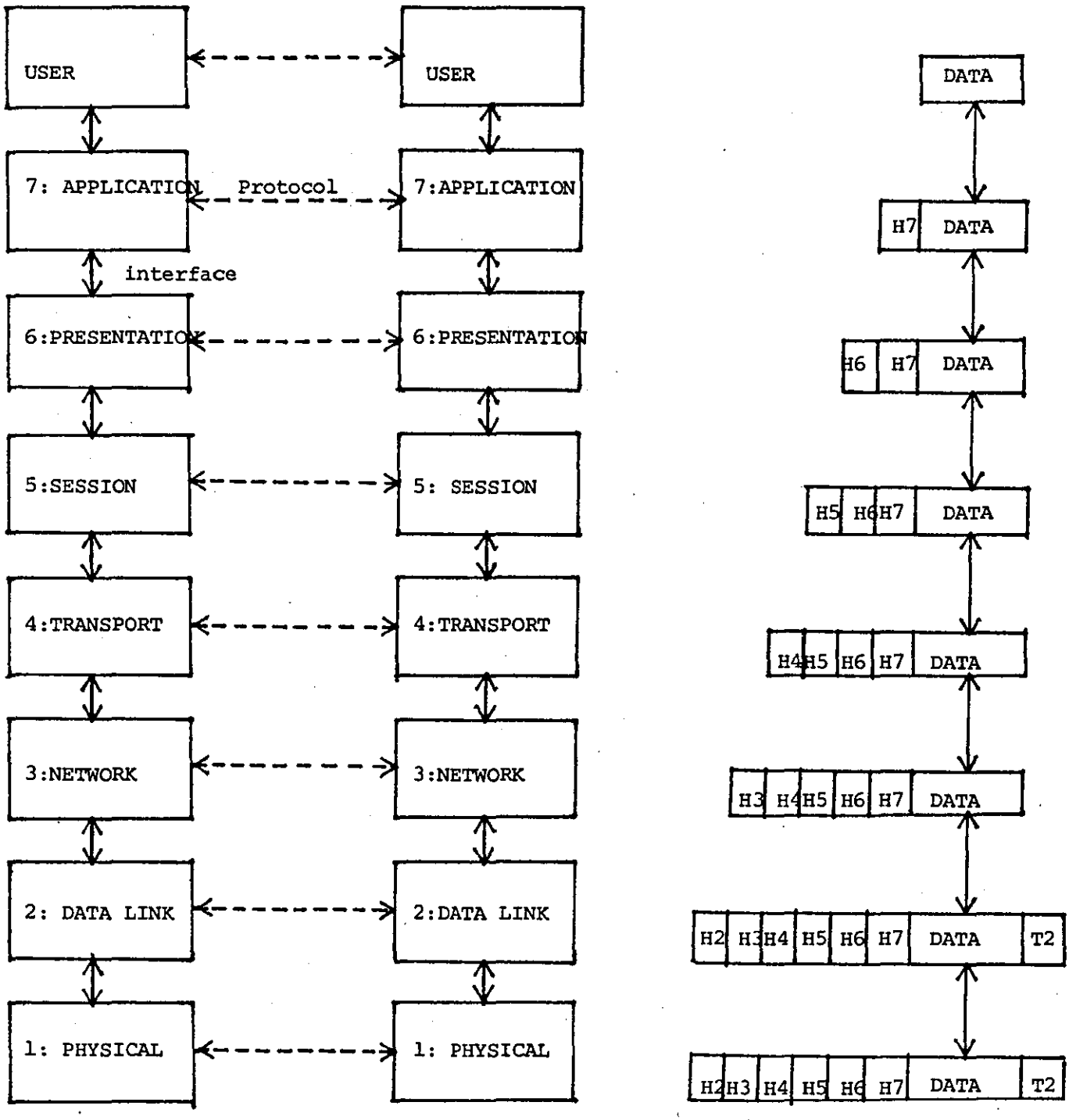
2.2.2 The OSI Network Architecture

In many computer networks different types of computers exist in the network. Yet orderly communication between the various heterogenous computers in the network (ref. 2.29, 2.30, 2.31, 2.32), is a major goal. In general these computer communications aspects can be considered from the point of view of the hardware and software. Both the computer and the communications hardware are reasonably standard and present fewer problems. But the architectures of the communications process for each computer in the network needs to be fairly standard too in order to facilitate proper

communication among the various heterogenous computers. The model of such an architecture which forms the framework for defining standards for linking the many heterogenous computers in a network is the Open Systems Interconnection (OSI) (ref. 2.33, 2.34).

The OSI model partitions the software for the communications function into a structured set of seven layers as shown in fig. 2.3. Such a structured and layered organisation also reduces the design complexity. The purpose of each layer is to perform a related subset of the functions required to communicate with processes in the other layers. By so doing each layer offers certain services to the higher layers, shielding those layers from the details of how the offered services are actually implemented. In this way, each layer performs a specific collection of properly defined functions and is also so defined that changes in one layer do not provoke serious changes in the adjacent layers.

In the OSI model layer n on one user machine must carry out its communication with layer n on another user machine in the network strictly according to the layer n protocols. Only at the lowest layer (layer 1) does direct physical communication between the corresponding users take place. All the other higher layers must establish and carry out virtual communication. By so doing each higher layer passes data and control information to the layer immediately below or above it until the lowest or the highest layer is reached depending on the direction of the communication process. The entities comprising the corresponding layers on the two different user machines are called peer processes. The protocols for the peer processes define such things as the data formats and signal levels, control information for proper coordination and error handling, and also



H_i = Header encapsulation for the i^{th} layer

T_i = Trailer encapsulation for the i^{th} layer

Fig. 2.3 The OSI Model Protocol and Data Encapsulation and Decapsulation

speed matching and sequencing.

The seven OSI model layers and their functions are as follows:-

2.2.2.1 Layer 1

Layer 1 is the Physical layer. (ref. 2.35, 2.36, 2.37). This layer is mainly concerned with the transmission of the raw unstructured bit stream over the physical transmission link. It is also concerned with the setting of such parameters as the signal voltage swing and the bit duration as well as dealing with the general issues pertaining to the

a) mechanical: connector pin configuration and pin arrangement

b) electrical: voltage swing, voltage change timing, transmission data rates, maximum transmission distance

c) functional: connector pin signal functions and interpretation

d) procedural: the sequence of events to be performed following the reception of a signal

characteristics necessary to establish, maintain and deactivate the physical link. RS-232-C is the most common standard in use. It can be used to connect a digital device to a modem which in turn connects to a voice-grade telephone line (ref. 2.38).

2.2.2.2 Layer 2

Layer 2 is the Data Link layer (refs. 2.39, 2.40, 2.41). Its main purpose is to take the raw transmission facility and to transform it into a line that is free of transmission errors to the higher Network Layer above it. The input data stream is broken up into blocks of data (data frames) and then transmitting these data frames sequentially to the higher Network Layer. It also processes such functions as frame acknowledgements. The Data Link layer shares many of its characteristics with the existing

bit-oriented protocols such as the HDLC (ref. 2.109, 2.111).

2.2.2.3 Layer 3

Layer 3 is the Network layer (refs. 2.42, 2.43, 2.44, 2.45, 2.46, 2.47, 2.48). Its main function is to control the operation of the communications subnet. It handles data in the form of packets. The packets may traverse the communications subnet either independently (datagram) or through a preestablished logical route (virtual circuit) and hence it is the work of the Network layer to govern the routing of the packets and also to deal with the issues of the flow and congestion control within the subnet. In this way it is the responsibility of this layer to provide for the transparent transfer of data between the transport entities. By so doing it relieves the higher layers above it, which provide end-to-end protocol, of the need to know anything about the underlying communications subnet. For terminals operating in the packet switching mode on public data networks the CCITT X-25 provides an interface between the data terminal equipment and the data circuit-terminating equipment. (ref. 2.49, 2.75, 2.76).

2.2.2.4 Layer 4

Layer 4 is the Transport layer (refs. 2.7, 2.50, 2.151). Its main function is to accept data from the Session layer above it, split the data up into smaller units and pass these to the Network layer below it. It is also the responsibility of this layer to ensure that these smaller data units arrive reliably at the other end. In this way the Transport layer has to provide also the end-to-end error recovery and flow control. It is also the function of this layer to specify such details as the type of service (datagram or virtual circuit) and the grade of service (error and loss levels, minimum delay, priority, and security). All this is

done in the most efficient way possible and in a way that isolates the Session layer above it from the inevitable changes in the hardware technology. Layers 4, 5, 6 and 7 of the OSI model provide end-to-end protocols and are not concerned with the details of the underlying communications subnet (ref. 2.52).

2.2.2.5 Layer 5

Layer 5 is the Session layer (refs. 2.7, 2.51, 2.52). The Session layer is the user's interface to the network. It provides a means for establishing, managing and terminating a connection (Session) with another process on another machine. In a distributed computation environment, it is also necessary that the Session layer manage the run-time environment. These are basically the routines that handle the interprocess communications and also monitoring the network for vital statistics relating to the interprocess calls at run-time during each computation session. Hence, the Session layer must make decisions, at run-time, concerning the location and movement of the various subproblems for computation as the workload in one machine changes, as well as the overall practicability of the network to support distributed computation. In this way the overall supervision and management of the distributed computation process is done by the Layer 5 protocols.

2.2.2.6 Layer 6

Layer 6 is the Presentation layer (ref. 2.53, 2.54, 2.55, 2.56, 2-57). The Presentation layer performs certain transformations on the data. It performs functions that are requested often enough to warrant a general solution and standardized application for them. Such functions can often be performed by special library routines that may be called by the

user. Examples of such transformations on data that may be performed by this layer are encryption, text compression and reformating (ref. 2.57, 2.58).

2.2.2.7 Layer 7

Layer 7 is the Application layer (ref. 2.59, 2.60, 2.61). The general content of the services provided by the Application layer is largely left to the individual user. When two user processes on different machines communicate, they alone determine the set of allowed messages and the action taken upon receipt of each. The boundary between the Presentation layer and the Application layer separates the domain of network designers from the domain of network users.

The Application layer defines the applications that can be run in a distributed environment. Such applications include the electronic mail, a transaction server, a file transfer protocol, and a job manipulation protocol such as the distribution of a computation. In a distributed computation environment a job scheduler has to be employed to partition the workload of one machine for distributing and sharing the workload with the various machines in the network so as to take maximum advantage of the resources of the network.

2.3 NETWORK CLASSIFICATIONS

Distributed processing can in general be characterised as a spectrum of activities which vary in the degree of their decentralization. At one extreme is remote networking in which one finds loose interconnection of previously isolated, widely separated, and rather large computing machines. These are the long-haul networks. A good example of a long-haul network is the ARPA network (ref. 1.12). At the other extreme is multiprocessing

in which one finds the construction of previously monolithic and serial computing systems from increasingly numerous and smaller systems computing in parallel such as the SIMD and MIMD structures (ref. 1.4, 1.5). Near the middle of this spectrum is local networking (ref. 2.62, 2.63, 2.64, 2.65, 2.66, 2.67, 2.68) in which one finds the interconnection of computers to gain the resource sharing of computer networking and the parallelism and speed of multiprocessing. This characterisation can be viewed in terms of the distance of separation in metres (m) and the physical size of the network as follows:

Separation Distance	Physical Size	Classification
less than 0.1 m	circuit board	Data Flow Machine
0.1 m to 1.0 m	system	Multiprocessors
1.0 m to 10 m	Room	Local Networks
10 m to 100 m	Building	
100 m to 1 km	Campus	
1 km to 10 km	City	
10 km to 100 km	Country	Long-haul network
100 km to 1000 km	Continent	
greater than 1000 km	Planet	

Another characterisation that has been used to classify networks in a distributed processing environment is the product of the distance of separation and the data transmission rate. This product, now estimated at about 1 Gigabit-metre per second, is also sometimes taken as an indication of the level of the communication technology. This classification is as follows:

Classification	Separation Distance	Bit-rate
Long-haul networks	greater than 10 km	less than 0.1 Mbps
Local networks	10 km - 0.1 km	0.1 - 10 Mbps
Multiprocessing	less than 0.1 km	greater than 10 Mbps

2.3.1 Local Networks

In general terms a local network is a communications network that provides interconnection of a variety of data communicating devices within a geographically small area such as a University campus or a factory complex (ref. 2.63, 2.67, 2.69, 2.70). Some of the major characteristics of a local network are:

a) a diameter of not more than a few kilometers

b) ownership by a single organisation

c) a data transmission rate exceeding 1 Mbps

d) low transmission error rates (10^{-8} to 10^{-11}).

Such a characterisation of local networks is quite general. By using such a definition of a local network three different and distinct types of local networks can be identified (ref. 2.71, 2.72): the CSLN network (ref. 2.73), the HSLN network (ref. 2.74) and the LAN network (ref. 2.47, 2.64), as described below.

2.3.1.1 The CSLN network

The CSLN is a circuit-switched local network (ref. 2.38, 2.86) that accommodates the characteristics of local network explained earlier. An example of a CSLN network is the Computerised Branch Exchange (CBX) (ref. 2.73, 2.77, 2.79). In the CSLN, the network-users are connected in a star topology to the main centrally located switching unit which establishes a dedicated path between any two users on the network. In this way hundreds or thousands of network-users can be thus interconnected within a relatively small geographical area (about 1 km), but the data transmission speed is usually low (9.6 to 64 kbps) (ref. 2.b7).

2.3.1.2 The HSLN network

The HSLN is a high-speed packet-switched local network (ref. 2.74) with all the characteristics of the local networks defined earlier. The

HSLN have been described as characteristically computer-room networks (ref. 2.63) which connect a few relatively expensive high speed mainframe computers and other mass storage or high speed data processing peripherals of large organisations such as large companies or research laboratories. The network-users in the HSLN local networks aim to obtain high end-to-end throughput at high data transmission speeds (about 50 Mbps) using wideband coaxial cable bus topology within a relatively small geographical area (less than 1 km).

2.3.1.3 The LAN network

The LAN are local area packet-switched networks that too share all the characteristics of local networks defined earlier. LANs can support mainframe computers, minicomputers, microcomputers and other terminal and peripheral devices (ref. 2.47, 2.67, 2.69). Bus or tree topologies using coaxial cables or ring network topology using twisted-pair, coaxial cable or optical fibre links (ref. 2.82, 2.83) can be used in LANs. Data transmission rate on LANs can average between 1 and 20 Mbps. Several hundred network-users can be supported on a LAN. A considerable amount of research has been directed to the study of LANs (ref. 2.68, 2.69, 2.47) and a draft IEEE 802 LAN standard (ref. 2.80) has been developed. Both baseband and broadband cable systems can be implemented on LANs (ref. 2.19, 2.21) and utilizing the entire bandwidth of the transmission channel.

One of the most well-known LAN network was based on baseband coaxial cable bus topology and was called the Ethernet System (ref. 2.68, 2.69, 2.81). An Ethernet is a branching broadcast communication network for carrying digital data packets among its locally distributed network-users. The packet transport mechanism provided by Ethernet has been used to build networks which can be viewed as either local computer networks or loosely

coupled multiprocessors. The Ethernet's shared communication facility, its ether, is a passive broadcast transmission medium with no central control. Coordination of access to the ether for packet broadcast is distributed among the contending network-users using a controlled statistical arbitration. Switching of packets to their destinations on the ether is distributed also among the receiving network-users using packet address recognition. Because of its flexibility, the ether can simply be added on and extended to accommodate more network-users as shown in fig. 2.4. The baseband coaxial cable system on which the ethernet is based was a special 50 - ohm cable rather than the standard 75 - ohm cable because the digital signals on the 50 - ohm cable suffer less intense reflections from the insertion capacitance of the taps, and also provides better immunity against low-frequency electromagnetic noise. Figs. 2.5, 2.6, 2.7 and 2.8 show the organisation of an ethernet communication network (ref. 2.68, 2.81).

The main components of an ethernet computer communication network include:

a) 50-ohm terminators

b) 50-ohm coaxial cable

c) Tap

d) Transceiver

e) Transceiver cable

f) Controller interface

g) Controller

h) Computer

The computers attach to the transmission cable by means of a tap. The distance between the taps is designed to ensure that reflections between adjacent taps do not add in phase. The transceiver taps into the

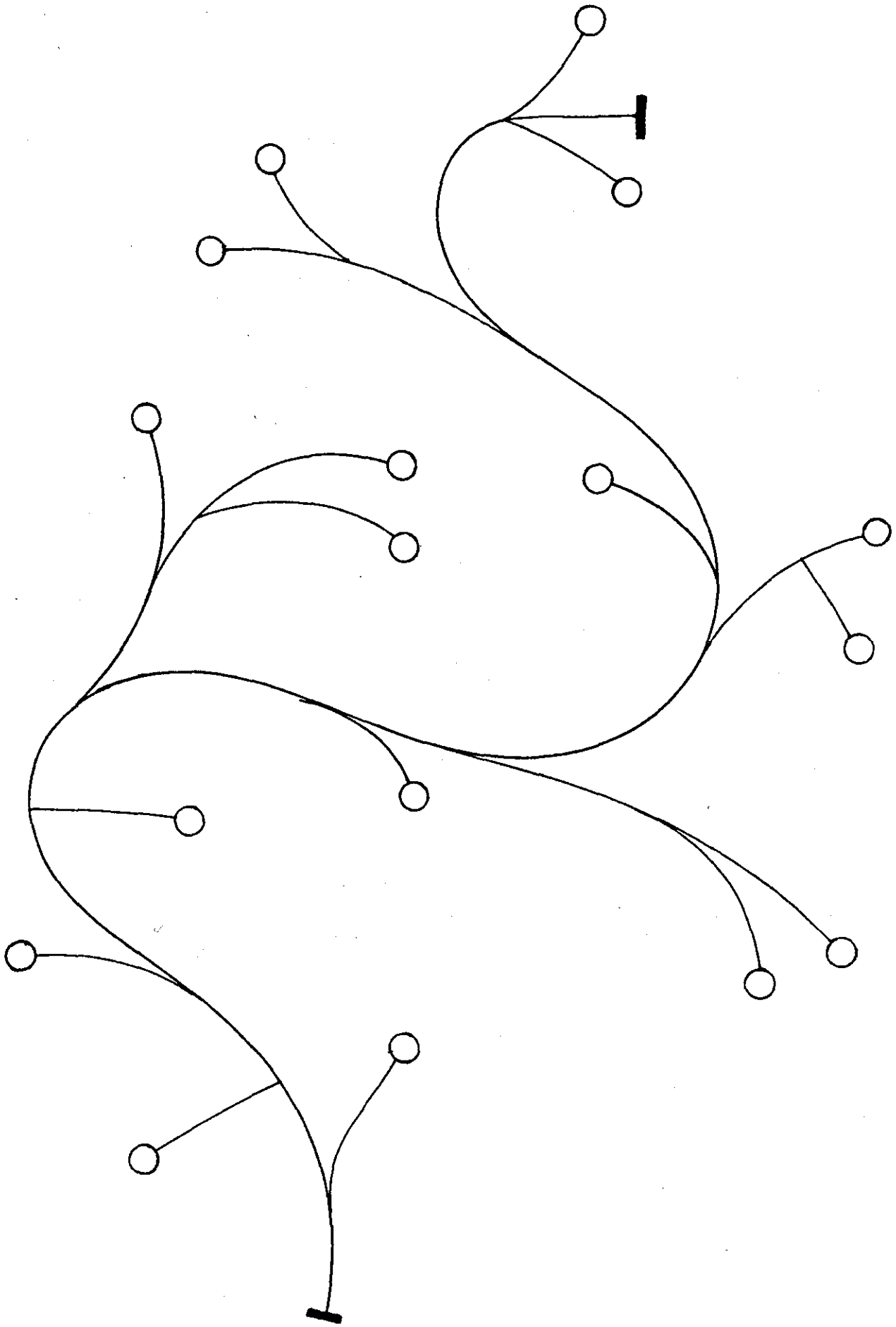


Fig. 2.4 The Ether

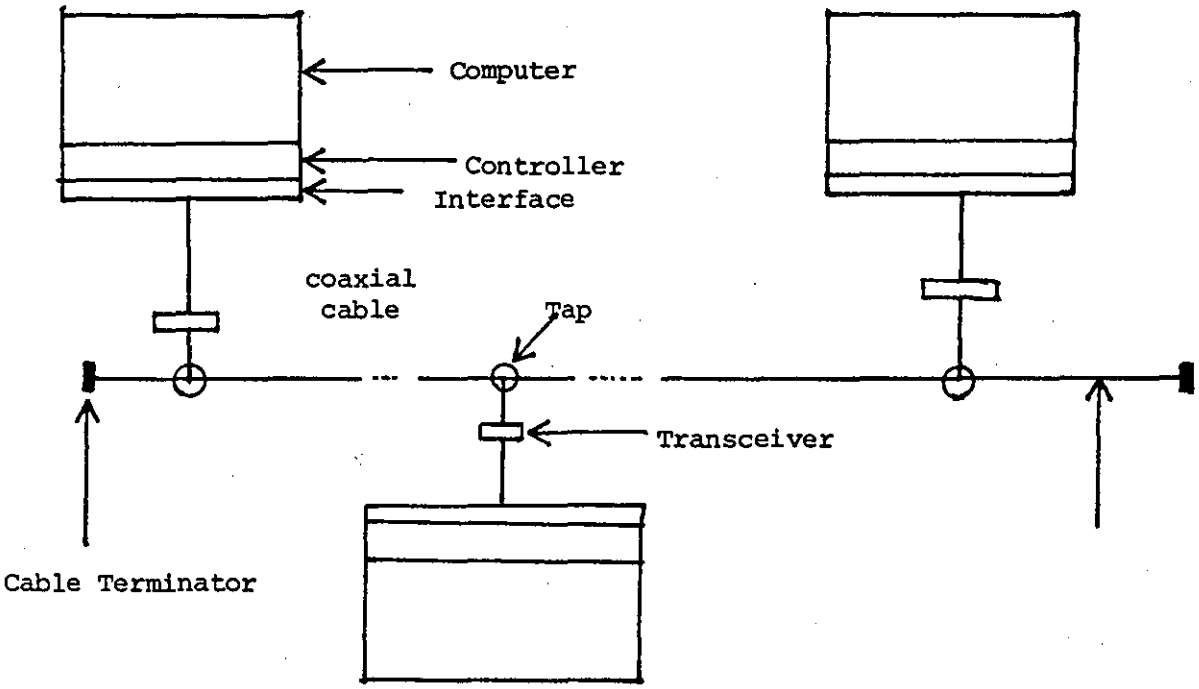


Fig. 2.5 Ethernet network

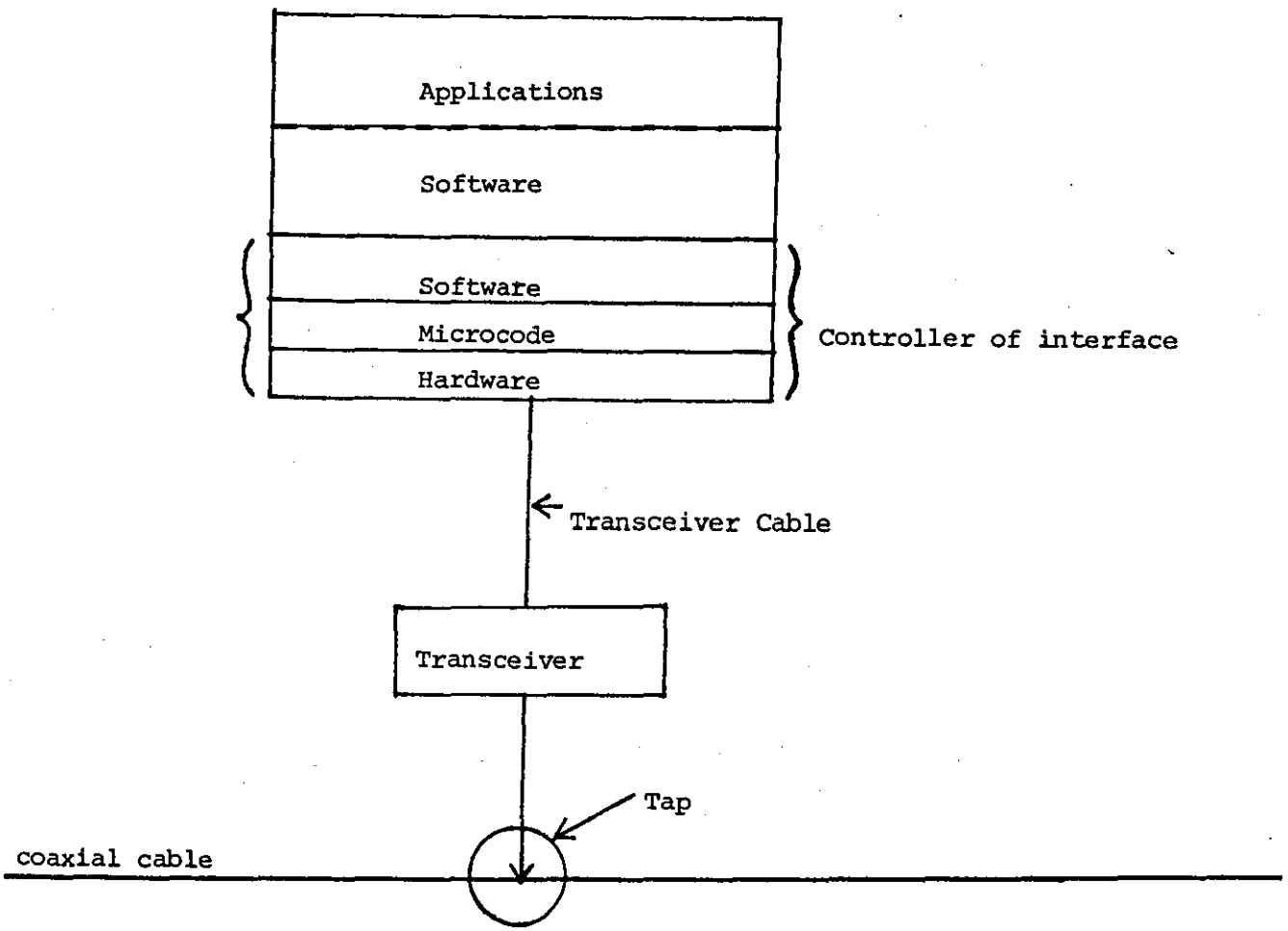


Fig. 2.6 Ethernet Connection

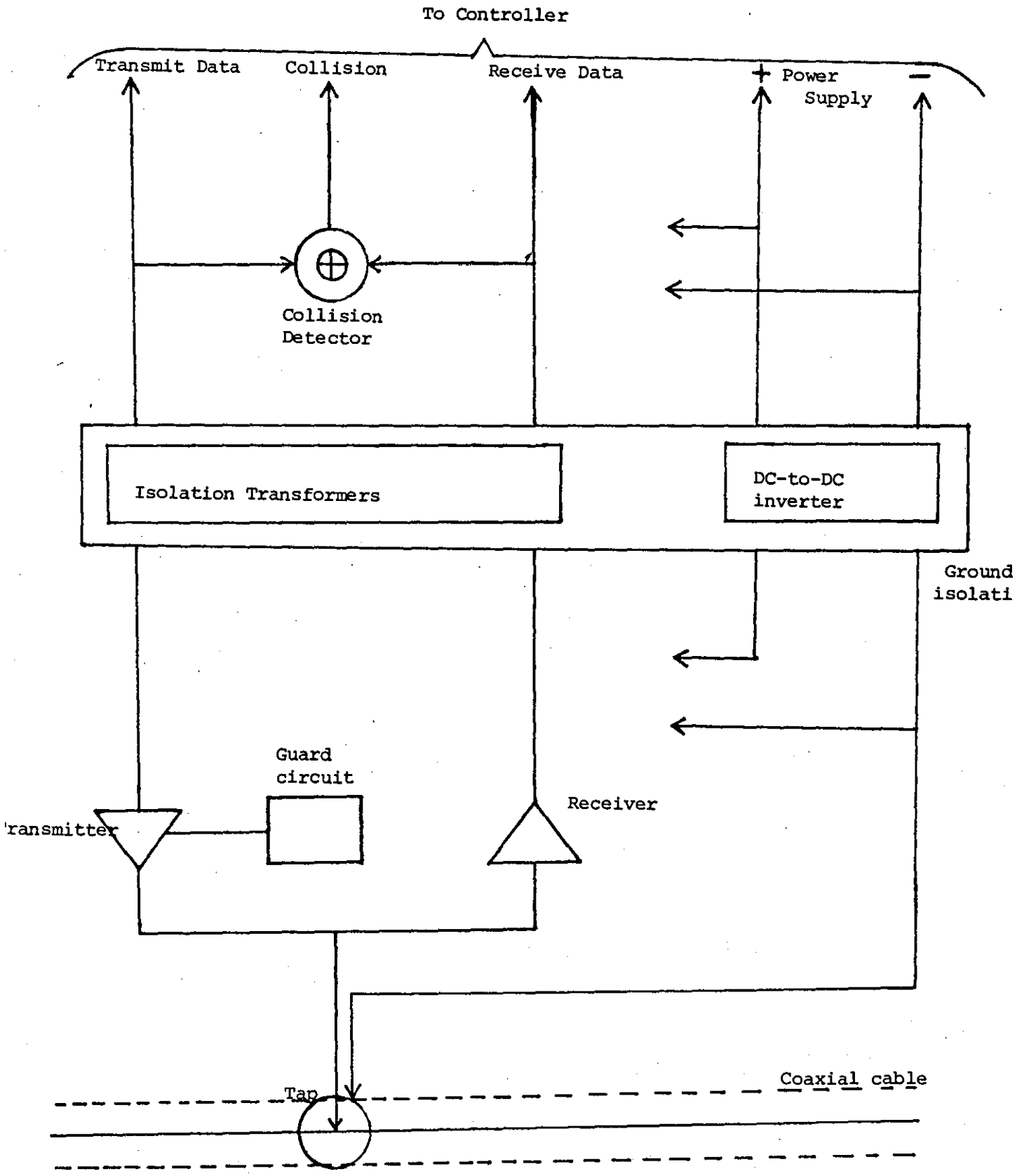


Fig. 2.7 Ethernet Transceiver

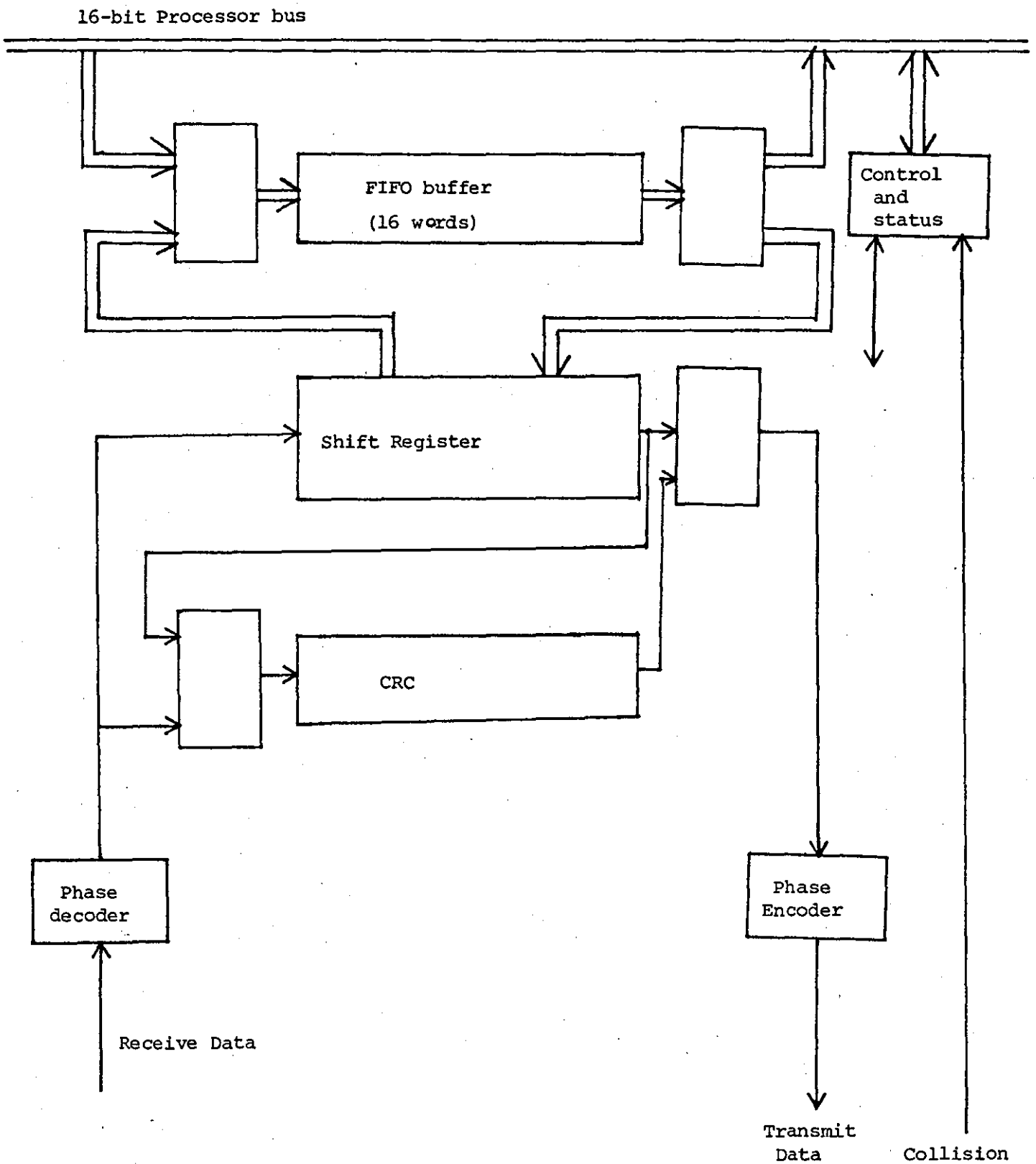


Fig. 2.8 Ethernet Controller Hardware

coaxial cable and facilitates the transmission and reception of digital signals to and from the transmission channel. All the electronics necessary to recognise the presence of valid digital signals on the transmission channel and the detection of invalid collision signals on the channel are contained in the transceiver. The transmission data may be Manchester encoded (ref. 2.80). This coding scheme has the property that it has a transition in every bit cell and has a 50% duty cycle. The bits are phase-encoded in the controller before being passed to the transceiver. The first half of a bit cell contains the complement of the bit while the second half contains the bit itself. In this way, there is a transition in the middle of the bit cell: a positive edge corresponding to a "1" and a negative edge corresponds to a "0". The voltage levels transmitted into the cable are 3 volts for "on" and 0 volts for "off". Carrier is detected by the presence of these transitions on the cable. The detection of valid and invalid digital signals on the transmission channel is a basic requirement in broadcast networks that employ carrier-sense multiple-access with collision detection (CSMA-CD) protocol. The transceiver connects to the controller via a four-pair transceiver cable and interface. The interface may be used to implement serial-parallel or parallel-serial data format. The controller contains the main hardware and software necessary to facilitate communication in the LAN communications subnet. Some of the controller functions include the detection and processing of the collision signal, the implementation of the packet retransmission strategy, enabling and disabling the transceiver, and managing the exchange of packets to and from the communication channel. The 50 - ohm cable terminators 'mop up' signals and prevent sustained signal reflections from the ends of the transmission cable.

The number of cable taps, the coaxial cable length, the transmission bit rate together with the electrical characteristics of the ethernet components have all to be taken into account to determine the physical size of the ethernet LAN network. A prototype bus topology ethernet communication network operating at 3 Mbps, connecting up to 255 computers, which span over a linear distance of 1 km was one of the earlier such LAN networks (ref. 2.68, 2.81).

A repeater can be used to extend the size of an ethernet network (ref. 2.26, 2.68, 2.84, 2.91). Such a repeater consists of two transceivers joined together and connected to two different segments of the coaxial cable as shown in fig. 2.9. It allows digital signals to pass in both directions between the two cable segments, amplifying and regenerating the signals as they traverse through the network but without buffering the signals. Despite the repeaters, the service provided by the ethernet LAN is transparent to the users in the network.

2.4 NETWORK ACCESS PROTOCOLS

The purpose of the communications subnet is to transport data between the network-users quickly and reliably. For this to be accomplished one of the major decisions to be made is how the various contending users in the network should gain access to the communications subnet. These decisions define the set of rules that must be implemented in order to queue and multiplex the ready users in the network who have data to transmit. The method used to accomplish such multiplexing depends very much on the nature of the users, the topology of the network and the switching mechanism to be used.

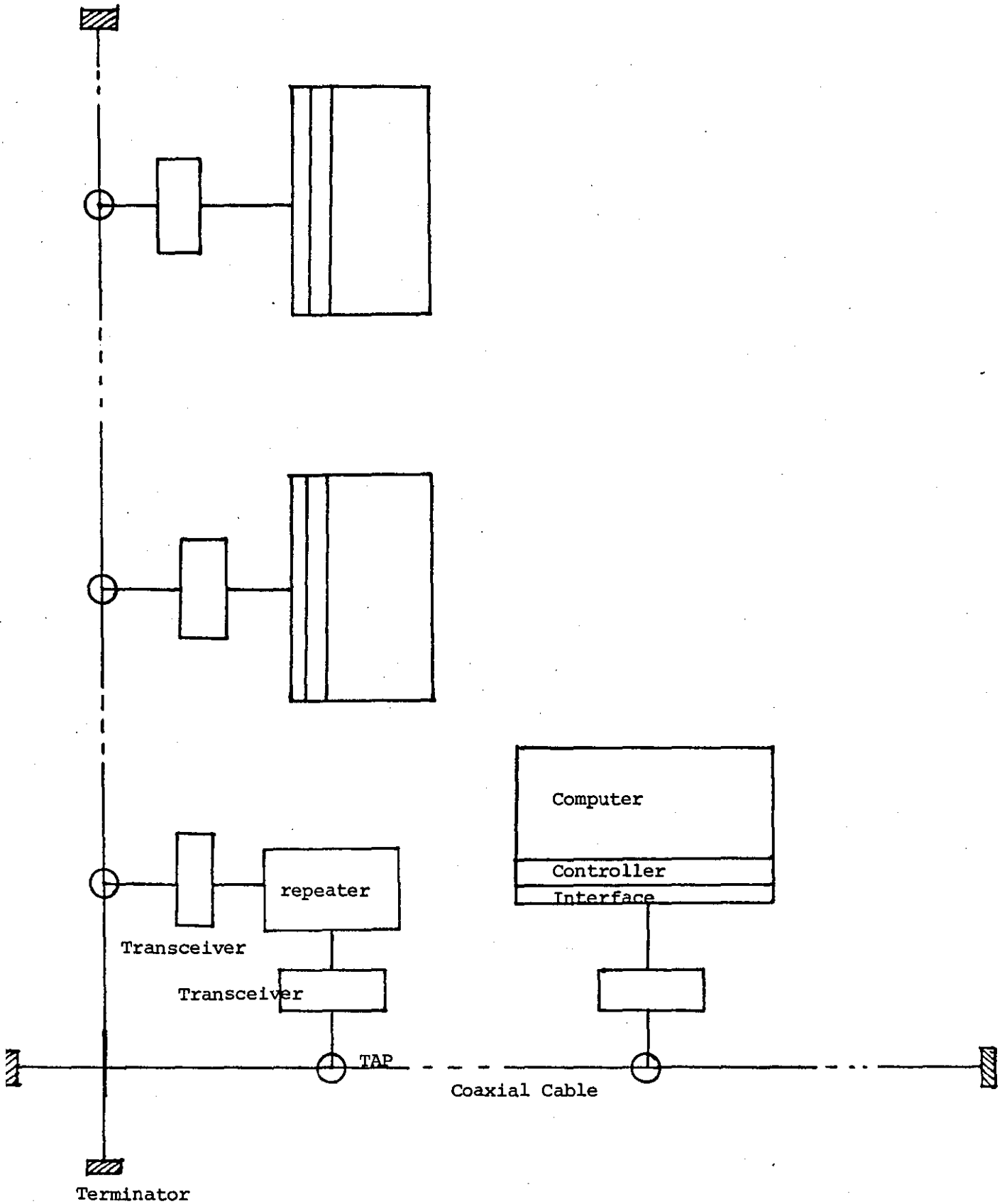


Fig. 2.9 A 2-segment Ethernet Network

The two main forms of multiplexing are the time division multiplexing (TDM) and the frequency division multiplexing (FDM) (ref. 2.108), as shown in fig. 2.10. In the type of network considered in this thesis FDM type multiplexing is not employed. TDM can be either synchronous or asynchronous. Synchronous TDM can be used with the circuit-switched Local networks (ref. 2.85, 2.87). Both synchronous TDM and FDM techniques of allocating access to the communications subnet are substantially wasteful of bandwidth, especially if the network-users require the services of the subnet infrequently. This is more so for computer communication networks in which the mode of data transmission in the subnet is bursty. Bursty data transmission (ref. 2.88) is characterised by a low duty cycle. In such a mode of data transmission asynchronous TDM is more efficient in the utilization of the communication subnet resource. Asynchronous TDM may further be subdivided into either random or controlled modes of channel access. Controlled access techniques can be further categorised depending on whether the control is centralized, in which case polling and reservation techniques can be used, or whether the control is distributed in which case token passing and reservation techniques can be used (ref. 2.74, 2.89, 2.90). Random access techniques are quite suitable for LANs. Various channel access techniques such as the pure ALOHA, slotted ALOHA, slotted ring, register insertion, CSMA and CSMA-CD can all be considered under random access (ref. 2.85, 2.88, 2.92, 2.93, 2.102).

2.4.1 Pure ALOHA Technique

Pure ALOHA channel access technique is the earliest random access protocol and was developed for UHF ground-based packet radio broadcasting networks (ref. 2.94). It is applicable to any broadcast communication subnet. In the pure ALOHA system, whenever a network-user is ready with a packet to transmit the user just proceeds and does so. In this way

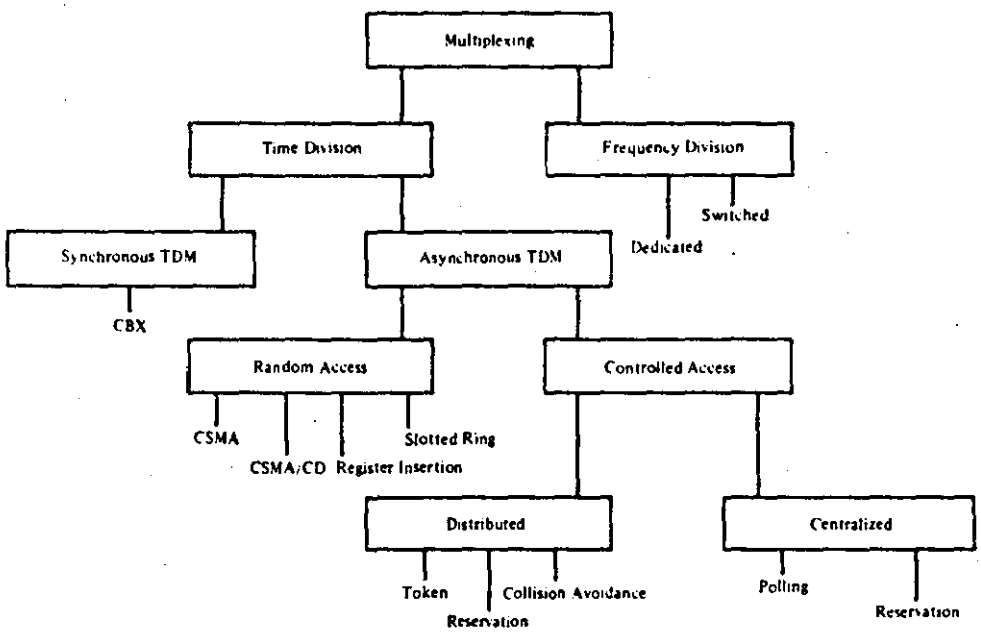


Fig. 2.10 Network Access Schemes

each user transmits the packets in an uncoordinated fashion and at times which are completely independent of packet transmission times of the other ready users. A consequence of this uncoordinated use of the communications subnet is that packets from different sources may be transmitted at the same time and therefore be involved in a collision, thereby destroying each other, and hence cannot be recognised at their destination. After transmitting the packet in a pure ALOHA fashion each user must listen for a length of time equal to the maximum round-trip propagation delay time of the network. If the user gets an acknowledgement during that period of time then the packet crossed the communications subnet safely; otherwise the user must assume that the packet was lost or damaged in transit and must retransmit the packet again. In such pure ALOHA networks the number of collisions rise very rapidly with increasing channel load and the maximum theoretical channel utilization is only about 18%.

2.4.2 Slotted ALOHA Technique

Slotted ALOHA (ref. 2.95) was a modification of the pure ALOHA system to improve the theoretical channel utilization and throughput of the pure ALOHA. In slotted ALOHA channel time is organised into uniform time slots of length equal to the packet transmission time. All network users are provided with a system synchronised clock which indicates the beginning of each transmission time slot. Users must transmit their packet only at the beginning of a time slot. In this way collisions may still occur but now the amount of channel time wasted per collision is halved to one transmission segment compared with the possible maximum of two segments of the pure ALOHA. By so doing the theoretical maximum channel utilization is doubled to about 37%.

Both pure and slotted ALOHA systems are inherently unstable under heavy channel load (ref. 2.96, 2.97, 2.107). In an unstable ALOHA system excessive traffic leads to more collisions which in turn leads to more retransmission and thus eventually useful throughput reduces to almost zero even though the channel is fully loaded.

2.4.3 CSMA Techniques

Both pure and slotted ALOHA techniques are reasonably suitable for UHF ground-based packet radio and satellite broadcast topologies in which the propagation delay between the network users is significant in comparison with the packet transmission time. In broadcast LANs employing bus and tree topologies the propagation delay is very small compared with the packet transmission time. For this reason CSMA protocol (ref. 2.92) can be used. CSMA protocol has also been used for ground packet radio networks (ref. 2.98, 2.99). In a CSMA protocol a ready user wishing to transmit a packet must first listen to the transmission channel for any on-going transmissions in progress. If the transmission channel is silent then the ready user may transmit; otherwise the channel is busy with the transmissions of another user and hence the ready user must defer his transmission for some period of time before attempting to transmit again. A successful ready user must wait for a reasonable period of time for an acknowledgement, taking into account the maximum round-trip propagation delay, and consider also that the acknowledging user must too contend for the transmission channel in order to respond. In this way much higher channel utilization than that obtained by slotted ALOHA can be achieved using CSMA. The maximum channel utilization that can be achieved depends on the packet transmission time and the propagation delay of the communication subnet.

With CSMA two algorithms have been used to specify how a ready user must behave upon finding the transmission channel busy. These two algorithms are the non-persistent and the p-persistent protocols (ref. 2.99, 2.100).

2.4.3.1 Non-persistent CSMA Protocol

In the non-persistent CSMA protocol a ready user must exercise the following steps:

Step 1 : If the channel is sensed silent, then transmit the packet.

Step 2 : If the channel is sensed busy, then wait an amount of time determined from a specified probability distribution (random retransmission delay) and repeat step 1.

In this way the probability of collisions is greatly reduced. But random retransmission times may introduce unnecessary delay in the network when there are few to moderate numbers of ready users and the collisions are few and far between.

2.4.3.2 1 - persistent CSMA Protocol

One way to improve the channel delay performance of a non-persistent CSMA protocol is to use the 1-persistent CSMA protocol (ref. 2.100). In a 1-persistent CSMA protocol a ready user must exercise the following rules:

Step 1 : If the channel is sensed silent, then with probability one transmit the packet.

Step 2 : If the channel is sensed busy, then continue to listen to the channel until it is sensed silent; in which case, with probability one transmit the packet.

Step 3 : If there is a collision, wait a random amount of time and repeat step 1.

With the 1-persistent CSMA protocol it is possible for collisions

to build up quickly and thus reduce the channel utilization and throughput.

2.4.3.3 p - persistent CSMA Protocol

The p-persistent CSMA protocol (ref. 2.100) attempts to reduce the build-up of collisions and hence improve the channel delay performance of the 1-persistent protocol. In the p-persistent CSMA protocol a ready user must obey the following rules:

Step 1 : If the channel is sensed silent, then with probability p transmit the packet, and with probability (1-p) defer transmission for one time unit. The time unit is usually equal to the maximum propagation delay.

Step 2 : If the channel is sensed busy, then continue to listen until the channel is sensed silent and repeat step 1.

Step 3 : If the transmission is delayed on time unit, then repeat step 1.

The p-persistent CSMA protocol can be made adaptive by choosing appropriate values of p. If the number of ready users in the network is small, considerably high values of p can be used. This would have the net effect that the performance is very nearly like that of the 1-persistent CSMA protocol. On the other hand if the number of ready users is high the value of p can be reduced to keep the number of collisions and the time delay in the channel to a desired minimum. This can be achieved by making the product of the number of ready users and p less than one.

2.4.4 CSMA-CD Protocols

Pure CSMA protocols are sometimes referred to as listen before talk (LBT) protocols (ref. 2.103). CSMA-CD protocols on the other hand are referred to as listen while talk (LWT) protocols (ref. 2.104, 2.105). The CSMA-CD protocols are particularly suitable for bus or tree broadcast LAN topologies (ref. 2.101. The CSMA-CD protocol is a modification of the pure CSMA protocol. In the CSMA protocol, when two ready users are

involved in a collision the transmission channel remains unusable for the entire duration of the transmission of both damaged packets. This wasted bandwidth can be quite considerable if the packets are long compared to the channel round-trip propagation delay. The CSMA-CD protocol attempts to reduce this wasted channel bandwidth by requiring each successful ready user to continue to listen to its own transmission and exercise the following rules:

a) If a collision is detected during transmission, immediately back-off and cease transmitting the packet, and transmit a jamming signal briefly to let all other users know that there has been a collision.

b) After transmitting the jamming signal, wait a random amount of time before attempting to retransmit the packet.

After a collision, the colliding ready users need to exercise a packet retransmission algorithm. Both the 1-persistent and the p-persistent algorithms can be used with CSMA-CD. But the 1-persistent algorithm is found to be more suitable for CSMA-CD protocol because the users involved in a collision using this protocol back-off a random amount of time (ref. 2.104). Binary exponential back-off algorithm has also been used (ref. 2.68) in ethernets to maintain stability. In the binary exponential back-off algorithm a ready user may attempt to retransmit the packet repeatedly in the face of repeated collisions, but after each collision the mean value of the retransmission delay is doubled. Eventually after sixteen unsuccessful retransmission attempts, the ready user must give up and report an error.

Many advantages have been attributed to the CSMA-CD protocol (ref. 2.68, 2.104). The IEEE 802 CSMA-CD standard has been developed (ref. 2.80) and is very close to that of the ethernet.

Some of the advantages of the CSMA-CD protocol are:

a) It uses a simple algorithm.

b) It is widely accepted.

c) It provides a fair access to the transmission channel.

d) It provides good delay and throughput performance at low and medium channel load.

But its main disadvantages include:

a) It exhibits complex collision detection hardware and software.

b) It exhibits poor performance under heavy load.

c) It exhibits a bias for long transmission.

d) It exhibits poor error faults diagnostic problems.

e) It specifies a requirement for a minimum packet size.

The random channel access protocols based on the CSMA have been a subject of intense research (refs. 2.68, 2.98, 2.100, 2.106, 2.107).

2.5 NETWORK SWITCHING AND ROUTING

As has been explained earlier all the users in the network connect to and share the communications subnet. The purpose of the subnet is to provide the necessary switching and transmission techniques to transport data from a ready user to any other user in the network. It is the primary function of the switching to provide the ready users access to all the others in the network. The type of switching employed is controlled by the user by specifying the data destination address to the switching mechanism which in turn routes the data to the specified destination.

Three different types of switching mechanism can be used: circuit switching, message switching and packet switching.

2.5.1 Circuit Switched Networks

The main characteristic of circuit switching (ref. 2.38, 2.86, 2.110) is that a dedicated communication path needs to be established between the two users in the network. Hence in circuit-switched networks a physical connection between the two users is necessary. One or more links in the physical communication path may need to be thus established in the subnet. Communication by means of circuit switching involves three distinct procedures:

a) An end-to-end circuit establishment.

b) Data transfer.

c) Circuit disconnection.

Thus, the end-to-end circuit establishment has to be done before data can be transferred. If the data transmission is bursty, then circuit switching can be inefficient and wasteful of channel capacity. Also the end-to-end circuit establishment may involve long delay in computer communication networks and hence interactive traffic may be cumbersome.

2.5.2 Message Switched Networks

In message-switched networks (ref. 2.112) it is not necessary to establish a dedicated physical path between any two users in the network. A ready user wishing to send a message to any other user in the network only needs to append a destination address to the message. In this context, a message is a large block of data and may consist of many tens of thousands of bits. The message is routed link by link through the network by the intermediate users. At each of these intermediate users the message is received in its entirety and may be temporarily stored before being transmitted along the next link, in the path, in a store-and-forward fashion. In this way detailed routing algorithms are needed by

each user to avoid congestion and to ensure safe arrival of the messages. The main disadvantage of the message-switched networks is that very long message transmission delay may be encountered and hence is not very suitable for interactive computer communication traffic.

2.5.3 Packet Switched Networks

Packet switching attempts to combine the advantages of message and circuit switching. Packet switched networks (ref. 2.28, 2.113) are essentially similar to message switched networks. The main difference is that in packet-switched networks packets rather than messages are the units of data transmitted across the communications subnet. The packets are smaller units of data with up to a few thousand bits. Messages much longer than the packets have to be broken down and reorganised into several packets. This method of breaking down larger message blocks into several smaller packets for transmission has a profound effect on the network delay performance and transmission channel utilization and throughput is enhanced and is also more efficient in moderate to heavy traffic. Both datagram and virtual circuit (ref. 2.115) packet switching can be employed in packet-switched networks.

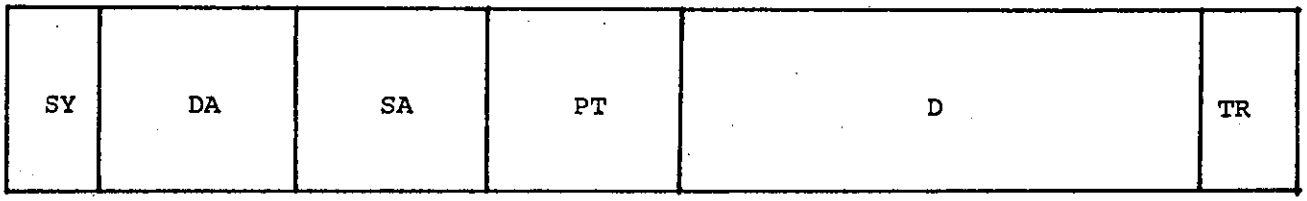
The various switching techniques have been used quite effectively in various network topologies of the communications subnet (ref. 2.93, 2.114). This thesis is primarily concerned with packet-switched LAN networks. In packet-switched LANs there is no necessity for intermediate switching and hence the issues of complex routing techniques do not play an important part. Also, for the correct functioning of packet-switched computer LAN networks the network architecture layers 1, 2 and 3 are necessary (ref. 2.101, 2.116).

2.5.3.1 LAN Packet Format : Ethernet

The basic prototype ethernet packet format (ref. 2.68) is shown in fig. 2.11. The packet format begins with a packet of synchronisation bit pattern of length one bit and whose leading edge enables the controller interface of the receiving user to detect the start of the packet and to acquire bit phase. Next follows two 8-bit fields which define the destination and source addresses respectively. This is then followed by a 16-bit word to identify the type of packet. Next follows several 16-bit words of data which in turn are followed by the 16-bit cyclic redundancy check (CRC) pattern.

The broadcast packets are copied into the memory of each network-user under the control of an address filter which can be implemented in microcode. On receiving the first word of the packet the microcode compares the destination user field against the address supplied by the software and then copies the packet into memory only if the addresses are equal. It is also necessary for the software to set the address filter to be selective to enable the user to copy selected packets received into its memory. This is useful for network monitoring and feedback especially in a distributed computation environment considered in this thesis. Such feedback mechanism is necessary for use by the larger and more powerful computer to announce to every user in the network the amount of workload existing at the large computer. The knowledge by the smaller computers of the workload at the larger computer will in turn enable them to decide better whether or not to share any computation with the larger computer. Hence with this feedback capability the workload in the network can adjust itself and be more uniformly distributed.

In a distributed computation environment the main commodities exchanged between the smaller computers and the more powerful large computer in the network are:



SY = Synchronisation (1 bit)

DA = Destination Address (8 bits)

SA = Source Address (8 bits)

PT = Packet Type (16 bits)

D = Data (0 - 4000 bits)

TR = CRC

Fig. 2.11 Packet Format

a) the program modules

b) the intermodule parameters and other data

c) the results of the computation.

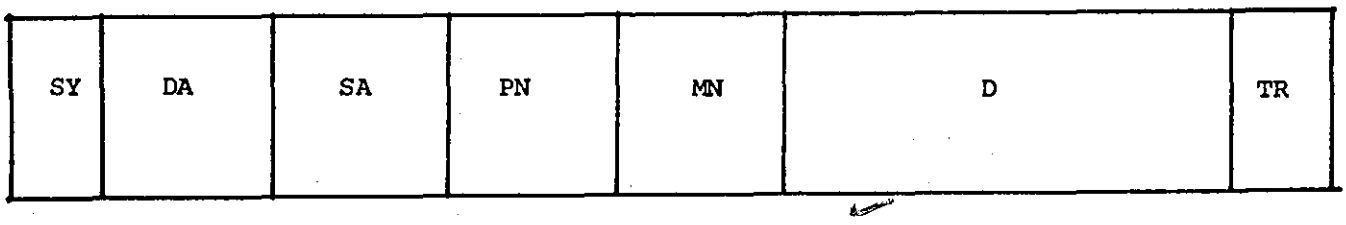
The smaller computers partition and transport the program modules to the more powerful large computers for processing. During module processing there will be interaction between the large and the small computers when intermodule parameters and other data will be organised into packets and transported across the communications subnet, fig.2.12(a). In such an environment it is possible to simplify further the format for the network packets. Fig. 2.12(b) shows the format of the packets transported from the small computers to the large computers, while fig. 2.12(c) shows the format of the packets transported from the large computers to the small computers. The packet format in a distributed computation environment has the extra two 8-bit fields to identify the program and program module. From this it can be seen that the module packets may contain thousands of bits. But large packets enable the communication channel to be utilized more efficiently.

2.5.3.2 LAN Packet Format: The IEEE 802 Standard

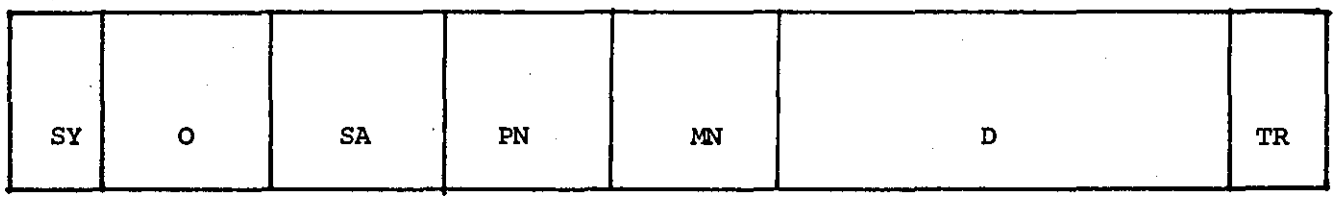
IEEE has produced a draft IEEE 802 standard for LANs (ref. 2.80). These standards are in the form of the 3-layer network communications architecture and with tree-like expansion capability, fig. 2.13. The three layers are derived from the lowest two layers (the data link layer and the physical layer) of the OSI reference model as they apply to the specific characteristics of communication within LAN. The two main characteristics pertaining to communications within LANs are:

a) data are transmitted in addressed packets.

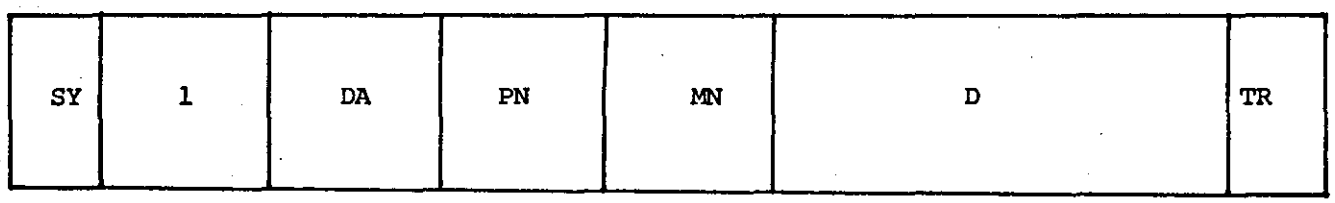
b) there is no intermediate switching and hence routing is not necessary.



(a) Basic Module Packet Format



(b) Source-to-sink packet format



(c) Sink-topsource packet format

PN = Program Number (8 bits)

MN = Module Number (8 bits)

Fig. 2.12 Module Packet Formats

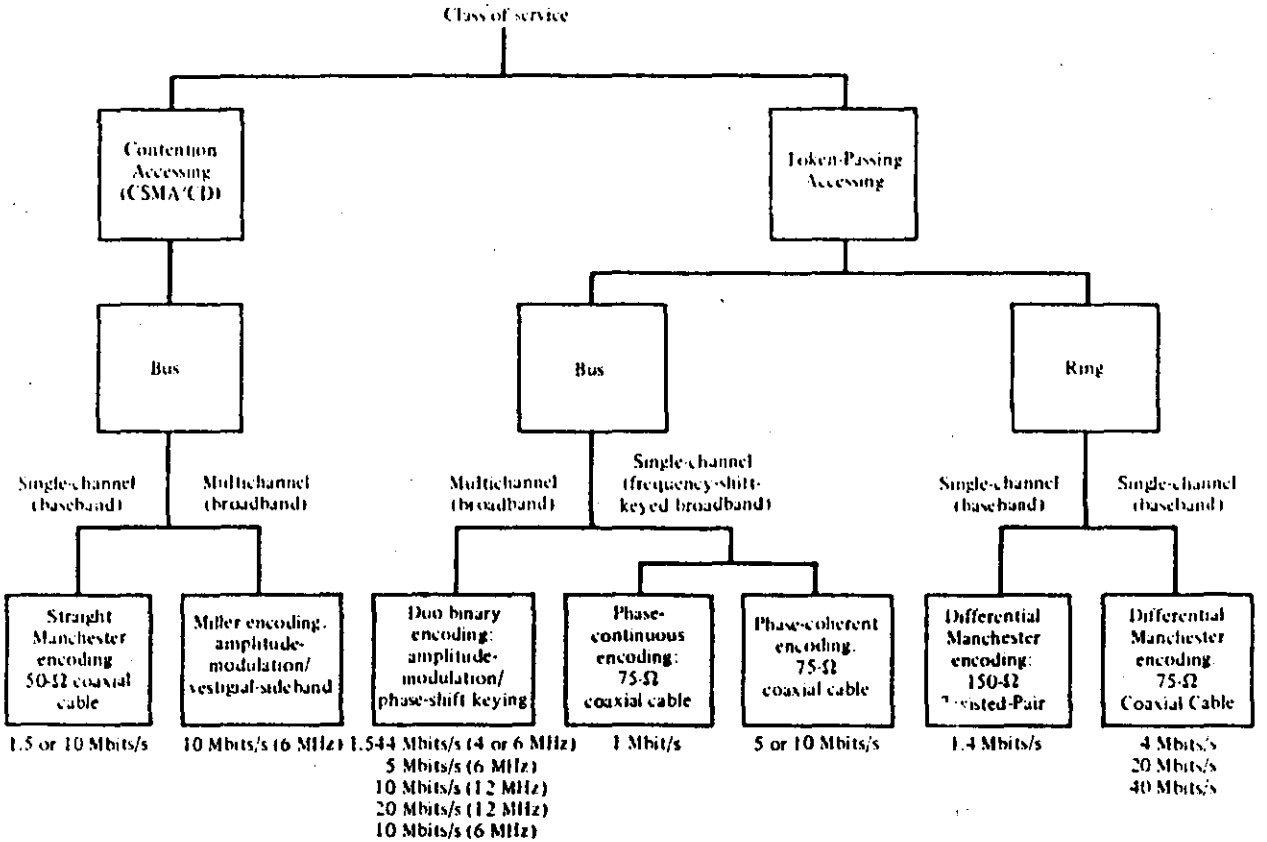


Fig. 2.13 IEEE 802 LAN Standard

These two characteristics lead to the three LAN layers: the Logical Link Control (LLC) layer, the Medium Access Control (MAC) layer and the Physical layer.

2.5.3.2.1 The LLC Layer

The functions of the LLC layer include

a) To provide one or more Service Access Points (SAP). A SAP is a logical interface for the connection and exchange of data between two adjacent layers.

b) To assemble data into frames with address and CRC fields for transmission.

c) To disassemble frame and perform address recognition and CRC validation on reception.

2.5.3.2.2 The MAC Layer

The main function of the MAC layer is primarily to manage communication over the link and to exercise the CSMA-CD channel access protocols.

2.5.3.2.3 The Physical Layer

As in the OSI model, the functions of the LAN Physical layer include

a) Encoding and decoding of signals.

b) Preamble generation and removal for synchronisation.

c) Bit transmission and reception.

Fig. 2.14 shows the communication architecture as it applies to LANs. The frame format for the IEEE 802 draft standard are the basis for the LLC, MAC and the Physical layer functionality as shown in fig. 2.15.

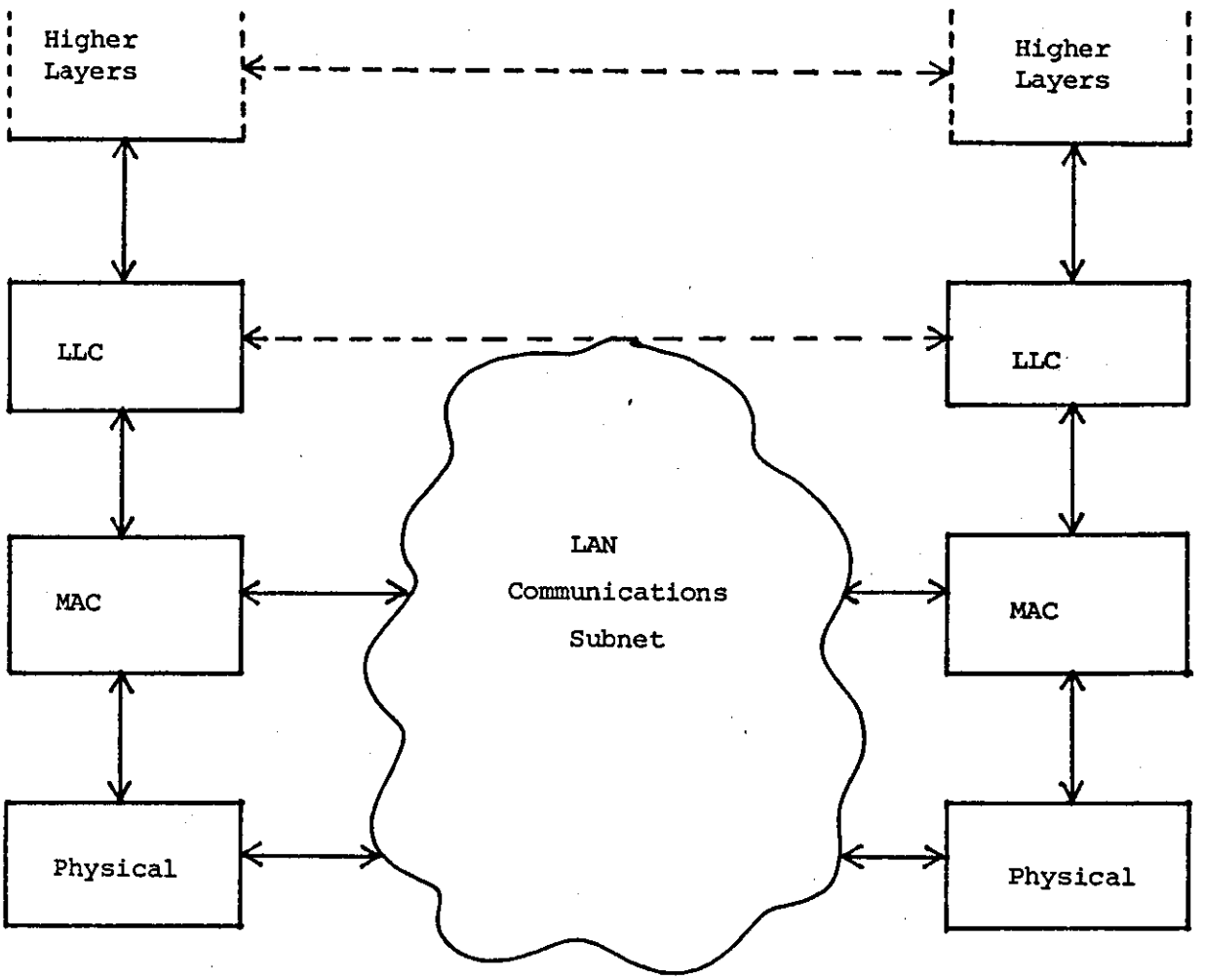
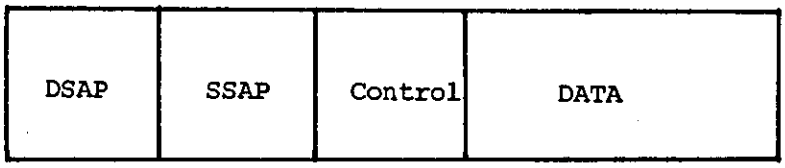
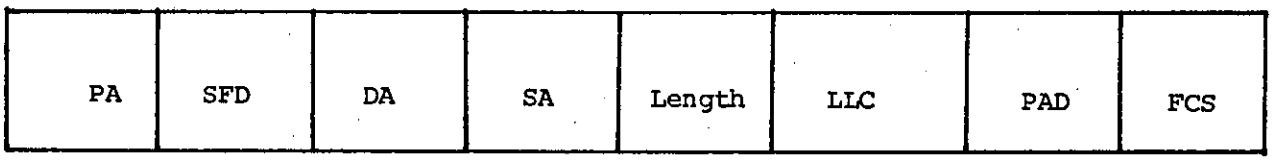


Fig. 2.14 LAN Communication Architecture

LLC frame format



MAC frame format



DSAP = Destination Service Access Point (8 bits)

SSAP = Source Service Access Point (8 bits)

Control = Control (8 bits)

PA = Preamble (8 bits)

SFD = Start Frame Delimiter (1 bit)

DA = Destination Address (2 - 6 bits)

SA = Source Address (2 - 6 bits)

length = length (2 bits)

LLC = LLC (0 - 1500 bits)

FCS = Frame Check Sequence (4 bits)

Fig. 2.15 IEEE 802 Frame Formats

CHAPTER 3

PROGRAM STRUCTURE AND PARTITIONING

3.1 INTRODUCTION

If a computer system's hardware provides both capabilities and limitations, then programs provide the flexibility. General-purpose computers are programmable and are designed to solve a variety of different types of problems. Every program that runs on the computer directs the vast power of the computer towards solving a particular problem. The fact that a computer is programmable is the most important element in computer design and is probably the most important legacy of Von Neumann (ref. 3.1). The major problem in developing programs for a general-purpose computer is to bridge the gap between the nature of real user problems and the way the computers solves the problems (ref. 3.13). This gap has continued to be bridged by the development of better and more problem-oriented languages that both the programmer and the computer can understand. In this way the programmer can write more and better programs and hence spend more time thinking about problems and less time worrying about details that, although important to the computer, are largely irrelevant to the solution of a problem. One of the recent developments in such languages is the introduction of the concept of structured programming (ref. 3.2, 3.3). By the use of structured programming large and very complex problems can be simplified and tackled.

This chapter examines how a problem can be expressed as a structured program and how such a program which is a candidate for distributed computation within a LAN is organised into smaller program modules. It also examines how, once the program has been organised into modules, a partitioning algorithm can be applied in order to schedule the modules and distribute the computation between different processors.

3.2 PROGRAMMING LANGUAGES

A program is a series of instructions that cause a computer to perform a particular task. Programmable general-purpose computers have been and can be programmed in three different types of languages: machine language, assembly language and high-level languages. Large computers generally use the high-level programming languages such as FORTRAN, PL/1, ALGOL, BASIC and PASCAL and the smaller computers have increasingly continued to adopt such high-level languages instead of the machine and assembly languages (ref. 3.4, 3.5). High-level languages are easier to write because they are problem-oriented rather than machine-oriented. Each statement in a high-level language performs a recognizable function and it will generally correspond to many assembly language instructions. A common estimate is that a programmer can write a program about ten times as fast in a high-level language as compared to an assembly language (ref. 3.6). But one of the major drawbacks in the use of high-level languages on the smaller computers is that they need translators or compilers to translate or compile the source programs written in the high-level language into the object machine language program which the computer can execute. High-level languages do not generally produce very efficient machine language programs. The translators are generally slow and compilers tend to be expensive and use a large amount of computer memory. While most assemblers occupy from about 2K to 16K bytes of memory, compilers occupy from about 4K to 64K bytes (ref. 3.6). So the amount of overhead involved in using the compiler is rather large. But good compilers generally speed up the program execution time. Applications that are better suited to high-level languages are those that require large memories. Hence a large program

will greatly enhance the advantages of high-level languages. With the falling cost of the memory chips and the increasing use and efficiency of high-level languages the few disadvantages of using these high-level languages on the smaller computers will no longer be very significant.

Some of the major characteristics of the three types of programming languages referred to above are now briefly examined.

3.2.1 The Machine Language

Virtually no one programs in machine language. Its use cannot be justified considering the low cost of an assembler and the increase in programming speed an assembler provides. The main difficulties associated with programming in the machine language are:

a) The programs are long, tiresome, confusing and difficult to write.

b) These binary machine language object programs are difficult to understand or debug.

c) The programs are difficult to enter since each bit must be entered individually.

d) The programs do not describe the problem which the computer is to perform in anything resembling a familiar human-readable format.

e) The programmer tends to make many careless errors that are difficult to locate.

3.2.2 The Assembly Language

One way to achieve programming improvement is to assign a name to each instruction code by the use of mnemonics. Such an instruction mnemonic should describe in some way what the instruction does. Assembly language uses such instruction mnemonics and hexadecimal numbers and thus greatly improves the programming effort. The source assembly language

is converted into the object machine language by the assembler program. Both the smaller and the large computers can employ assembler programs, but the smaller computers generally have much simpler assemblers than do the larger computers (ref. 3.7). Some of the main features of using the assembly language and assembler programs are:

a) They allow the programmer to assign names to memory locations, input and output devices and even to sequences of instructions.

b) They convert data or addresses from various number systems such as the decimal and the hexadecimal into binary and also converting characters into their ASCII or EBCDIC binary codes.

c) They perform some arithmetic as part of the assembly process.

d) They help in directing the loader program where in the memory certain parts of the program or data should be stored.

e) They enable the programme to assign areas of memory as temporary data storage and to store fixed data in areas of program memory.

f) They provide the information required to include standard programs from the program libraries, or programs written at some other time, in the current program.

g) They allow the programmer to control the format of the program listing and the input and output devices used.

However, programming in assembly language is still a tedious and time-consuming job. This is made even more difficult by the fact that the programmer must have a detailed knowledge of the particular computer to be used. Also, assembly language programs are not very portable. The other main important features that favour the use of the assembly language are that they are suitable for:

a) Short to moderate size programs.

b) Applications where the memory cost is a major factor.

c) Real-time control applications.

d) Limited data processing.

e) High-volume applications.

f) Applications requiring more input/output (I/O), or control than computation.

3.2.3 The High-Level Languages

The solution to many of the difficulties associated with assembly language programs have been largely overcome by the use of high-level or procedure-oriented languages, because they are more problem-oriented and less machine-dependent (ref. 3.5, 3.8, 3.9), as mentioned earlier. The main advantages of using these languages are that

a) They provide a more convenient description of the problems and tasks.

b) They provide more efficient program coding.

c) They enable easier documentation.

d) They provide standard syntax.

e) They are less dependent on the organisation of a particular computer.

f) They are portable.

g) They enable the provision of library routines and other programs.

h) They are flexible and can be modified to handle structured data and control.

But they also have the disadvantages in that

a) They require special rules.

b) They tend to require extensive hardware and software support.

c) They tend to be tuned to a particular application.

d) They have a tendency to be inefficient.

e) They exhibit a difficulty in optimizing code to meet speed and memory requirements.

f) They show an inability to use special features of a computer conveniently.

However, they tend to be quite suitable for

a) Long programs.

b) Applications requiring large memories.

c) Low-volume applications requiring long programs.

d) More computation than input/output, (I/O), or control environment.

e) Compatibility with similar applications using larger computers.

f) Availability of specific programs in a high-level language which can be used in the application.

Many other factors in the decision concerning the particular programming language to use are also important and need to be taken into account. But a trade-off of the various factors involved has to be weighed carefully. If the hardware, for example, is the largest factor or if the speed is critical, then, for some applications, assembly language should be favoured. But limitations in hardware may mean a major software development and support in exchange for the lower memory costs and higher execution speeds. On the other hand, if software is the major factor in an application, then the high-level language should be favoured. But because of the continuing developments in the microelectronic technology and in computer software, the future can be expected to continue to favour the use of high-level languages (ref. 3.5), and also because

a) Programs can always be expected to continue to add more features and hence will grow larger.

b) Hardware and memory are becoming less expensive.

c) Memory chips are becoming available in larger sizes, at lower "per bit" cost.

d) More versatile compilers are becoming available.

e) More suitable and more efficient high-level languages are being developed.

f) More standardization of high-level languages can be expected to occur.

g) Software and programmers are becoming more expensive.

h) More and more specific program packages for libraries will continue to be written and stored in a data-base environment.

i) The general tendency now is for decentralization and hence more standardization.

3.3 PROGRAM STRUCTURE

The final program structure of a program which is run in a distributed computation environment within LAN is only a part of a larger program design, or problem-solving, process. A problem has first to undergo a problem-solving phase before it can be coded and run on a computer. Program design is the stage in which the problem definition is formulated as a program (ref. 3.8, 3.9). If the program is small and simple, this stage may require relatively little effort. But if the program is large or more complex, the program designer has to consider more elaborate methods. In general, the problem-solving process may be divided into four phases:

a) Defining the problem.

b) Analysing and developing an algorithm to solve the problem.

c) Implementing the solution through the design and development of a computer program.

d) Debugging, testing, documenting, and maintaining the program over time.

In the problem definition phase some of the important factors to take into account are

a) The specification of the output: i.e., precisely what is to be output by the program.

b) Information needed to solve the problem: i.e., what data must be available to produce the required output.

c) The specification of the processes needed to solve the problem: i.e. the formulae and sequence of actions to be used to solve the problem.

Once the problem has been defined, it is then necessary to develop an algorithm to solve the problem. An algorithm is a logical sequence of unambiguous operations that, when carried out, lead to the solution of the problem specified. An algorithm is essential in the solution of a problem using a computer. Two or more algorithms may exist for the solution of one problem. Hence, it is also necessary to determine and choose the most efficient algorithm. An algorithm may be efficient in the time taken to execute it on the computer or it may be efficient in terms of its storage in the computer memory. Two of the most common ways of representing algorithms are by means of

a) Flowchart.

b) Pseudocode.

Thus, algorithms can be carried to the desired level of detail. Flow-charting is the oldest and better-known method of analysing and developing algorithms and it has the basic advantage that it provides the programmer with a pictorial representation of the entire program structure. But,

one of the major drawbacks of flowcharting is that it allows for unstructured design (ref. 3.10). The lines and arrows of the flowchart, backtracking and looping all over the chart are the antithesis of good structured design principles. Hence, the pseudocode is becoming increasingly popular in the structured design of large or complex programs (ref. 3.11, 3.12). Also, with the pseudocode design method it is easier to employ the following programming methods which provide a unified approach to the program design process:

a) Structured programming

b) Top-Down design

c) Modular programming

3.3.1 Program Modules

Once programs become large and complex, the method of flowcharting is no longer a satisfactory program design tool as mentioned above. However, the problem-definition phase and the flowchart can be used together to give a good idea as to how the program structure can be organised into reasonably sized sub-tasks or program modules (ref. 3.14, 3.15, 3.39). The division of the entire program into such modules is called modular programming. The major aim of modular programming is how to organise the program into modules and how to put the modules together. A program module is basically an autonomous program unit that performs a well-defined task necessary to the completion of the larger program (ref. 3.16). The main advantages of modular programming are:

a) A single module is easier to write, debug and test than the entire program.

b) A module is likely to be used in many places within the same program and in other programs, particularly if it is reasonably general

and performs a common task. In this way, a library of standard modules can be built and used in a resource sharing, data-base, or distributed computation environment.

c) Through modular programming, the programmer can divide tasks, use previously written programs, and thus simplify his task and shorten the time to solve his problem.

d) It is easier to introduce changes into one module rather than into the entire program.

e) It is easier to isolate and locate errors in modules than in the entire program.

f) Through modular programming, it is easier to build a better picture of how much progress has been made and how much work is left.

g) It is possible to use modules written in a different programming language.

h) High quality modules written by specialists in particular fields can be resource shared.

The main disadvantages of modular programming on the other hand, include

a) If the modules are written by many different people or if they have undergone many changes over a long period of time, fitting the modules together can be a major problem.

b) If the modules are many they will require very careful documentation since they may affect other parts of the program, such as the global variables and data structures used by all the modules.

c) Testing and debugging modules separately may be a difficult exercise, since other modules may produce the data used by the modules being debugged and still other modules may use the results. This may necessitate writing of the special driver programs just to produce sample data to test the modules. This driver program requires

extra programming effort that adds nothing to the original exercise.

d) The original programs may be difficult to modularize. If it is then modularized poorly, integration will be difficult since most of the resulting errors and changes will involve several modules.

e) In some cases modular programs may require extra processing time and memory, especially if the separate modules repeat functions.

Considering the above advantages and disadvantages for organizing the program into program modules it can be seen that important considerations should include restricting the amount of information shared by the modules, limiting design decisions that are subject to change to a single module and also restricting the access of one module to another (ref. 3.15, 3.16). A major drawback in modular programming is that there are no proven, systematic methods for modularizing programs. But a few principles for modularizing programs can be identified (ref. 3.15, 3.16, 3.17), because they lead to a realization of better and more autonomous modules:

a) Modules should be distinct and should perform one logically coherent task and nothing more. A good rule of thumb is that: if it takes more than one sentence to describe what a module does, then the module does too much.

b) Modules are autonomous units of a program. They should receive only data that are necessary to perform their specific task, and they should perform their task in such a way that only those data values that need changing are changed.

c) Modules should be relatively short, usually containing fewer than about 100 lines of code.

d) Those modules that reference common data should be parts of the same overall module.

e) In the case of two modules in which the first uses or depends on the second, but not the reverse, such modules should be separate.

f) A module that is used by more than one other module should be part of a different overall module from the others.

g) Two modules in which the first is used by many other modules and the second is used by only a few other modules should be separate.

h) Two modules whose frequency of usage are significantly different should be part of different modules.

i) The structure or organisation of related data should be hidden within a single module.

Inherent in these principles of modularization is the fact that if it is found difficult to modularize the program, then it is strong indication that the problem itself is poorly defined, and hence a re-definition is necessary. For example, too many special cases, each requiring special handling, or the use of a large number of variables, each requiring special processing, are problems that can be most efficiently handled by redefining the problem. Simple tasks should not be modularized. Once the task has been organised into distinct and logically separate modules the methods of structured programming and top-down design can be applied.

3.3.1.1 Structured Programs

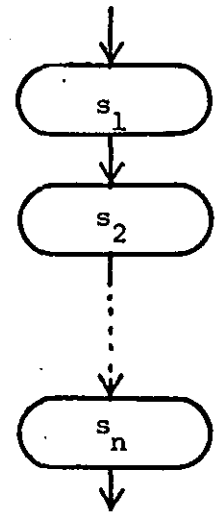
One way of ensuring that the modules are distinct and logically separate program units is accomplished by utilizing the recent design methods of structured programming (ref. 3.2). Pascal is one of the more recent block-structured and procedure-oriented high-level programming language that is based on the concept of structured programming. In such a structured programming environment both the data and control are

organised in a highly block-structured way.

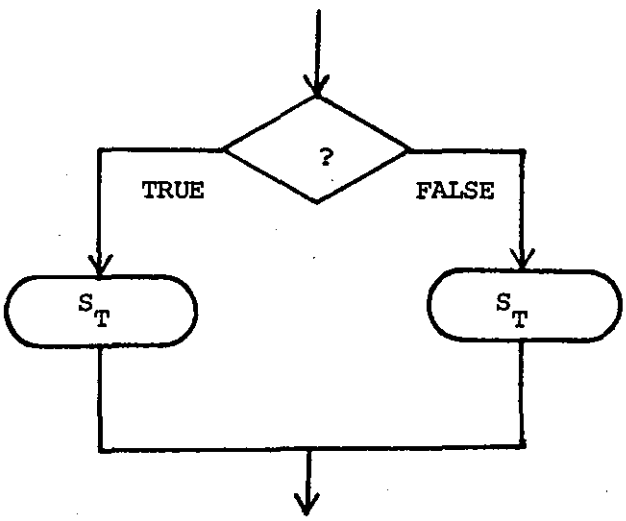
Structured data are organised as abstract data types (ADT), (ref. 3.18), which are defined by Pascal type definitions and the operations associated with these ADT are defined by Pascal procedures and functions when the final program is created. In this way the ADT can be thought of as a mathematical model with a collection of operations defined on that model. Sets of integers, together with the operations of union, intersection, and set difference, is a simple example of ADT. ADTs are generalizations of the primitive data types such as integer, real, boolean, etc. In this way the ADT encapsulates a data type in the sense that the definition of the Pascal type and all operations on that type can be localized to one section of the program. Hence in a high-level language the significance of an item of data is expressed in the type it belongs. By specifying the type of a variable the programmer defines the set of values that can be assumed by the variable. One of the main advantages of high-level language over assembly languages is that the former provide types that correspond to the concepts of their particular application area. Thus, whereas an assembly language program has to manipulate items of data at the bit-pattern level, a high-level language program manipulates atomic items of data such as integers or reals. In block-structured high-level languages data types can be user-defined.

Similarly, control is also structured. A structured program is defined as a program with single-entry and single-exit control structures of fig. 3.1. The simplest single-entry single-exit control structures that have been found to be sufficiently powerful to construct any computer program are:

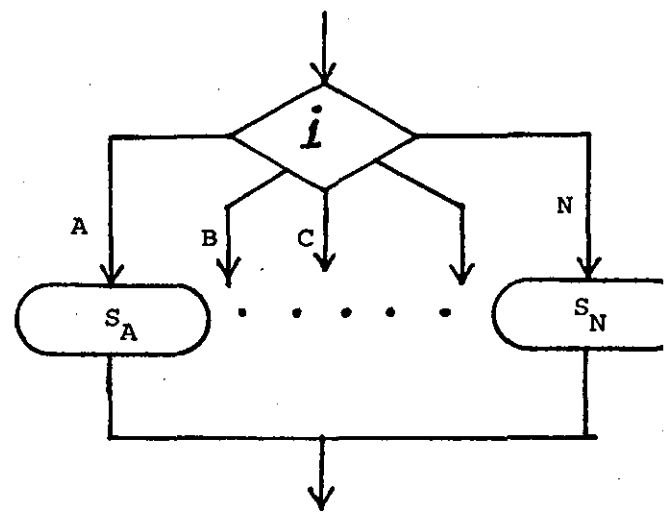
a) Concatenation structure: This is a linear structure in which the statements or elements of the structure are executed strictly sequentially and consecutively, fig. 3.1(a).



(a) Concatenation Control Structure

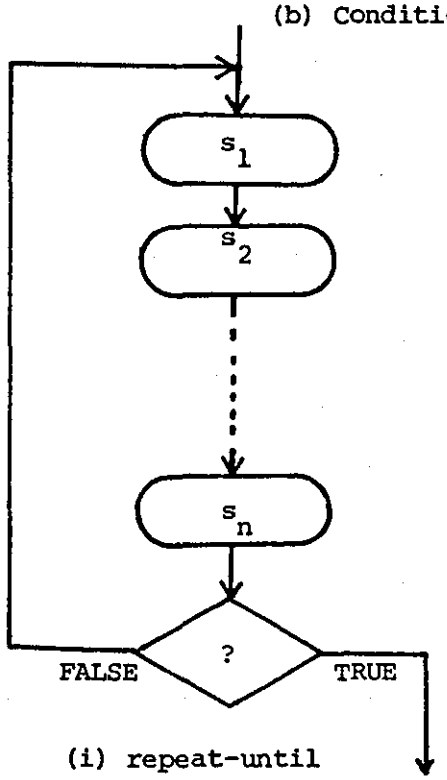


(i) if-then-else

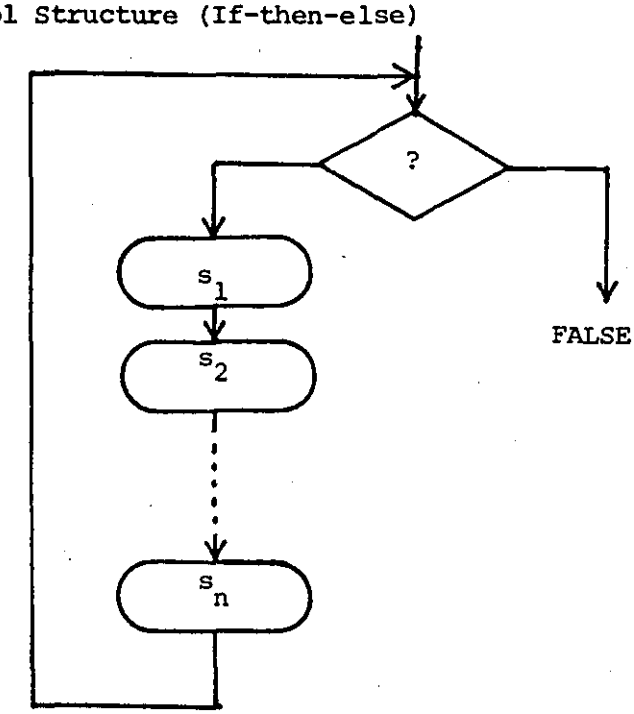


(ii) case i of A, B, C ...

(b) Conditional Control Structure (If-then-else)



(i) repeat-until



(ii) Do-while

(c) Iteration Control Structures

Fig. 3.1 Basic Control Structures

b) Conditional structure: This is a structure in which control branches from a single point into two or more paths; then all paths merge into a single point of exit, fig. 3.1(b).

c) Iteration structure: In this structure control repeatedly passes through one or more inner structures and then finally exit to a single point, fig. 3.1(c).

The following important features of structured programming are prominent:

a) Each structure has a single-entry point and a single-exit point.

b) Only the three basic control structures, and possibly a small number of auxiliary structures are permitted.

c) The structures may be nested to any desired level of complexity so that any program can, in turn, contain any of the structures.

The following are some of the main advantages of structured programming, (ref. 3.2):

a) The number of the control structures is limited and hence it is easier to standardize the terminology.

b) The sequence of the operations performed is simple to trace and hence it is easier to debug.

c) The control structures can easily be used to form modules.

d) It has been shown that the given set of structures is complete and hence all programs, irrespective of their complexity, can be written in terms of the three structures.

e) The indented structured version of a program is partly self-documenting and fairly easy to read.

f) Structured programs are easy to describe with program outlines.

g) Structured programming has been shown, in practice, to increase programmer productivity.

h) Structured programming often makes the programmer aware of inconsistencie or unlikely combination of inputs.

i) Structured programming allows the use of meaningful programmer-defined identifiers freely.

The main drawbacks associated with structured programming, however, are as follows:

a) Only a few high-level languages (e.g. Pascal, PL/M) will directly accept the structures. If the program is needed in the assembly language format, the programmer has to go through an extra translation stage to convert the structures into the assembly language. But other high-level languages (e.g. Fortran, Basic, etc.) are slowly adopting the principles of structured programming.

b) There is a likelihood that the structured programs will use more memory and execute more slowly than their unstructured counterparts.

c) Limiting the control structures to just three basic forms may sometimes make some tasks very awkward to perform. The fact that the three control structures are complete and that all programs can be implemented with them does not necessarily mean that a given program can be implemented with them effectively or conveniently.

d) Multiple nested control structures, such as the if-then-else, can often be very difficult to read.

e) The program flow of control may not correspond with the program flow of data and hence the control structures may handle data awkwardly.

Despite the disadvantages of structured programming mentioned above, it is one of the few methods of systematizing program design. It is found most useful in such situations as in the

a) Application in which memory usage is not critical.

b) In applications involving large programs, perhaps exceeding 1000 instructions.

c) Low-volume applications in which the software development costs, particularly testing and debugging, are important design factors.

d) Applications involving string manipulations, process control, or other algorithms rather than in simple bit manipulations.

More and more high-level languages such as BASIC and FORTRAN are incorporating the concepts of structured programming (ref. 3.19). This is mainly due to the recognition of the advantages to be gained by implementing program design using structured programming. Also the per-bit memory cost is decreasing and most of the drawbacks cited for modular programming, as well as the structured programming, methods will lessen in significance. As the cost of memory continues to decrease, the average size of many microcomputer programs increase, and the cost of software development continues to increase, structured programming, which tends to decrease software development costs for larger programs but use less memory, will become more valuable.

3.3.1.2 Top-Down Design

Larger and more complex problems are most easily solved by breaking them into smaller problems and then, if necessary, breaking these smaller problems into even smaller subproblems pieces (ref. 3.20, 3.21). This process of taking a problem and successively breaking it down into its component parts is referred to as top-down design or stepwise refinement and is fundamental in any problem solving process. In this way, the original problem is solved in steps (ref. 3.17). Each step is essentially a small refinement of how the problem is to be solved. This method of problem solving by stepwise refinement is carried out until one arrives

at a program the meaning of whose steps are formally defined by a programming language manual. Fig. 3.2 illustrates graphically this hiearchical process of stepwise refinement, in that the solution to the original problem P is accomplished by solving progressively smaller subproblems (P_1, P_2, P_3, etc.) and, if necessary, solving even smaller and smaller sub-subproblems (P_{11}, P_{21}, etc.) and (P_{111}, P_{112}, etc.), etc.

Hence, in the program design stage, several techniques can be used to systematically specify and document the logic of the program. Modular programming provides the programmer with the techniques of dividing the total program into the smaller, distinct and logically separate program modules. Structured programming provides a systematic way of defining the logic of those modules, while the top-down design facilitates a systematic method for further refinement, integrating and testing them. These three techniques provide a unified approach to program design or problem solving process.

3.3.1.3 Subroutines and Pascal Procedures

Subroutines and Pascal procedures (and Pascal functions), like program modules, are program units that perform well-defined tasks necessary to the completion of a larger problem. In this context, program modules on the one hand, and subroutines and procedures on the other hand, can be used interchangeably and can be considered to be equivalent. However, in some situations a program module may have more than one subroutine or procedure. Hence subroutines, procedures or functions provide one of the most powerful tools for solving complex problems.

Pascal procedures, an essential tool in programming, generalize the concept of an operator. Instead of being limited to the built-in

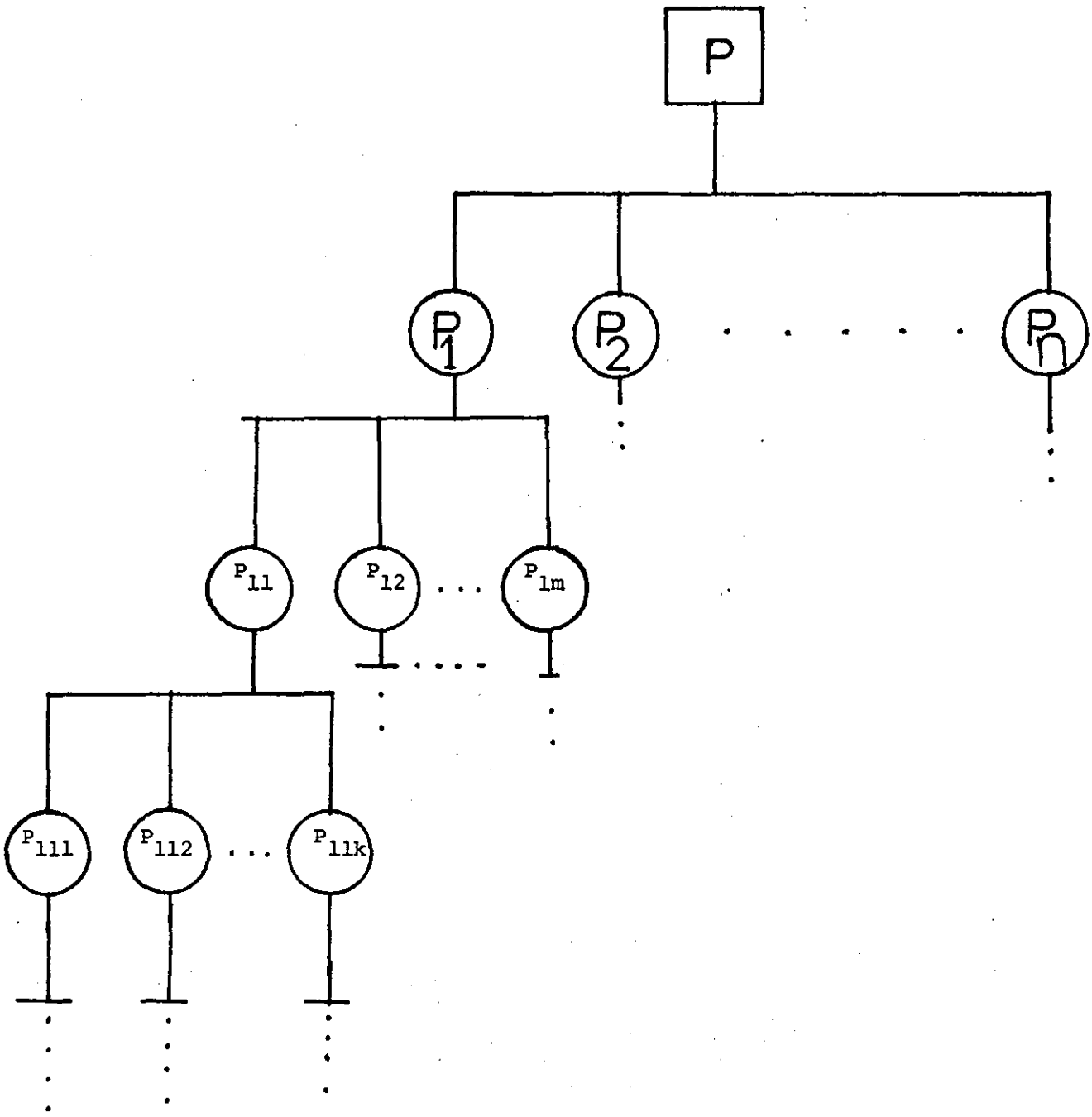


Fig. 3.2 Step-wise Refinement

operators of a programming language (like addition, subtraction, square root, etc.), by using procedures a programmer is free to define his own operators and apply them to operands that need not be basic types, (ref. 3.18) An example of a procedure, on a module, used in this way is a matrix multiplication routine, or a routine for the generation of random numbers. One of the basic advantages of procedures is that they can be used to encapsulate parts of an algorithm by localizing in one section of a program all the statements relevant to a particular aspect of a program. An example of such an encapsulation is the use of one procedure to read all input and to check for its validity. The advantage of such encapsulation, as mentioned earlier, is that one knows where to look to make changes to the encapsulated aspect of the problem. For example if one wanted to check that the inputs are nonnegative, one only needs to alter a few lines of code, and these lines are known where they are. Fig. 3.3 shows a basic format of a Pascal procedure.

In order to encapsulate autonomous program modules it is necessary for the high-level programming language to have enough capacity for local variables (ref. 3.3, 3.18). Local variables are those variables that have a value only in the module (procedure) and are otherwise unknown outside the module. Pascal allows a liberal use of local variables in both the function and procedure declarations. As mentioned earlier, the procedures (and functions) are the building blocks of modules. Procedures have the following basic characteristics:

a) They have a programmer-defined name.

b) They have zero or more dummy variables, or formal parameter arguments.

c) They may return zero or more values.

d) They do not have type.

```
PROCEDURE A (. . . formal parameter list ... ) ;
      .
      .
      ( . . local parameters declared ... )
      .
      .
      BEGIN (* procedure body *)
          .
          .
           ( * main procedure body or Block * )
          .
          .
      END  ;  ( * A * )
```

Fig. 3.3 Pascal Procedure Format

e) They have zero or more local variables

f) They are invoked simply by using, or mentioning, their name in the program. When invoked in this way control is passed to the called procedure, the called procedure is executed and then control is passed back to the calling procedure, or function. In this way control is passed back and forth as the procedures and functions are executed.

g) Procedures and functions can be nested to any arbitrary depth.

A Pascal function on the other hand is a procedure that performs a sub-task and returns a single value via the function name. Functions have similar characteristics to the procedures, but they have a type.

Pascal requires that the procedure (or function) definition to appear prior to any statement that uses the procedure (or function). In this case the function should be declared as the first executable statement in a program. Other structured languages (e.g. BASIC-PLUS, CP-6 BASIC), do not make such restrictions, so function declarations can appear anywhere in the program.

One of the major advantages of the formal parameters or dummy arguments is that they help to make the procedures and functions as general as possible: the dummy argument list can be shortened or expanded as needed. The dummy arguments are parameters that appear in the procedure or function definition and have no values of their own. When the function is invoked, the dummy arguments are replaced by the real arguments. The real arguments must correspond in position, type, and number to the dummy arguments. When the function (or procedure) is invoked the actual values of the arguments are sent to it and the function (or procedure) is

then executed. The arguments are local to the function (or procedure) and so when the function (or procedure) is executed the result of the processing should not produce side-effects - i.e. the global variables should not be changed after the procedure or function call and execution. In general side-effects violate the essential concept of modular autonomy. Any variable that appears in the list of dummy arguments is a local variable and is only known within the segment of code constituting the function or procedure block. The variables local to the function or procedure occupy different memory locations even though they are declared similarly locally in the procedure as globally in the main program. Anything done to a local variable has no effect on a variable of the same name which appears in the main program or in another procedure. Also local variables have a value only while the function or procedure is being executed. As soon as the procedure or the function block is exited, the values for all the local variables disappear. In this way strict rules are imposed that prohibit the appearance of side-effects in well-designed programs so that the procedure or module should receive all its data via the argument list and should perform its operations using no global variables. The main errors associated with modules are typically those caused by incorrect data flow between modules and by the inadvertent side-effects. Block-structured programming languages like Pascal provide the mechanism to make such a desirable rule possible. In this way too, large or complex programs can be organised as consisting of a large number of relatively autonomous modules (ref. 3.14, 3.16).

Thus the overall structure of a Pascal program is a set of procedure and function blocks, some of which are nested within others to an arbitrary

level of nesting, as shown in fig. 3.4. The rules concerning the way procedures can call each other are governed by the scope rules associated with them (ref. 3.2). Hence a program or procedure named A can call a procedure named B if:

a) B is declared in A

b) A is nested in some procedure C, and B and C are declared in the same program or procedure, provided that the definition of C follows that of B

c) A and B are both declared in C, provided that the definition of A follows that of B

With respect to the global and local variables declared in procedures A and B, the scope rules governing their use are that:

A variable V that can be referenced in a procedure A can also be referenced in a procedure B, that is nested in A, unless there is also a declaration of V in B.

In this way procedures (or modules) can be made as self-contained as possible so that each procedure and its variables can be understood without reference to its containing procedures.

3.3.2 Inter Module Organisation

A modular program is organised as a collection of modules (or procedures), as explained above. The total running time of such a modular program will therefore depend, not only on the number of modules, but also on how these modules are organised and the way they communicate and interact with one another. It will also depend on the efficiency of the modularization process and the way data flows between the collection of modules.

```
PROGRAM A (input, output, ... ) ;
   ( * comment * )
CONST ( * constants declaration * )
   .
   .
   .
TYPE ( * type declaration * )
   .
   .
   .
VAR ( * global variable declaration * )
   .
   .
   .
PROCEDURE B ;
   PROCEDURE D ;
      BEGIN (* D * )
        .
        .
        .
      END ; ( * D * )
   BEGIN ( * B * )
     .
     .
     .
   END ; ( * B * )
PROCEDURE C ;
   PROCEDURE E ;
     BEGIN ( * E * )
       .
       .
       .
     END ; ( * E * )
   PROCEDURE F ;
     BEGIN ( * F * )
       .
       .
       .
     END ; ( * F )
   BEGIN ( * C * )
      .
      .
      .
     END ( * C * )
PROCEDURE X ;
   PROCEDURE Y ;
     PROCEDURE Z ;
          BEGIN ( * Z * )
            .
            .
          END ( * Z )
      BEGIN ( * Y * )
        .
        .
      END ; ( * Y * )
   BEGIN ( * X * )
     .
     .
    END ; ( * X * )
   BEGIN ( * A * )
     .
     .
    END ( * A * )
```

Fig. 3.4 Program Block Structure

3.3.2.1 Module and Intermodule Times

When a modular program is executed, control is passed back and forth as the procedures that make up the program modules are processed. In this way the computation of the original program is equivalent to the sum of the running time of the individual modules. Hence, the computation of each module represents a fraction of the computation of the original program, i.e. the sum of the times of executing the statements comprising the module. The running time of each program module (and of the original program), will therefore depend on many factors, (ref. 3.2), some of which are:

a) The amount of input to the module

b) The nature and speed of the instructions on the computer used to execute the module

c) The quality of code generated by the compiler used to create the object program

d) The complexity of the algorithm underlying the module.

Hence, if the running time of the modules are known, then it is possible to estimate the running time of the modular program. Furthermore, if the running time of the various modules on several different computers is known, then it is possible to shorten the total running time of the program by assigning the modules to the computers on which they run faster. As has already been explained before, this is the basis of the CPU cache model of distributed computation within LAN.

In the case in which the modules run on two or more computers, it is necessary too to take into account the time involved in the modules

communicating with other modules across the interface. Such intermodule communication time is effectively additional to the running time of the individual modules. Hence, it is necessary to keep the intermodule communication time to a minimum if the benefits of running a modular program on two or more different computers are to be realised. But if the modules are separate, logically distinct and relatively autonomous, as explained earlier, the number of intermodule references can be kept to a minimum. There are two methods by which procedures (or modules) can communicate with other procedures (or the modules):

a) By means of the global variables. This assumes that global variables are implicitly declared in some universal environment. Within this environment is a subenvironment in which the modules are declared.

b) By means of the formal parameters. The formal parameters of the procedure can be treated as local variables which are initialized to the values of the actual parameters or they can serve as place holders in the program, in which case, actual parameters are substituted for every occurrence of the corresponding formal parameters. If the actual parameters is an expression, then the corresponding formal parameter is treated as a local variable initialized to the value of the expression.

The intermodule communication time will principally be the time required to transmit parameters and partial results across the interface.

3.3.2.2 The Intermodule Graph

The overall organisation of a modular program as a collection of modules can be presented using graph-theoretic concepts (ref. 3.23). Graph-theoretic concepts are often needed to represent arbitrary relationships among data and other objects. A graph (ref. 3.24, 3.25), consists

of a set of vertices (or nodes) and a set of edges (or arcs) and can be defined as follows:

A graph G = (V,E) consists of

a) a finite set V = (v_1, v_2, v_3, ... v_n) whose elements are called vertices, and

b) a subset E of the Cartesian product ExE, the elements of which are called edges.

If the edges are ordered pairs (i, j) of vertices, then the graph is said to be directed and i is called the tail and j the head of the edge (i,j). If the edges (i,j) are unordered pairs of distinct vertices, then the graph is said to be undirected. In a directed graph G = (V, E), if (i,j) is an edge in E, then vertices i and j are said to be adjacent and the edge is said to be from i to j. The number of vertices adjacent to i is called the degree of i. In an undirected graph G = (V,E), if (i,j) is an edge in E, then it is assumed that (i,j) = (j,i), so that (j,i) is one and the same edge. In this case j is adjacent to i if (i,j) is in E and the degree of a vertex is the number of vertices adjacent to it. Alternatively, the degree of a vertex may be defined as the number of edges meeting at the vertex. A graph is said to be regular if all the vertices of the graph have the same degree. In particular, if the degree of each vertex is d, then the graph is regular of degree d. A graph is said to be complete if each pair of distinct vertices are joined by exactly one edge. Furthermore, two graphs G and H are said to be isomorphic if H can be obtained from G by relabelling the vertices, such that there is a one-to-one correspondence between the vertices of G and those of H, in which case the number of edges joining any pair of vertices in G is equal to the number of edges joining the corresponding pair of vertices in H.

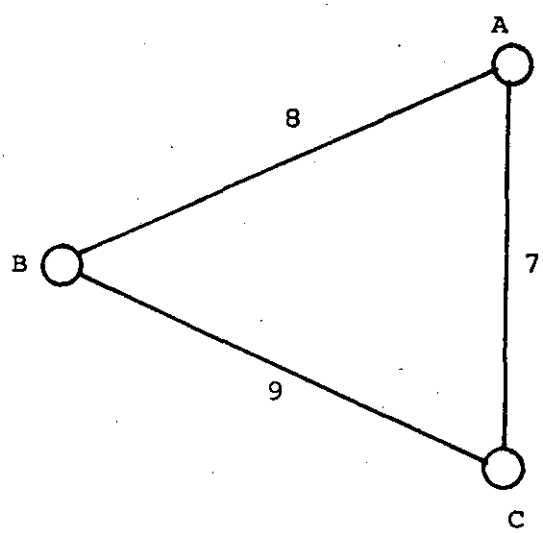
A path in a directed or undirected graph is a finite sequence of edges of the form (i_1, i_2), (i_2, i_3), ... (i_{n-1}, i_n). In this case the path is from i_1 to i_n, and is of length n-1, i.e. the number of edges it contains. In general, there may be several paths between a given pair of vertices. A path is said to be simple if all edges and all vertices on the path, except possibly the first and the last vertices, are distinct. A cycle is a simple path of length at least equal to 1 which begins and ends at the same vertex. In an undirected graph, a cycle must be of length at least equal to 3.

There are several common representations for a graph G = (V,E). An n-node graph G = (V,E) has an $||V||$ x $||V||$ adjacency matrix A, whose $(ij)^{th}$ element, A(i,j), is the weight of the edge (i,j). The adjacency matrix representation is convenient for graph algorithms which frequently require to determine the presence of certain edges since the time to determine whether an edge is present is fixed and independent of $||V||$ and $||E||$. However, the adjacency matrix representation has the disadvantage that it requires $||V||^2$ storage even if a graph has only V edges.

In this way, an intermodule graph (or a program graph) can be constructed. An intermodule graph consists of nodes (or vertices) which represent the modules, and arcs (edges) which represent the intermodule cross-references (ref. 3.23). The weights of the edges represent the intermodule communication times. Fig. 3.5 shows how an undirected, complete intermodule graph is constructed from the values of intermodule communication times, (or adjacency matrix), in the case of a modular program with three modules A, B and C. The intermodule graph of fig. 3.5 is complete and regular and hence the intermodule organisation is such that

		Modules A	Modules B	Modules C
Modules	A	0	8	7
	B	8	0	9
	C	7	9	0

(a) Intermodule Communication times (adjacency matrix)



(b) Intermodule graph (program graph)

Fig. 3.5 Intermodule Graph

any module can reference the other two modules. Hence, a general intermodule organisation is assumed by the interconnection of fig. 3.5. In many cases the intermodule organisation will not be so general and the intermodule graph is then constrained to take on a different shape, such as a tree. The final shape of an intermodule graph will be determined by the precedence relations existing among the modules of the modular program.

3.4 PARTITIONING ALGORITHMS

As explained earlier, this thesis is based on the CPU cache model of distributing a computation within a LAN (ref. 3.26, 3.27). In a CPU cache model two processors are involved in the computation of the single modular program. These processors have so far been referred to as the larger and more powerful processor, and the smaller processor respectively. The terms sink and source processors will be used to refer to larger and the smaller processors respectively, as these terms are also consistent with the graph-theoretic concepts. In this context, some graph based scheduling algorithms can be used to partition the modular program into two so that one portion of the program is assigned to the source processor while the other portion is assigned to the sink processor, whenever possible.

Once the modules have been identified and the module running time and intermodule communication times specified, it is now necessary to run the scheduler. The scheduler works out the module assignment to the processors on the basis of the shortest processing time for the program. A basic assumption in this is that the time to run the scheduler is small compared to the total processing time of the program. Three main graph-theoretic scheduling algorithms are now examined.

3.4.1 The Max-Flow Min-Cut Scheduler Algorithm

The max-flow min-cut algorithm is based on the correspondence of the concept of the amount of a maximum flow (ref. 3.28) and the value of the minimum cut in a network. A network is a useful general concept which can be used in solving a wide range of practical problems. In this connection, a network can be thought of as a graph which carries some additional information (ref. 3.29). Essentially, a network is a graph in which each edge or arc is assigned a number, called its weight. Each arc represents a possible channel for some kind of flow, and the weight gives some information about the flow along it. This may be the capacity, i.e. the maximum flow possible in that channel, it may be a length of the channel, the cost of sending a commodity along it, or it may be some other quantity. The relevant flow commodity in this case is the expected (or the worst-case) processing time. Examples of networks which can be modelled in this way are Railway networks, Road networks, Road maps, Fluid distribution systems, Commercial networks, Electrical networks, Communications networks, Computer networks, Software networks, etc. The network problems which can be thus modelled (ref. 3.29) include

a) Maximum flow problems

b) Flow and potential problems

c) Transportation problems

d) Assignment problems

e) Scheduling problems

f) Location problems

In the maximum flow network problems, the capacity of each arc represents the maximum flow possible along that arc. If the network is

a directed graph, then the direction of the arc represents the direction of the flow. The characteristics of flow in a network can be explained by the concept of a basic network of fig. 3.6. A basic network is a directed graph which satisfies the following conditions:

a) It has exactly one source (S) node and one sink (T) node.

b) To each arc (i,j) of the basic network there is assigned a positive number c(i,j) called the capacity of the arc (i,j).

Hence a flow in the basic network with source S and sink T is an assignment of non-negative number f(i,j), called the flow along arc (i,j), to each arc (i,j) of the basic network, and satisfying the following feasibility and flow conservation conditions, (ref. 3.28):

a) The flow along arc (i,j) does not exceed the capacity of the arc (i,j), i.e.

$$f(i,j) \leq c(i,j)$$

for each arc (i,j) of the basic network:

(the feasibility condition)

b) For each vertex V, other than the S and T, the sum of the flows along the arcs into V is equal to the sum of the flow along the arcs out of V : (the flow conservation condition).

Furthermore,

a) The source has no inward arcs

b) The sink has no outward arcs

c) The outflow at the source must equal the inflow at the sink.

Hence, the major objective in the maximum flow problems is to maximise the total flow F in the network in which each arc (i,j) has capacity $c(i,j)$. This, for a basic network with n vertices, including

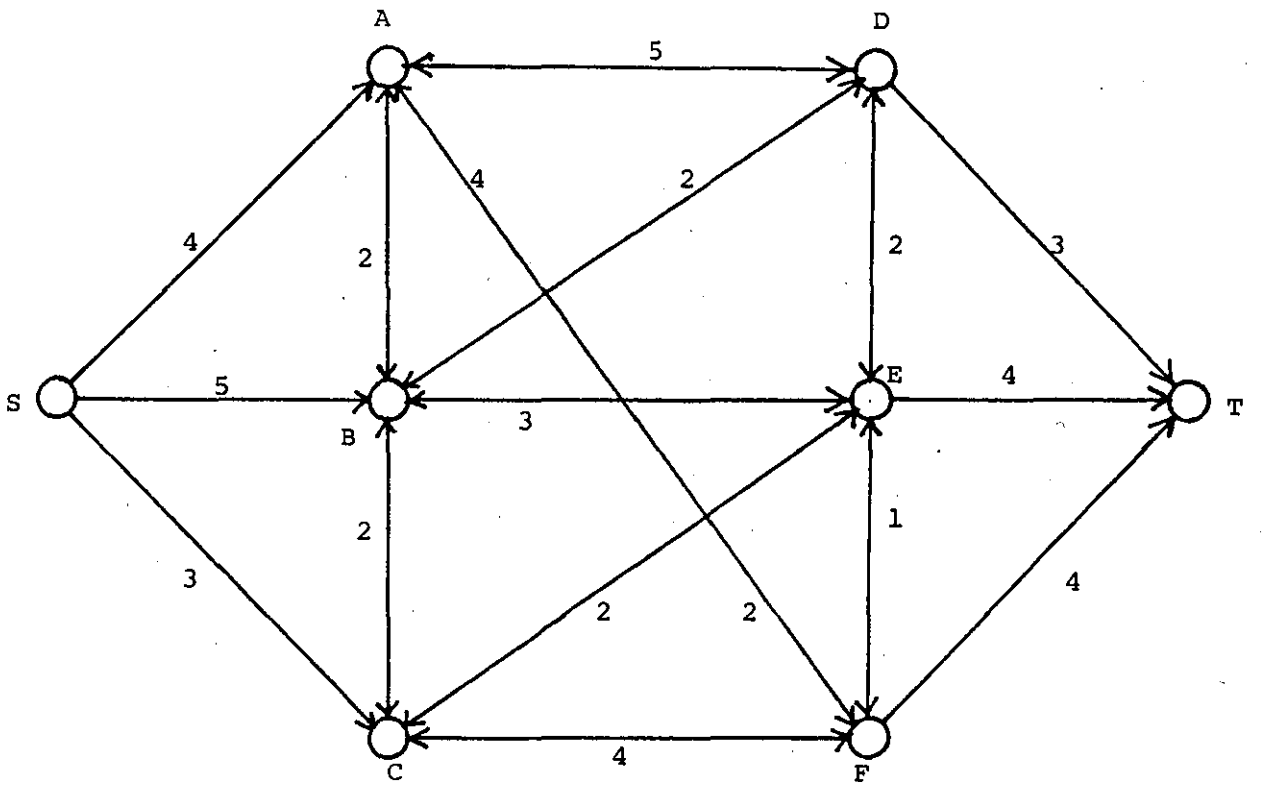


Fig. 3.6 A flow network

the source node (vertex 1) and sink node (vertex n), can be expressed as

$$\text{Maximise } F = \sum_{j=1}^{n} f(i,j)$$

subject to

$$\sum_{j=1}^{n} f(i,j) - \sum_{j=1}^{n} f(j,i) = \begin{cases} F \text{ if } i = 1 \\ 0 \text{ if } i \neq 1 \text{ or } n \\ -F \text{ if } i = n \end{cases}$$

and

$$0 \leq f(i,j) \leq c(i,j)$$

In a basic network an arc (i,j) is said to be saturated if $f(i,j)=c(i,j)$, and unsaturated if $f(i,j) < c(i,j)$. In the maximum flow problems the objective is to find the largest possible (S,T) flow that the network can support. This is accomplished by finding the flow-augmenting (S,T) paths in the basic network. Such paths consist of

a) forward arcs - i.e. unsaturated arcs directed along the path.

b) backward arcs - i.e. arcs directed against the direction of the path and carrying a non-zero flow.

In order to obtain a maximum (S,T) flow, a succession of such flow-augmenting paths have to be found and then the flow along them increased step by step until the flow can be increased no further. When there are no more flow-augmenting paths left, the flow thus produced is the maximum flow. An algorithm that is based on the idea of systematically locating flow-augmenting paths along which the flow can be increased can be applied to any basic network to find the maximum (S,T) flow.

The alternative method for determining whether or not a given flow is a maximum flow is based on the concept of a cut, or a network bottleneck.

A cut in a network with source S and sink T is a set of arcs whose removal separates the basic network into two graph components X and Y, one containing the source S and the other containing the sink T, as illustrated in fig. 3.7. The capacity of such a cut is equal to the sum of the capacities of those arcs in the cut which are directed from X to Y. The cut with the smallest possible capacity is called the minimum cut. The bottle-neck may consist of a few arcs of small capacity through which the relevant commodity has to flow. The connection between the minimum cut and the maximum amount of flow in a network has been an important established result (ref. 3.28), and forms the basis of the Max-flow min-cut theorem, which states that:-

In any basic network, the value of the maximum flow is equal to the capacity of the minimum cut.

The max-flow min-cut algorithm, (ref. 3.30), based on the above theorem, can be used to determine the maximum feasible flow between a given source S and sink T. The algorithm takes a weighted, directed graph of input and determines the maximum (S,T) flow by building a layered network from the source to the sink. The source node is put in layer zero. Any node intermediate between the S and T nodes and connected directly with the S node by a flow-augmentable arc is put in layer 1. Any node connected to a layer 1 node by a flow-augmentable arc is in turn put in layer 2, etc. This process continues until the sink node has been reached, and assigned a layer, when the process terminates. In this way, the algorithm effectively labels each node by its distance from the source node. For each layered (S,T) path each node is inspected in turn to find the potential flow increase it can handle. The intermediate layered network node with the smallest potential (the reference node) and its reference potential is determined so that an amount of flow equal to the

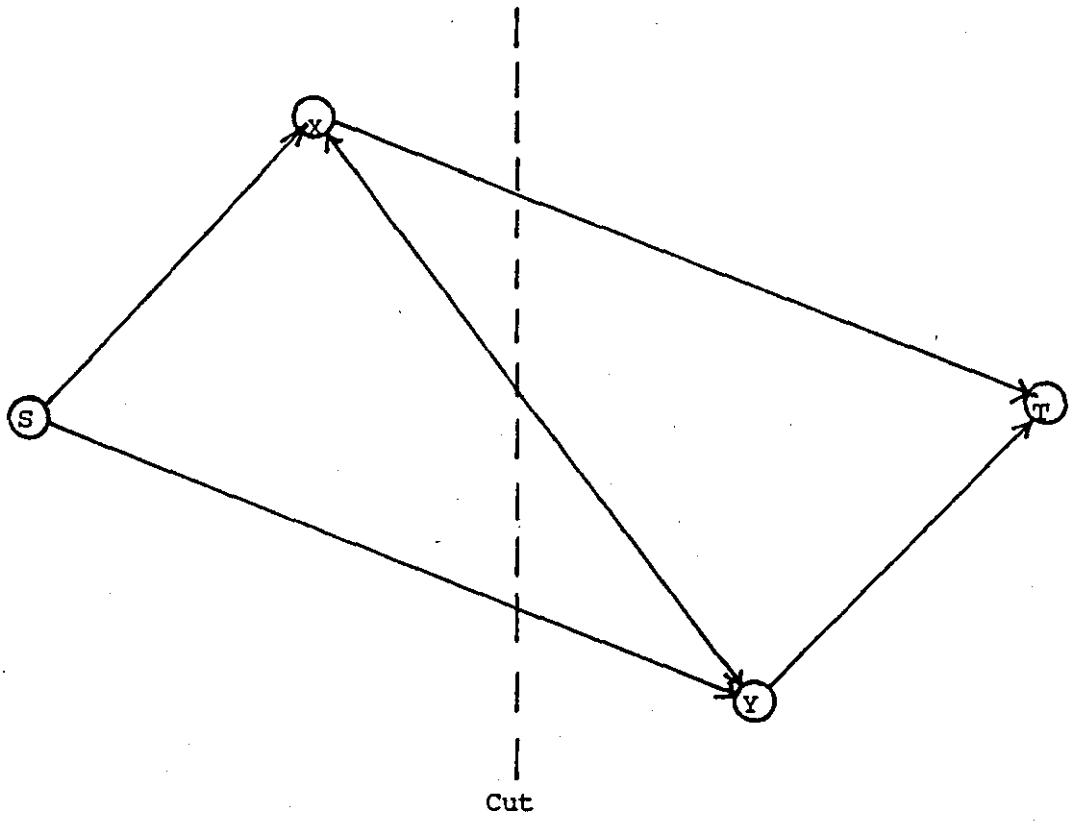


Fig. 3.7 The Cut

reference potential is pushed to the direction of the sink, or pulled from the direction of the source. This process of building the layered network and(S,T) paths, finding the reference node, and augmenting the flow continues until it is not possible to build a layered network. When this situation obtains, then the current flow is the maximum flow, and the set of saturated arcs forms a cutset.

Fig. 3.8 incorporates a Pascal procedure (ref. 3.31, 3.34), to carry out the max-flow min-cut algorithm. The body of the procedure operates, as described above, by first trying to build a layered (S,T) path (Function Layering-Possible, and Procedure Walk), then finding the reference node (Procedure Find-Ref-Node), and finally by augmenting the flow in both directions from the reference node (Procedure Push-Pull).

As explained above, the max-flow min-cut algorithm takes in a basic network consisting of a weighted, directed graph, as input and determines the maximum feasible flow between the source vertex S and the sink vertex T. The basic network is constructed from the intermodule graph, as follows (ref. 3.23, 3.26). Two new nodes S and T, representing the source and sink, are added to the intermodule graph. An arc is drawn from the source node S to each intermediate node (module) and is labelled with a weight C(S,i) equal to the running time of that module at the sink processor. Similarly an arc is drawn from T to each intermediate node (module) and labelled with a weight C(i,T) equal to the running time of that module at the source processor. If a module cannot run on either the source or the sink processor the corresponding weights of the arcs joining that module to the source or the sink processor is equal to infinity (OO). All that is left now is to input the resulting basic network to the max-flow min-cut scheduler algorithm to determine the assignment of modules to the two processors together with the expected processing time for that program.

Fig. 3.8 Flow-Augmentation in Max-Flow Min-Cut Algorithm

```
 1                    program distcomp (output) ;
 2   const
 3    n=5 ;
 4    unscanned=-5 ;
 5    infinity=10000 ;
 6
 7   type
 8       node=1..n ;
 9       xnode=-n..n ;
10       vector=array[node] of xnode ;
11       matrix=array[node,node] of real ;
12       whichway=(push,pull) ;
13
14   var
15       i,s,t : node ;
16       j : node ;
17       c,f : matrix ;
18       x,y,flowleft :real ;
19       p: whichway ;
20       minimumcut  :  real  ;
21
22   procedure maxflow (s,t:node ;c:matrix ; var f:matrix) ;
23    var  refnode :node ;  (*node with least excess capacity  *)
24     minpotential :real ;  (*excess capacity of the ref node *)
25     layer :vector ;  (*the layered network is defined by this array *)
26     r : real ;
27     i,j :node ;  (*indices *)
28
29    function  min (x,y:real):real ;
30     (*determines the minimum amount of flow *)
31     begin
32      if x<y
33      then min :=x
34       else min :=y  ;
35     end ;
36
37    procedure walk (i:node ) ;
38     (*traverse the layered network from t,inverting layer numbers.*)
39     var  j:node ;  li :xnode ;
40      begin
41       layer[i] := -layer[i] ;
42       li :=layer[i] ;
43       for j:= 1 to n do
44        if (j<>s) and (-layer[j]=li-1) and ((f[j,i]<c[j,i]) or (f[i,j]>0
\c))
45         then walk (j)  ;
46      end ;  (*walk *)
47
48    function layeringpossible : boolean ;
49      (*is it possible to build a layered network, if so build it *)
50     var  i,j :node ;
51      k:0..n ;
52      emptylayer :boolean ;
53      begin
54       k :=0  ;    (*k keeps   track of layer being built *)
55       for i:= 1 to n do
56        layer[i] :=unscanned ;  (*initialize each node *)
57        layer[s] :=k ;  (*source node is in  layer 0 *)
58        writeln ('*',layer[s]:30) ;
59       repeat
60        k :=k+1 ;   (*now locate all nodes in layer k *)
```

```
      emptylayer := true ;  (*an empty layer stops the algorithm *)
      for i:= 1 to n do
       if -layer[i] = k-1
        then
         (*i is in layer k-1,its neighbors may be in layer k  *)
         for j := 1 to n  do  (*check each node adjacent to i *)
          if (layer[j]=unscanned) and ((f[i,j]<c[i,j]) or (f[j,i]>0))
           then
            begin
             layer[j] := -k ;
             writeln ;
             writeln ('*',-layer[j]:30) ;
             writeln ;
             emptylayer := false
            end ;
     until  (layer[t] <> unscanned ) or emptylayer ;
     layeringpossible := not  emptylayer ;
     walk (t) ;   (*prune off the dead ends  *)
     writeln ('layering is possible    ',not emptylayer) ;
    end ;  (*layeringpossible *)

  procedure findrefnode  (i:node)  ;
   (*traverse the layered network from t, seeking the ref node *)
   var  j :node ;
    li,lj :xnode ;
    incap, outcap : real ;
    begin
     li  := layer[i] ;
     incap :=0  ;
     outcap := 0 ;
     for j :=1 to n do
      (*examine each node adjacent to i *)
      begin
       lj :=layer[j] ;
       if (lj =li-1) and (j<>s) and ((f[j,i]<c[j,i]) or (f[i,j]>0))
        then  findrefnode (j)  ;
       if  lj = li-1
        then incap :=incap + (c[j,i]-f[j,i])+f[i,j]  ;
       if lj =li+1
        then outcap :=outcap + (c[i,j]-f[i,j])+f[j,i]
      end ;
      if  (i<>s) and (i<>t) and  (min (incap,outcap)<minpotential)
      then
       (*node i has smaller potential than the current ref node *)
       begin
        minpotential := min (incap, outcap)  ;
        refnode := i  ;
       end ;
      writeln ('the reference potential =', minpotential:6:2 ) ;
      writeln ('the reference node =', refnode:3 ) ;
    end  ;  (*findrefnode *)

  procedure  pushpull (i :node ; flowleft :real ; p : whichway ) ;
   (*augment the flow thro' i by pushing or pulling minpotential units
\c *)
    var  j, k1, k2, layersought : 0..n ;
     begin
     j := 0 ;
      while (flowleft >0) and (j<n) do
       begin
        j :=j+1 ;
```

```
121          if p=push
122           then
123            begin
124             k1:=i ;
125             k2:=j ;
126             layersought:=layer[i]+1
127            end
128           else
129            begin
130             k1:=j ;
131            k2:=i ;
132             layersought :=layer[i]-1
133            end ;
134           r:=min (flowleft,c[k1,k2]-f[k1,k2]+f[k2,k1] )  ;
135            (*amount of flow to move *)
136           if  (r>0) and (layer[j]=layersought)
137            then
138             begin  (*push/pull some flow to/from an adjacent  layer *)
139              flowleft :=flowleft -r  ;
140              f[k1,k2] :=f[k1,k2]+r-min(r,f[k2,k1])  ;
141               (*augment positive flow *)
142              f[k2,k1] := f[k2,k1]-min (r,f[k2,k1])  ;
143               (*push reverse flow backwards *)
144              if (j<>s) and (j<>t)
145               then  pushpull (j,r,p)
146             end
147         end ;
148         writeln ('forward flow =',f[k1,k2]:6:2) ;
149         writeln ('reverse flow =',f[k2,k1]:6:2) ;
150         writeln ('flowleft =',flowleft:6:2) ;
151         writeln ('r=',r:6:2) ;
152         writeln ;
153         writeln ;
154       end  ;(*pushpull *)
155
156      begin   (*maxflow *)
157       for i :=1 to n do
158        for j :=1 to n do
159         f[i,j] :=0  ;  (*initialy no flow *)
160         f[s,t] :=c[s,t]  ;  (*if an s_t link exists , saturate it *)
161         minimumcut := 0 ;
162        while layeringpossible do   (*assign nodes to layers  *)
163         begin
164          minpotential := infinity  ;
165          findrefnode (t)  ;   (*find the reference node *)
166          pushpull (refnode,minpotential,push) ;  (*push flow towards the
\csink*)
167          pushpull (refnode,minpotential,pull) ; (*pull flow from source*)
168           minimumcut := minimumcut + r ;
169         end  ;
170         writeln ('minimumcut =', minimumcut:6:2) ;
171      end  ;  (*maxflow*)
172
173    begin    (*main program *)
174      s :=1 ;
175      t :=5 ;
176      c[1,1]:=0 ;
177      c[1,2]:=6 ;
178      c[1,3]:=50 ;
179      c[1,4]:=6 ;
180      c[1,5]:=0 ;
```

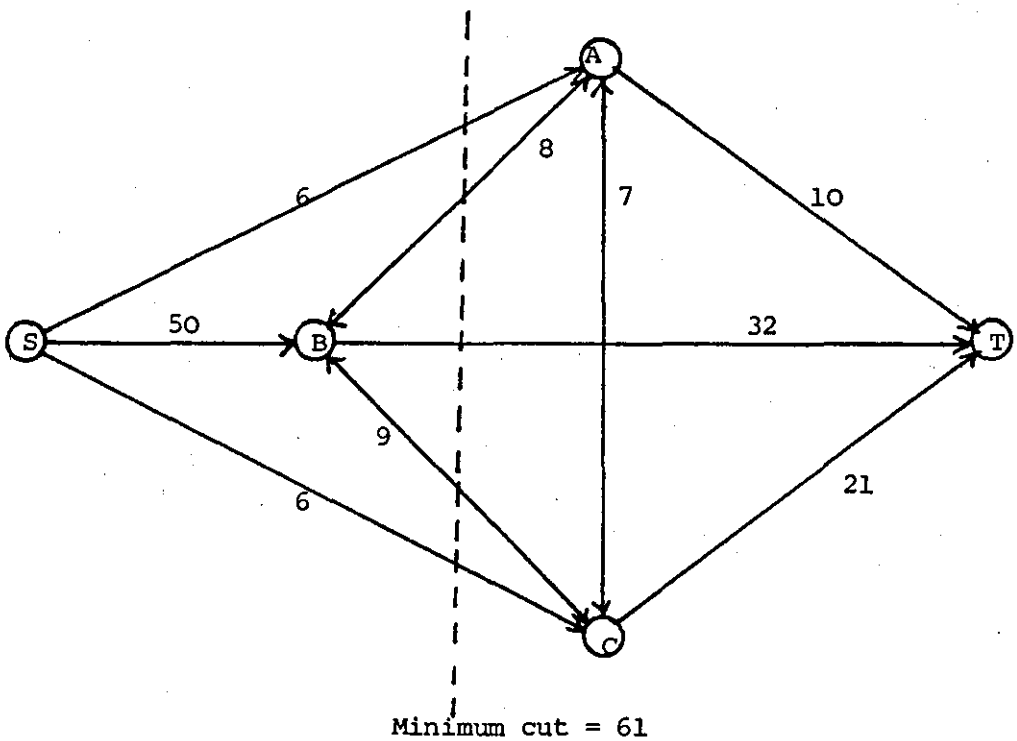
```
181     c[2,1]:=0 ;
182     c[2,2]:=0 ;
183     c[2,3]:=8 ;
184     c[2,4]:=7 ;
185     c[2,5]:=10 ;
186     c[3,1]:=0 ;
187     c[3,2]:=8 ;
188     c[3,3]:=0 ;
189     c[3,4]:=9 ;
190     c[3,5]:=32 ;
191     c[4,1]:=0 ;
192     c[4,2]:=7 ;
193     c[4,3]:=9 ;
194     c[4,4]:=0 ;
195     c[4,5]:=21 ;
196     c[5,1]:=0 ;
197     c[5,2]:=0 ;
198     c[5,3]:=0 ;
199     c[5,4]:=0 ;
200     c[5,5]:=0 ;
201     writeln ('************************************')  ;
202     writeln ('maximum flow _ minimum cut algorithm ') ;
203     writeln ('************************************')  ;
204     writeln ;
205     maxflow (s,t,c,f)  ;
206  end.
```

For the previous example of a program with three modules A, B and C, fig. 3.9 shows the basic network, the assignment of modules to each of the two processors and the expected processing time (equal to the value of the minimum cut) of the program. Hence, the max-flow min-cut algorithm determines both the minimum running time and the module assignment to processors in a dual processor distributed computation environment.

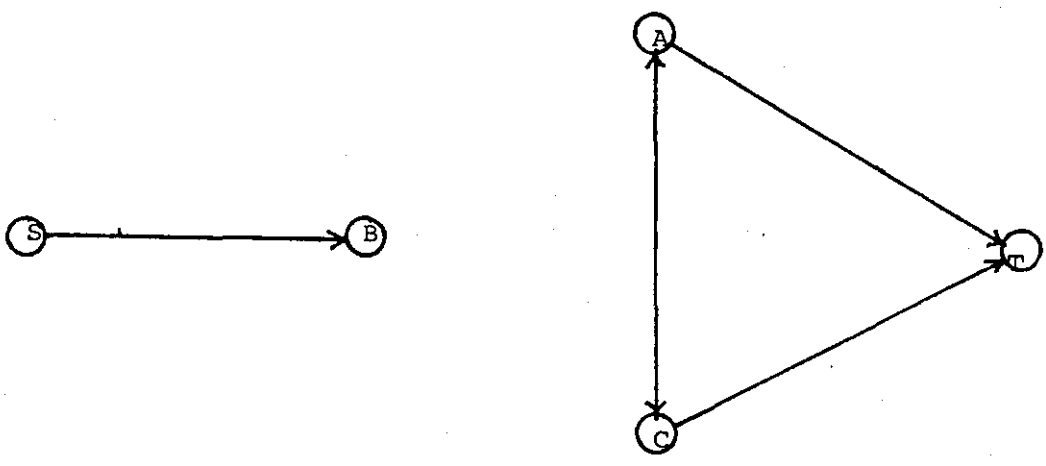
As can be seen from fig. 3.9, the value of the minimum cut is equal to 61 time units (milliseconds, seconds, minutes, hours, etc.). Hence, the running time of the program whose modules are A, B and C in such a dual processor distributed computation system is 61 time units. This time represents the shortest possible time to run this program in such a set up, and this is accomplished by having module B assigned to run at the source processor and modules A and C assigned to run at the sink processor. Had the program run at the source processor alone, the time taken to process it would be 63 time units, while if it had run on the sink processor alone it would have taken 62 time units. Hence, a time of 2 time units are saved by running the program in such a dual processor system. Similarly, a time equal to 1 time unit is saved by deciding not to run the entire program at the sink processor. This saving in the total processing time of the program assumes that the time overhead to run the max-flow min-cut scheduler is negligible compared to the total processing time. Also, another major assumption in this calculation is that the sink processor is immediately available to run the portion of the computation scheduled to it. This will not be always the case. The sink processor will most probably have a certain amount of workload to process because it is time-shared, multiaccessed, and multiprogrammed and hence the scheduled modules will be run in that time-shared and multiprogrammed invironment. Under these circumstances it is necessary

	T	S
A	6	10
B	50	32
C	6	21

(a) Module Running Time



(b) Basic Network



(X) (Y)

(c) Graph components X and Y

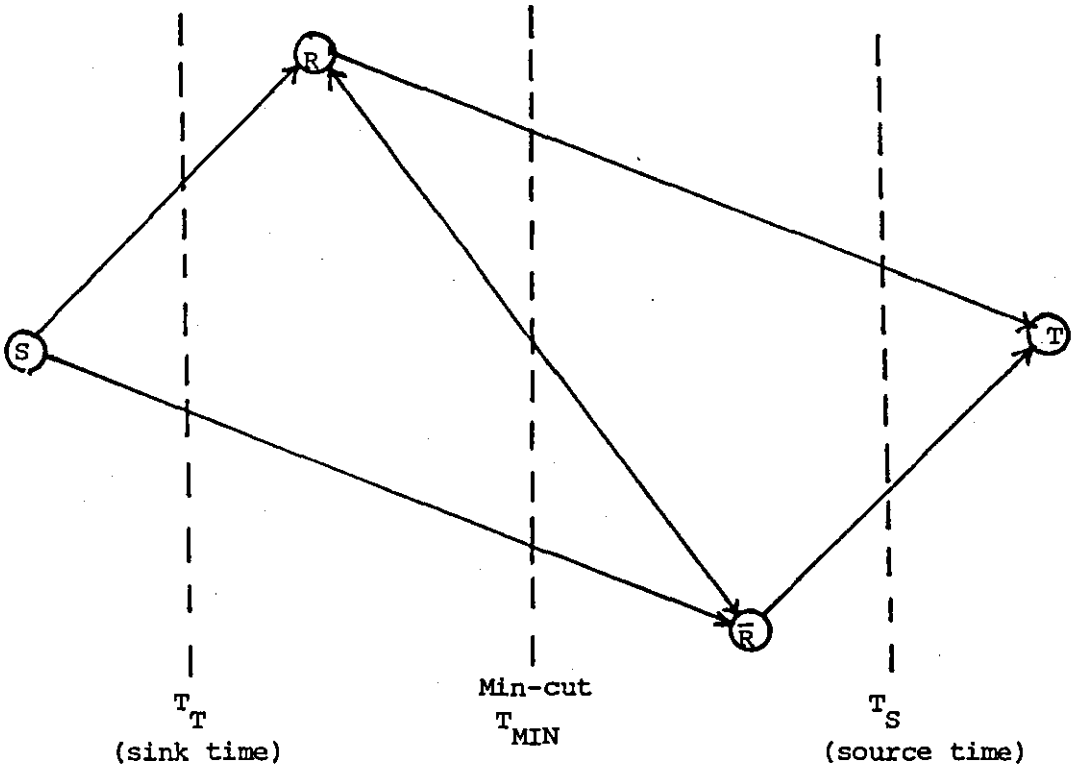
Fig. 3.9 Two graph components

that the sink workload is not excessive if the benefits of distributing a computation thus are to be realised. Furthermore, in such a dual processor distributed computation system, the two modules A and C, which are assigned to the sink processor, have to be assembled into packets, queued for access to the transmission channel, and then transmitted via the LAN communications subnet to the sink processor. Hence, the total channel time delay should also be negligible too in comparison with the value of the minimum cut. These and other factors will be examined later.

It can be seen that according to the max-flow min-cut scheduling algorithm, the prevailing simplified situation is as illustrated in fig. 3.10. In fig. 3.10, R represents the set of all the modules assigned to process at the source processor (the resident modules) and $\overline{R}$ represents the set of all the modules assigned to run at the sink processor (the non-resident modules). All the $\overline{R}$ modules have to be assembled into packets and transmitted across the LAN communications subnet, to the sink processor. These module packets are effectively queued up at each individual user where they wait for channel access and eventual transmission to the sink processor. In this way they must experience channel acquisition and transmission delays across the interface. As mentioned above, it is necessary that the total delay experienced by the module assigned the sink processor, the time interval from when the scheduler is run to the time they are available for processing there, is as small as possible and in any case very small compared to the value of the minimum cut, T_{MIN}. With reference to fig. 3.10, it is necessary in a distributable computation that:

$$T_{MIN} << T_S \text{ and } T_{MIN} << T_T$$

where T_S and T_T are the run times (real time), of the program at the source and sink processors respectively. Also, there are only three



$$T_{MIN} < T_S$$

$$T_{MIN} < T_T$$

Fig. 3.10 The 3 cut combinations

possible cases for the minimum cut. These three cases correspond to:

a) $T_{MIN} = T_S$, in which case $\bar{R}$ is a null set and all modules are resident and process at the source processor alone.

b) $T_{MIN} = T_T$, in which case R is a null set and all modules are non-resident and process at the sink processor alone.

c) $T_{MIN} < T_S$, and $T_{MIN} < T_T$, in which case R and $\bar{R}$ are non-null module sets and the computation is distributable.

It is also possible that $T_{MIN} = T_S = T_T$. In this case no benefit can be obtained in assigning any modules to the sink processor. In cases a) and b) above, all the modules are processed at one local site. However in case c), a distributed computation environment exists in which the set R modules run at the source processor and the set $\bar{R}$ modules run at the sink processor. In this mode of distributed computation the modules are processed strictly sequentially and concurrent or parallel processing is not assumed. Each module time is equal to the time to process that module to its completion. Also, as the modules are processed at the two sites, parameters and data are passed back and forth across the interface. Each such intermodule communication involves organising the parameters and data into packets for channel transmission. Hence, the channel acquisition and transmission delay accounts for the longest part of the intermodule communication time. It is therefore important to keep the intermodule communication time as low as possible. This can be achieved both by making the modules as autonomous as possible to reduce the number of intermodule references, and by reducing the channel acquisition and transmission delay. From fig. 3.10, it can be seen too, that the set of modules assigned to the same processor incur zero intermodule communication

time. It can be seen that the main factors that contribute to the time cost of each assignment are:

a) The amount of computation required by each module

b) The amount of data transmitted between each pair of modules

c) The speed of each processor

d) The speed of the communication channel separating each pair of processors

The main objective in the dual processor distributed computation environment is to obtain an assignment that minimises the sums of the module execution times, RUN, and the intermodule communication times, COMM, i.e.

$$\text{minimise } \sum_{i} \text{RUN}(i) + \sum_{i} \sum_{j \neq i} \text{COMM}(i,j)$$

In the case of a distributable computation which is represented by modules in a fully connected, complete intermodule graph, the value of the minimum cut, T_{MIN}, can be expressed by

$$T_{MIN} = n_S.\overline{T}_S + n_T.\overline{T}_T + n_1.n_2.n_T.\overline{T}_1 + n_S.n_T.\overline{T}_2 + k \leq n.\overline{T}_S$$

where

$n.\overline{T}_S = T_S$ = time to process all modules at the source

$n_S + n_T = n$ = total number of modules comprising the program

n_S = number of modules assigned to source processor

n_T = number of modules assigned to sink processor

n_1 = average number of packets per module transmitted to the sink processor

n_2 = average number of times parameters and data are transmitted across the interface

$\overline{T}_S$ = average processing time per module at the source processor

$\overline{T}_T$ = average processing time per module at the sink processor

$\overline{T}_1$ = average intermodule communication time per module pair

$\overline{T}_2$ = average time to transmit intermodule parameters

k = a constant representing such factors as

a) time to run the scheduler

b) time to assemble/dissassemble the module packets

c) time to perform error detection.

Hence, for a given program with n modules, the value of the minimum cut can be reduced by reducing n_1, n_2, $\overline{T}_1$, $\overline{T}_2$, and k. The value of k can be reduced by such factors as

a) Using a more efficient scheduler

b) Using long packets

c) Reducing the time to perform error detection

d) Using more autonomous modules

The value of n_1 can be reduced by using long packets while the values of n_2 and $\overline{T}_1$ can also be reduced by designing autonomous and more independent modules to minimise the number of the intermodule references. The value of $\overline{T}_2$ can be reduced by minimising the average channel and transmission delay. In a high speed LAN communications subnet, the average channel and transmission delay under low traffic conditions can be expected to be low. It can also be expected that not all the modules will need data and parameters from every other module during the module processing time. Hence, in the majority of situations, the program graph will not be a fully connected graph. This too can result due to strong module autonomy and the prevailing precedence relationships among modules. The net effect of this is to reduce the number of intermodule

references and hence the intermodule communication times. Such strong module autonomy will further have the result that the time interval between the intermodule references will increase.

The time to run the max-flow min-cut scheduler will also depend on the number of the augmented (S,T) flow paths in the basic network. The number of these augmented (S_1T) flow paths depend very strongly on the weights capacities of the arcs in the basic network. It is the weight capacities of these arcs which determine the arcs in the (S,T) paths that saturate first. Also, the relative weights of the C(S,i) and C(i,T) are important in determining how many (S,T) flow augmentations a given basic network can support. If a program has n modules, then the number of nodes in the basic network is (n+2). The number of nodes in the program graph is also equal to n. Hence, in a fully connected intermodule graph, the maximum number of the feasible (S,T) flow augmentation paths scanned during a scheduler run is $\frac{1}{2}.n.(n+1)$ paths. In the basic network of fig. 3.9, the number of these paths is 6, as shown in fig. 3.11. But, as mentioned above, not all of these paths may be scanned because of the fact that the intermodule graph is not fully connected and also because of the relative and absolute values of the arc weights. On the other hand, the minimum number of the (S,T) flow augmentations paths is n. Hence, in an arbitrary intermodule graph with arbitrary arc weights, the actual number of these paths will lie between n and $\frac{n}{2}.(n+1)$. These $\frac{n}{2}.(n+1)$ paths are also chosen from a very large number of possible combinations of such (S,T) paths. By choosing the values of the arc weights, it is possible to investigate how the value of the minimum scheduling time, corresponding to the time taken to scan just the minimal n paths, varies with the number of modules. These minimal

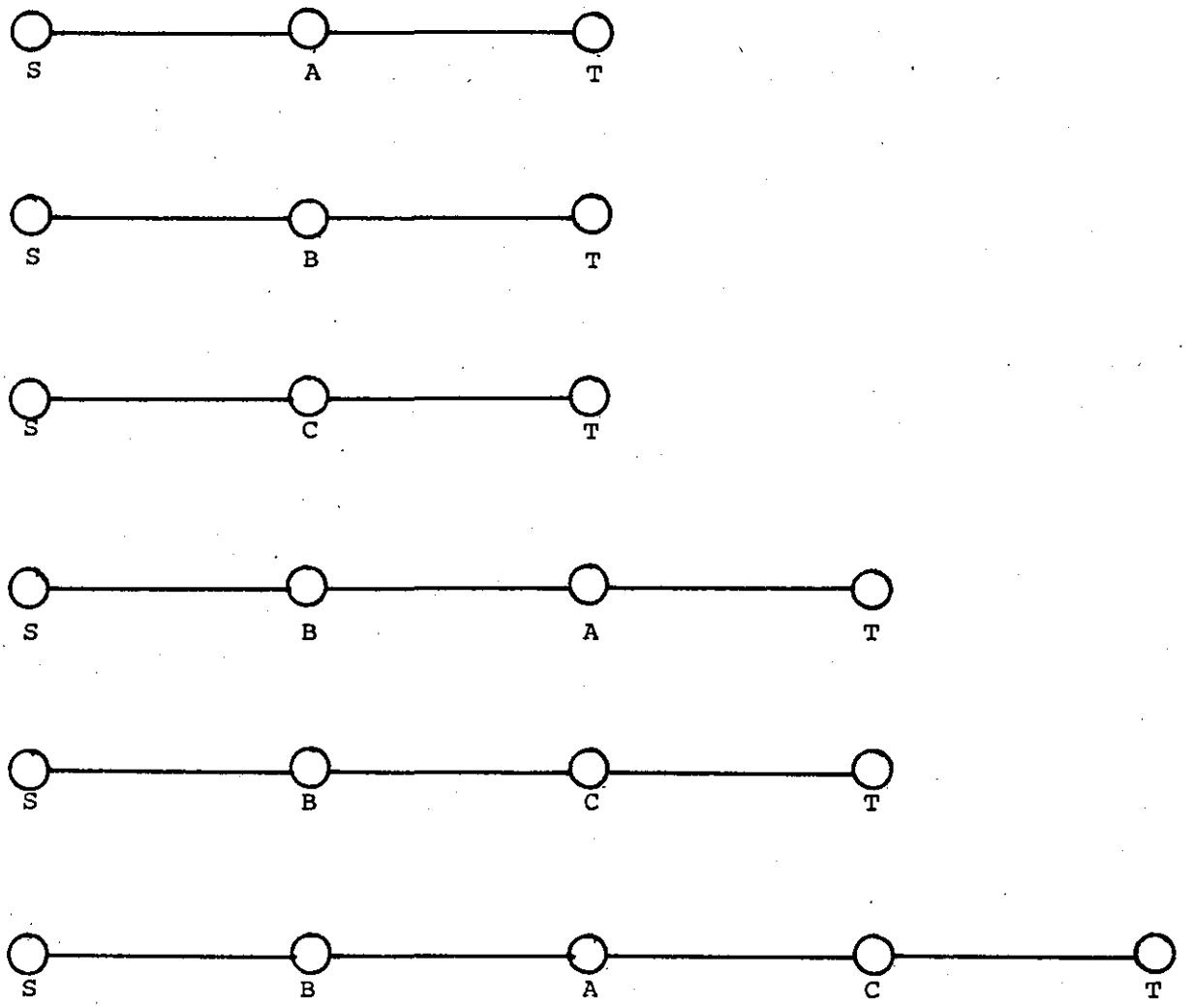


Fig. 3.11 6 (S,T) - flow augmentation paths

n (S,T) paths correspond to an assignment of all the n modules to either one of the two processors and none to the other.

3.4.2 The Enumerative Scheduler Algorithm

The enumerative scheduler is also a graph-theoretic algorithm. Like the max-flow min-cut scheduler it is based on the concept of the maximum amount of flow of a commodity and the corresponding value of the minimum cut. It takes in the basic network consisting of a weighted, directed graph, as input and determines both the program processing time and the module assignment to the processors, as explained earlier. It accomplishes this by exhaustively enumerating all the possible (S,T) cuts in the basic network and then searches and chooses the minimum cut in the (S,T) cutset. By so doing it partitions into two the total number of modules in the modular program so that some of them are assigned to the source processor and the others are assigned to the sink processor. In this way, it exhaustively takes into account the total number of combinations of the partitions (ref. 3.32, 3.33). Hence, for a basic network with n modules, the total number of partitions of the cuts is 2^n, i.e. the number of unordered selections

$$\begin{bmatrix} n \\ 0 \end{bmatrix} + \begin{bmatrix} n \\ 1 \end{bmatrix} + \begin{bmatrix} n \\ 2 \end{bmatrix} + \cdots\cdots \begin{bmatrix} n \\ h \end{bmatrix} = 2^n$$

For large n, it can be seen that the total number of partitions is very large. Hence, the algorithm exhibits the phenomenon of combinatorial explosion and is therefore very inefficient for large values of n. In this way, it can be expected that the module scheduling time increases almost exponentially with the number of program modules. But, for small values of n, the algorithm may be quite efficient and even performs faster

than the more efficient max-flow min-cut algorithm. Also, this algorithm is not capable of taking advantage of a simplified nature of the interconnections within the intermodule graph and the scheduling for modules in the case of a fully connected intermodule graph would be only marginally different from that of an intermodule graph with much fewer number of edges.

3.4.3 The Shortest Tree Scheduler Algorithm

Bokhari (ref. 3.35) has analysed the problem of optimally assigning the modules of a modular program over the processors of an inhomogeneous distributed processor system using a shortest tree algorithm. This algorithm too is graph-theoretic in nature. As before, the objective is to assign the modules, whenever possible, to the processors on which the modules execute most rapidly while taking into account the overhead of interprocessor communication. The shortest tree algorithm aims to minimise the sum of the execution and intermodule communication times for arbitrarily connected distributed systems with an arbitrary number of processors, provided that the interconnection pattern of the modules forms a tree. A tree (ref. 3.36), is a connected graph which contains no cycles. Programs that have a tree-like structure form an important class and include programs written as a hierarchy of subroutines. This tree-like structure is found to be quite suitable for large modular programs (ref. 3.39). Also, the structured programming high-level languages such as Pascal and PL/1 tend to take the advantage of tree arrangements with their nested block structures.

As explained earlier, the modules will transfer control to each other at various times during the lifetime of the program in execution. By drawing up a directed graph in which each mode represents a module and in which there is an edge from node i to node j if and only if module i

calls module j during the program execution, the resulting intermodule graph is called a calls graph. As mentioned above, the shortest tree algorithm for optimal assignments assumes that such a calls graph of the modular program is a directed tree. Such a directed tree is also invariably called an invocation tree because it describes the way modules invoke other modules during the execution of the program. An invocation tree made up of four modules is shown in fig. 3.12(a). Should a module invoke another module that is not coresident with it on the same processor, this invocation would have to be transmitted across the LAN communications subnet and thus incur an interprocessor communication time cost. This time cost is dependent on the amount of data transmitted from one module to the other. The cost of invoking a coresident module is zero, as explained earlier.

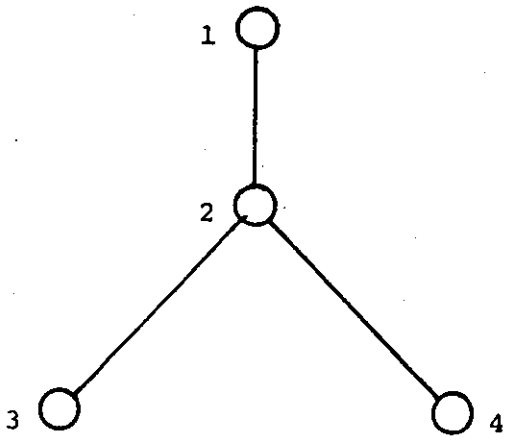
The cost of executing module i on processor j is denoted by c(i,j) and equals the sum of the costs of the various periods of execution of the module throughout the lifetime of the program. The minimum processing time assignment over the distributed processor system minimises the sums of the execution times and the intermodule communication times. Given the invocation tree of a modular program, and the execution and intermodule communication times, an assignment graph may be drawn up, as shown in fig. 3.12(b). An assignment graph has the following characteristics:

a) It is a directed graph with weighted edges

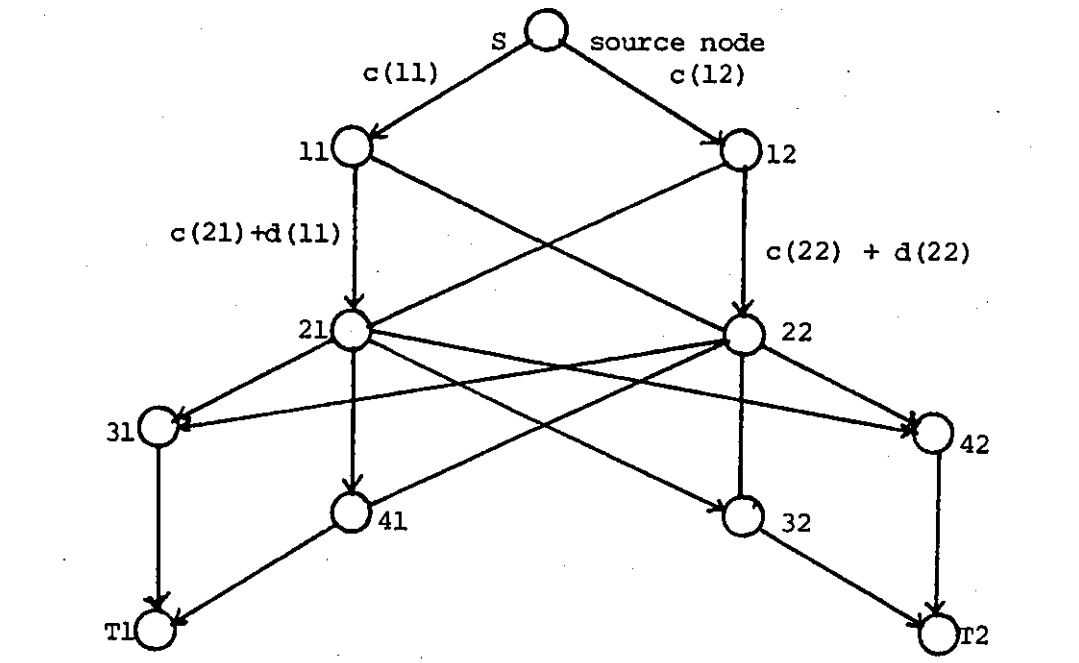
b) It has one distinguished node called the source node, denoted by S

c) It has one or more sink (or terminal) nodes, denoted by T_1, T_2, T_3, etc., one for each leaf node of the invocation tree.

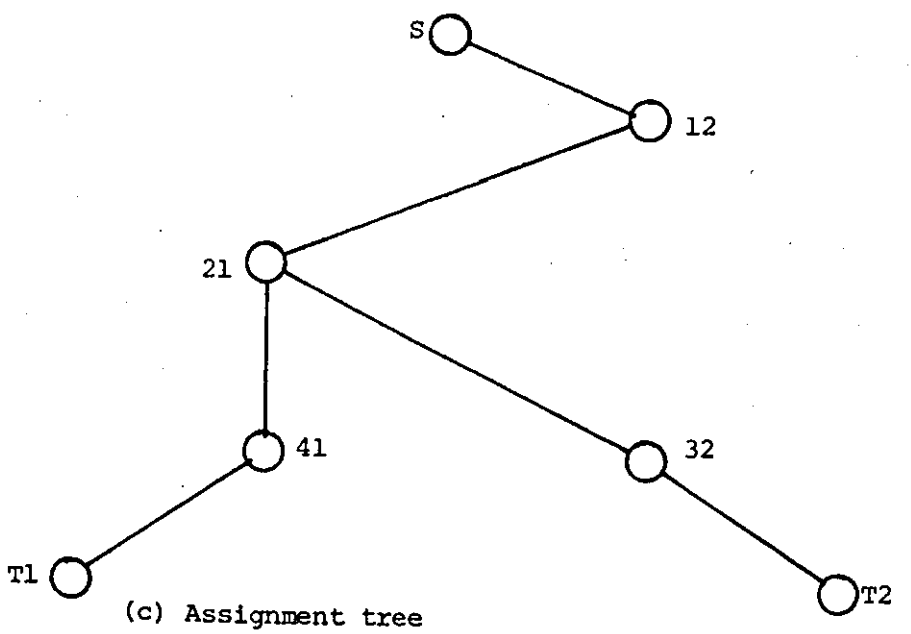
d) In addition to the source and sink nodes, there are pxn further nodes in the assignment graph (for a modular program with n modules



(a) Module Invocation Tree



(b) Assignment graph



(c) Assignment tree

Fig. 3.12 Shortest Tree Scheduler

and p processors). Each node is labelled with a pair of numbers (i,j) which represents the assignment of module i to processor j.

e) Each layer of the assignment graph corresponds to a node of the invocation tree (e.g. the layer comprising nodes (22) and (22) in fig. 3.12(b) correspond to node 2 of the invocation tree of fig. 3.12(a))

f) Nodes in layers corresponding to nodes in the invocation tree having outdegree greater than one are called forknodes. Each layer of forknodes is called a forkset.

In addition, the edges have weights on them according to the following guidelines:

a) All edges incident on the sink nodes T_1, T_2, etc., have zero weights on them.

b) The edges joining the source node S to nodes (11), (12), etc., have weights C(11), C(12), etc., which represent the time to execute module 1 on each of the processors 1, 2, etc., in the distributed processor system.

c) The edge joining node (i p) to node (j q) has weight equal to the sum of the time-to-execute module j on processor q, i.e. C(j q), and the intermodule communication time for assigning module i on processor p, given that module j has been assigned to processor q.

Hence, to each assignment of the n modules to the p processors, there corresponds some subset of nodes of the assignment graph. The subgraph generated by these nodes together with the source and sink nodes, is called an assignment tree, as shown in fig. 3.12(c), and has the following characteristics:

a) It is a tree

b) It connects the source node, S, to all the sink nodes, T_1, T_2, etc.

c) It contains one, and only one, node from each layer of the assignment graph.

In this way, there is seen to be a one-to-one correspondence between the assignment trees and module assignment. Furthermore, the weight of each assignment tree, (i.e. the sum of the weights of all edges forming it), equals the total processing time of the corresponding assignment. To find the minimum cost assignment, it is only necessary to find the minimum weight assignment tree in the assignment graph.

3.4.4 Module Scheduling Time

As was explained earlier, the time taken to run a scheduler depends on the number of nodes and edges in the basic network, in the case of the enumerative and the max-flow min-cut algorithms. The time to run the max-flow min-cut scheduler is proportioned to the number of the augmented (S,T) flow paths, which in turn depend strongly on the relative weights of the various edges of the basic network. The number of edges in a fully connected connected intermodule graph is equal to $\frac{1}{2}.n.(n-1)$. The basic network formed from such an intermodule graph has $\frac{1}{2}.n.(n+3)$ edges. In such a basic network the total number of all the possible (S,T) flow augmention paths is

$$\sum_{i=1}^{n} \frac{n!}{(n-i)!}$$

where n is the number of modules of the modular program, or the number of nodes in the intermodule graph (ref. 3.37). In the case of the modular program consisting of three modules A, B and C of fig. 3.9, the total number of all the possible (S,T) paths are as shown in fig. 3.13. However, not all these paths are scanned during the running of the max-flow min-cut scheduler, as many of them are collapsed and deleted during the first

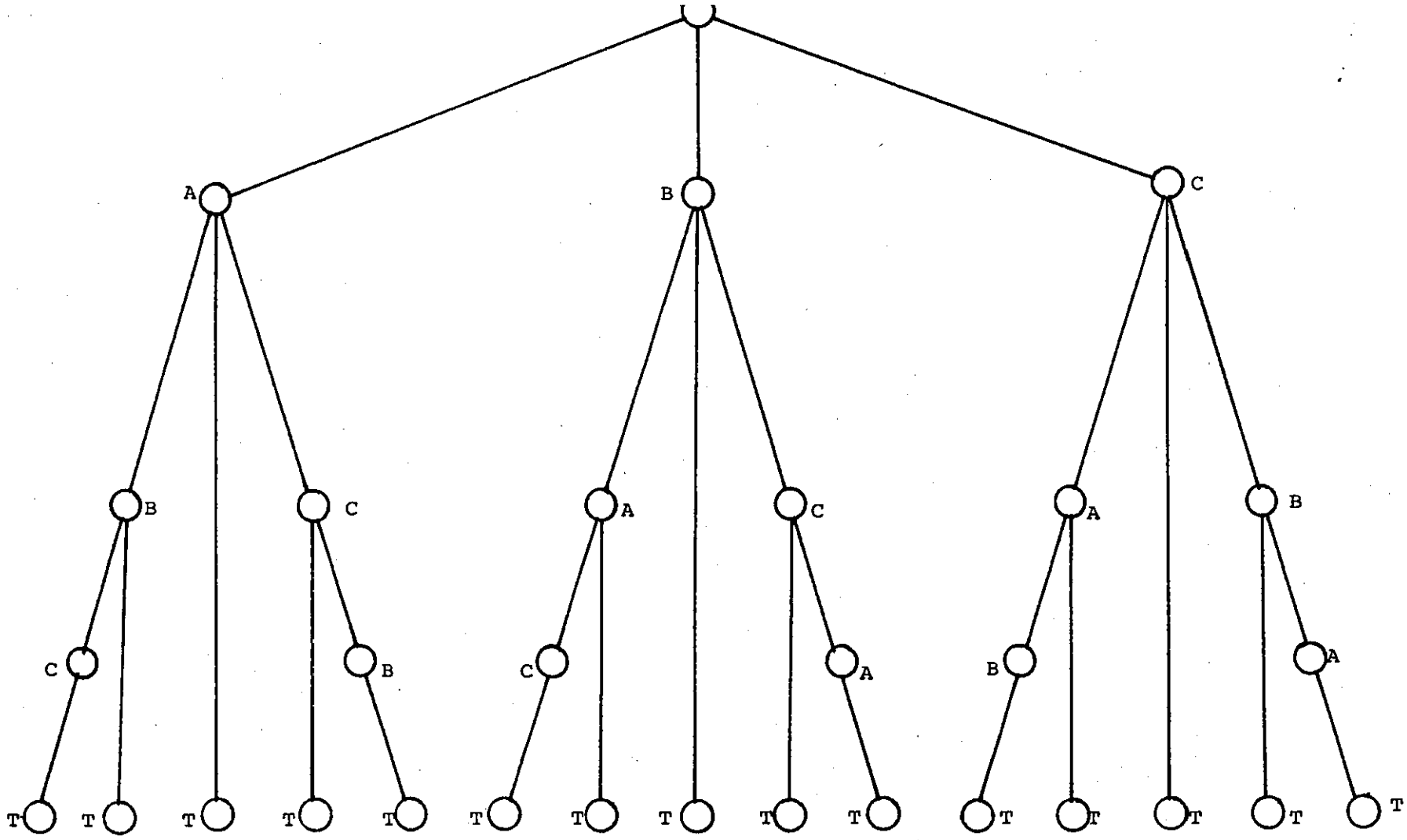


Fig. 3.13 Total possible scannable (S,T) paths

n searches of the algorithm. The maximum number of paths that can be scanned is only

$$\sum_{i=1}^{n} i = \frac{1}{2}.n.(n+1),$$

while the minimum number of such paths is n. The $\frac{1}{2}.n.(n+1)$ (S,T) flow paths that are actually scanned in the case of the previous basic network of fig. 3.9, are as shown in fig. 3.14.

The time to run the enumerative scheduler, on the other hand, strictly depends on n, the number of nodes in the intermodule graph, and is proportional to 2^n, the total number of cuts in the cutset, as shown in fig. 3.15. Hence, by an appropriate choice of the values of the arc weights in the basic network, it is possible to investigate how the value of the minimum scheduling time varies with the number of modules of a modular program.

3.4.5 Time Performance Comparison of the Max-flow Min-cut and the Enumerative Schedulers

Fig. 3.16 shows how the minimal scheduling time varies with the value of n in a dual processor system. These values are obtained by using three different computers with different speed and processing power capabilities. The arc weights were chosen in such a way that only the minimal n (S,T) flow augmentation paths were searched and scanned in order to completely accomplish module assignment to the processors. Similarly fig. 3.17 shows how the value of the scheduling time varies with n when the arc weights are slightly more randomized. In this case the total number of the (S,T) flow paths may lie anywhere between the minimal n and the maximum $\frac{1}{2}.n.(n+1)$ during any one scan. Three pairs of curves

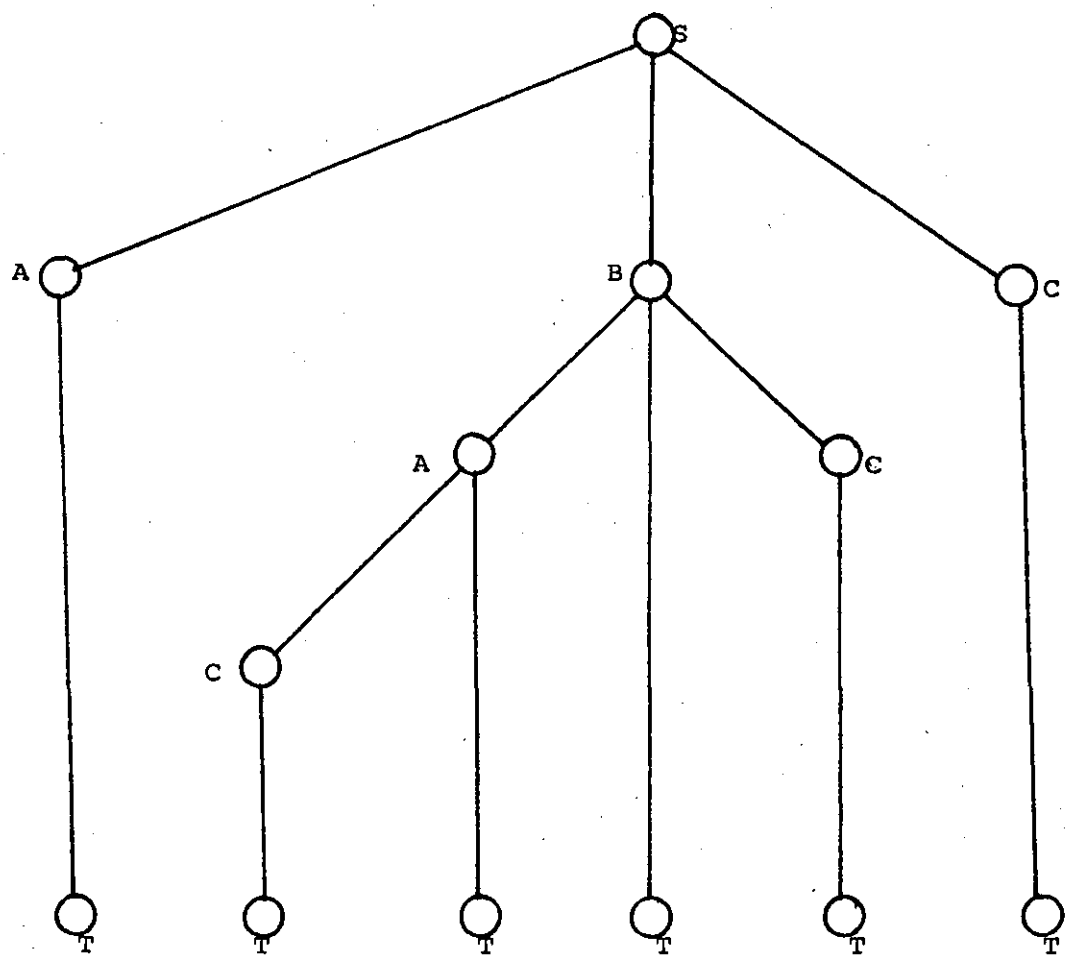


Fig. 3.14 Scanned (S,T) paths

Cut Number	Module Assignment	
	S	T
1	None	All ; A, B, C
2	All : A, B, C	None
3	B	A, C
4	A, B	C
5	B, C	A
6	A, C	B
7	A	B, C
8	C	A, B

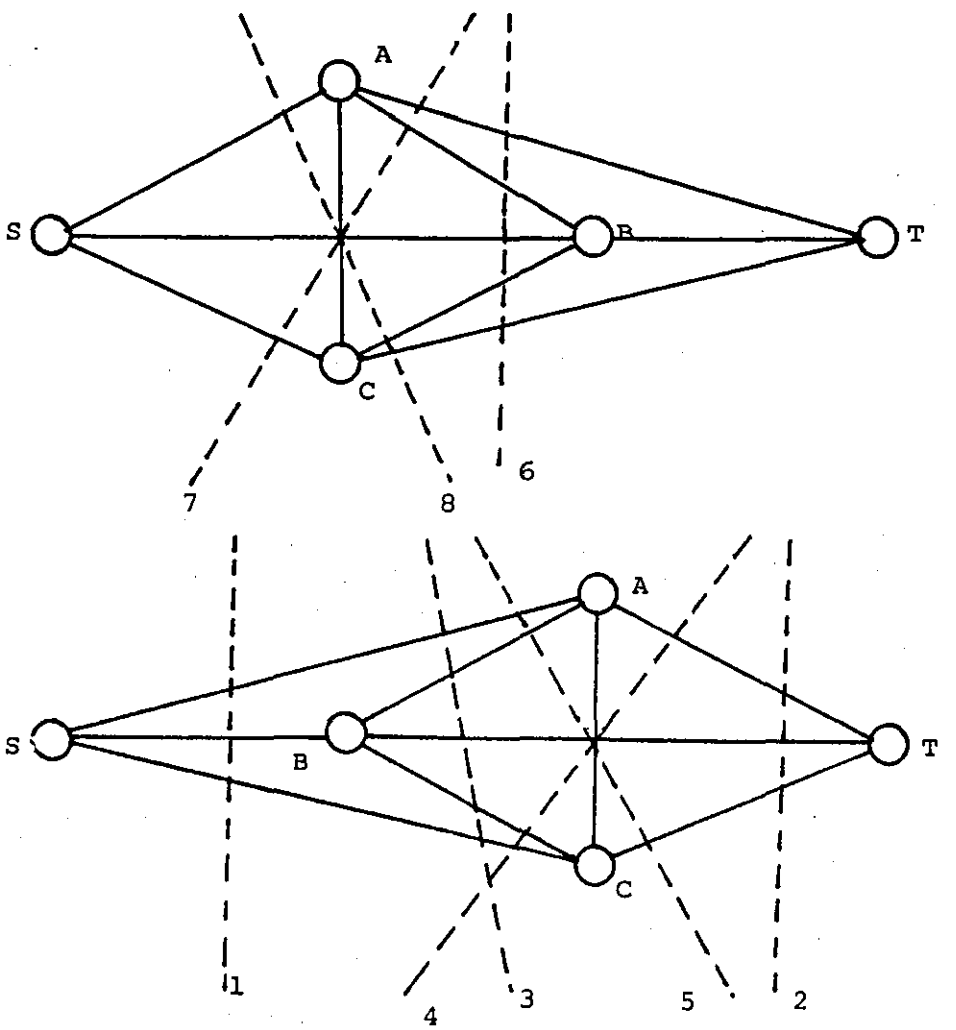


Fig. 3.15 The Enumerated Cuts

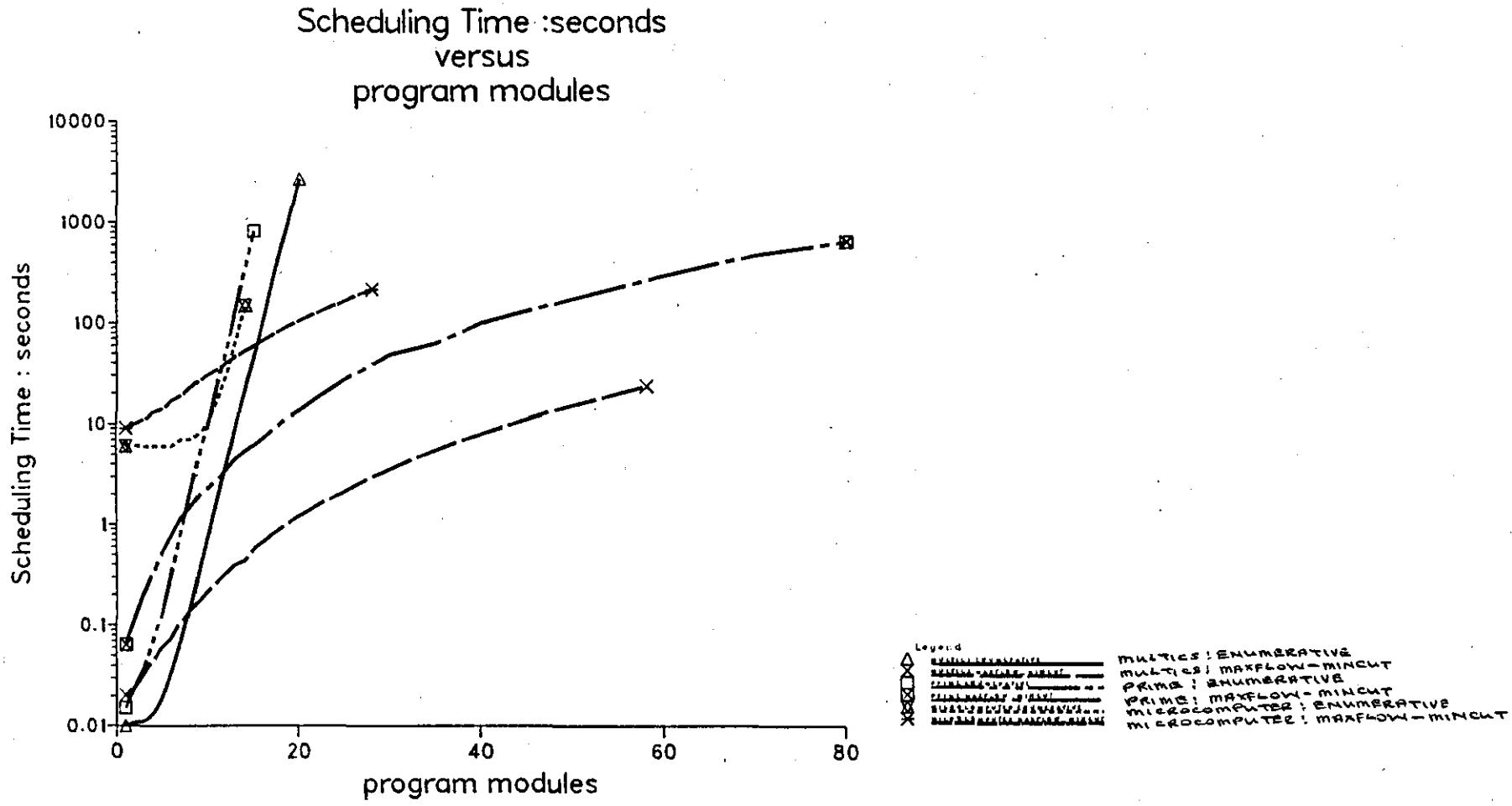


Fig. 3.16 Minimal Scheduling Time

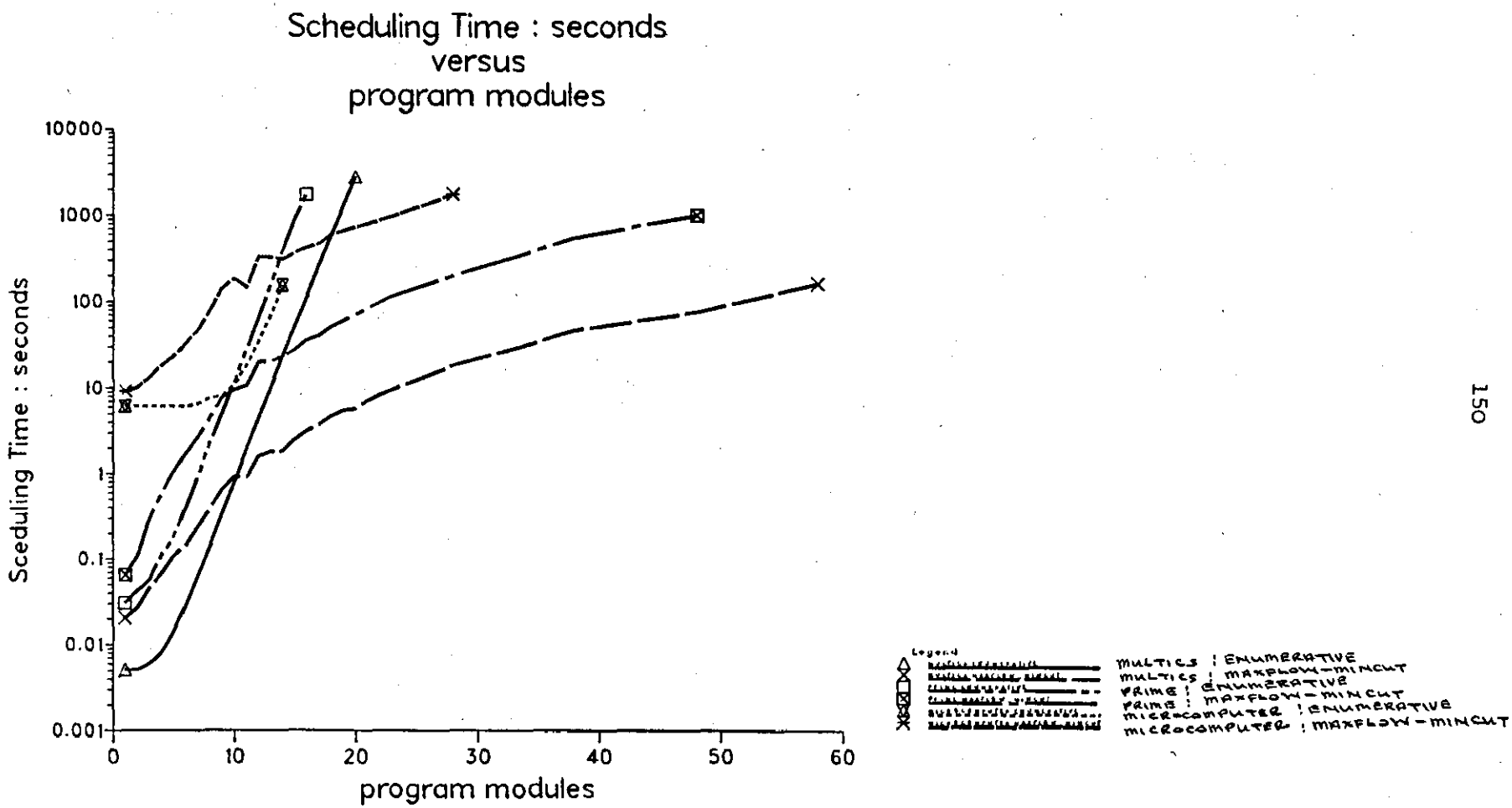


Fig. 3.17 Randomized Scheduling Time

are shown in the two figures: three curves for the enumerative scheduler on the three computers and three other curves for the max-flow min-cut scheduler on the three computers. The three computers used were:

a) An 8-bit microcomputer employing the Intel 8088 processor, with a clock frequency of 8 MHz, and having a 1 Mb (Megabyte) memory. This computer runs the CP/M-86 operating system.

b) A multiprogrammed, multiaccessed, and time-shared mainframe computer with a virtual memory space of 2.75 Mb. This computer was the Prime A computer of the computer centre of the University.

b) A faster and more powerful, multiprogrammed, multiaccessed, and time-shared mainframe computer with a virtual memory space of 8 Mb. This was the Multics computer of the computer centre of the University.

The scheduling time is the CPU time of the computers. Also, the two scheduling algorithms were both coded in Pascal. From this comparison it can be seen that the max-flow min-cut algorithm is more efficient than the corresponding enumerative algorithm. Also, below a critical number of modules, the enumerative scheduler performs faster than the max-flow min-cut scheduler.

3.5 THE MODULE INTERACTION ENVIRONMENT

In the dual processor distributed computation environment of the type considered herein, the source and the sink processors are dissimilar. For example, a specific processor may possess a hardware floating-point unit and thus be able to carry out floating-point operations with a higher speed than a processor without such a hardware facility. Similarly, other processors may be able to perform byte manipulations more efficiently than others. Alternatively, the distributed processors may be from the same

computer family, with varying computational power, but can execute the same instruction set (for example, a system based on the PDP-11 family computers). But, although the processors may be dissimilar, each program module is able, in principle, to run on either of the two processors. This may be achieved if, for example, all the modules are coded in a procedure-oriented high-level language and separate object versions of these modules are made available for each processor. Despite the dissimilarities, the major goal of the dual processor system is to provide the programmer with easy access and use of the two processors. In this way, the programmer is provided with a set of more powerful tools to enable him to divide his applications program between the source and the sink processors without constant reference to the fact that he is working with two dissimilar processors.

The most important facility in such an environment is that of module allocation and reallocation (or movement) between the source and the sink processors, whenever the need arises. Thus the application programmer should be free to move, at run time, various pieces of the application between the two processors. As a consequence of this module movement, the dual processor system supports interprocessor module calls. Once a module has been moved, say, from the sink processor to the source processor, a mechanism is required which can trap calls to that module and pass the required parameter information to the remote copy of the module. Such a kind of mechanism can be thought of as being analogous to the high level access methods supported by the operating systems in which all hardware details, communications protocol, timing dependencies, etc., are hidden from the user.

The dual processor system can affect the implementation of the applications in the following three areas corresponding roughly to the usual compilation, link-edit, and execute sequence (ref. 3.38, 3.40).

3.5.1 Module Language Features

A high-level language with translators and capable of generating the code for both processors. Such a high-level language, as mentioned earlier, should let the programmer disregard, as far as possible, the differences in the hardware and operating system characteristics of the two dissimilar processors. It should also provide the following general features necessary for the dual processor environment:

a) Machine Transportability: There should be either two equivalent compilers for the language for the sink processor and the source processor, or one compiler capable of generating the code for both machines. It is also desirable to have only one language which can run on both machines.

b) Procedure Oriented: It is necessary that the language be procedure oriented (but not necessarily block structured). A procedure oriented language provides a very natural mechanism for dividing an application into modules which can be dynamically moved between the processors.

c) Symbol Table Output: If the module movement and the transfer of module calls between the sink and source processors is to be supported, then it is necessary that the compiler for the language retain complete information about all the module symbols and module parameters defined by the programmer for later use by the link-edit preprocessor.

From the above, it may be noted that the use of an assembly language is not precluded, provided a sufficiently powerful assembler is available.

3.5.2 A Link-Edit-Time Preprocessor

This should be capable of setting up the information necessary for reallocation of modules to either the sink or the source processors. After all the modules of some application system have been coded and compiled, so that some modules have been assigned to the source processor and the rest to the sink processor, the following situation prevails, (ref. 3.40):

a) A set of source object modules

b) A set of sink object modules

c) A set of symbol tables, one table for each module.

If all the modules are assigned to the source processor alone, so that none processes at the sink processor, then the next step would be to invoke a link editor which would combine the various object modules into a load module. The same situation would prevail if all the modules are assigned to the sink processor alone. But in a dual processor distributed computation environment, a new stage between compile and link edit is introduced to accomplish the following twofold purpose:

a) To set up linkage information necessary for the dual processing run-time environment to resolve intermodule calls across the communications interface

b) To save from the compile stage sufficient symbol information so that modules can be moved at run time from one processor to the other.

Thus, the link-edit preprocessor performs the following three important functions:

3.5.2.1 Scanning The Input

To scan the three input streams corresponding to the object modules, the object modules symbol table, and the user commands. The user commands are analogous to the link-editor commands in that they specify attributes

of the object modules and characteristics to be assigned the final load modules. In this case, the most important attributes are:

a) The initial locations, (source or sink), of the various modules

b) Whether or not the modules are eligible for being moved between the processors, (reallocatable). A non-reallocatable module has a module-to-processor weight equal to infinity in the basic network, so that it can only process at one processor alone.

3.5.2.2 New Object Modules

In order to create a new object module for each module, the new object module is given the name of the original object module, and the name of the original is altered in some unified fashion. For example, if a module is represented by procedure A, and is declared reallocatable, then the link-edit preprocessor creates a new module called A and renames the original A to, say, A'. The code in the new A is logically equivalent to that shown in fig. 3.18. This has the effect of routing all calls to the original A from other modules (or procedures) in the application system through the new version of A. In this way control is quickly passed to the real A (now renamed A'), or to the dual processor run-time environment, depending on the state of the "THE-LOCAL-SWITCH-IS-ON". The dual processor run-time environment has control over whether a local version of A is called or whether the call invokes a version of A in the remote processor.

3.5.2.3 Static Variables

The preprocessor is also changed with the responsibility of placing in each new object module information from the symbol table about all static variables and parameters for the reallocatable or remote module. This information, (e.g. variable type, length, etc.), is used by the run-time environment.

```
PROCEDURE A ( . . . . . ) ;

   ( * the code for the new A * )

BEGIN (* A *)

   IF THE-LOCAL-SWITCH-IS-ON (*a run-time environment bodean variable*)

      THEN (* the module runs locally *)

         BEGIN
           .
           .

           A' (. . . . .) ;
           .
           .
         END

      ELSE (* the module runs at the remote site * )

         BEGIN
           .
           .

           A' (. . . . ) ;
           .
           .

         END

END ;  (* A * )
```

Fig. 3.18 The New Object Module

3.5.3 The Run-Time Environment

The main functions of the dual processor run-time environment is to manage the routines that handle:

a) Interprocessor calls

b) Module movements

c) Taking measurements and statistics

d) Debugging facilities

e) I/O between the source and sink processors

The most important aspect of this environment is that it is completely transparent to the applications programmer. Such a situation is analogous to that of an overlay supervisor in that while writing a program to be placed in an overlay structure, the programmer does not need to worry about whether the program will be in memory when it is called or the details of how the program will be fetched from the secondary memory (ref. 3.40). The above main functions of the run-time environment may be summarised briefly in more detail as follows.

3.5.3.1 The Dual Processor Run-Time Monitor

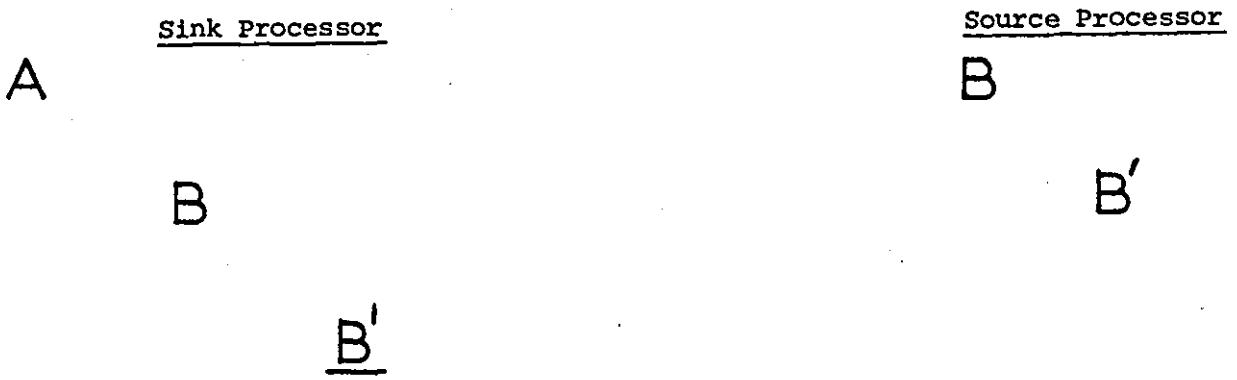
When running under the dual processor environment, a user program is controlled by a supervisory monitor. The main purpose of this monitor is to accept, interpret and execute commands issued by the user during the execution of the applications program. The monitor environment is entered either by a direct call from the user program or by a pre-defined asynchronous interrupt mechanism (e.g. an attention from the user's terminal). When the monitor is entered, the user's program is temporarily halted, and the user is prompted for a command. Possible commands include requests for module reallocation, statistics, and trace facilities.

3.5.3.2 The Intermodule Call Resolution

This is the basic function of the run-time environment, i.e. to resolve calls between modules which were written as if they were to run on the same machine, but which are in fact running on two separate machines.

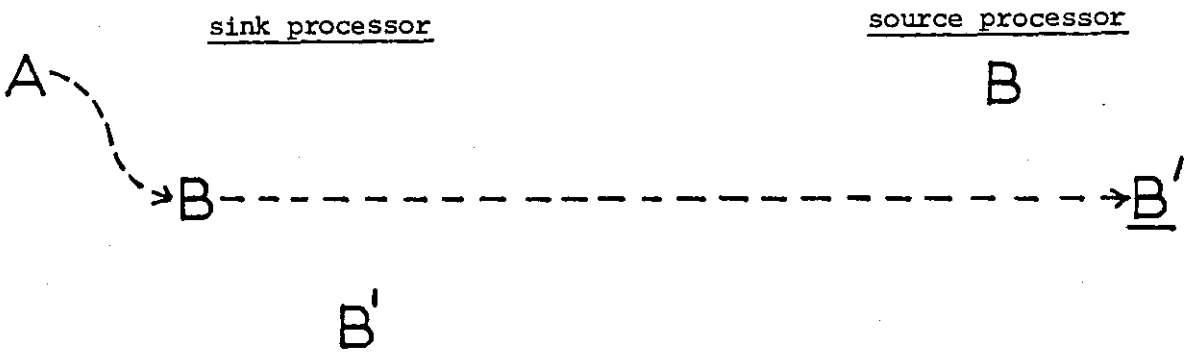
As a simple example, consider a modular program with two modules A and B, B having been declared reallocatable and initially resident in the sink processor. The module (procedure) allocation would then be as shown in fig. 3.19 (a), (the underlining indicates the copy of B currently being used). If now A calls B, control passes from A to B directly to B', the real copy of B, as shown in fig. 3.19(b). If the user now finds that the loading on the sink processor is unacceptably high and hence requests that B be moved to the source processor, any reference to B' in B is changed to point to the dual processor environment so that the next time A calls B, control is passed through this environment to the remote copy of B, as shown in fig. 3.19(c).

Passing control to the remote copy of B is only part of the problem. Of more importance is the problem of providing the remote module with a copy of the parameter list passed by module A. This is accomplished by referring to the symbol table information which was stored in the new B by the link-edit preprocessor. Using the information about the length and type attributes of the parameters which B expects, the appropriate variables are obtained, passed across the interface to the source processor, and converted (if necessary) into equivalent source processor formats. An appropriate list is then built in the source processor and passed to B', the original satellite copy of B. Similarly, when B returns to A, any



(a) Initial module allocation

(b) Intermodule referencing



(c) New module allocation

Fig. 3.19 Run-time Environment

modified parameters are passed back to the sink processor and modified in the processor memory.

3.5.3.3 Module Movements

The mechanism for moving a module between processors is very similar to that of resolving the interprocessor calls. The user first enters the dual processor monitor and specifies the module to be moved. The monitor then calls the routine in charge of the module movement. This routine first marks the local copy of the module as inactive, and reference to B' in B is replaced with a reference to the remote copy. The module movement routine then calls its counterpart in the remote processor and passes it the static variables which were declared in the module to be moved. These variables are treated just like parameters, except that instead of being placed in a parameter list for the remote module, they are used to update the static environment of the remote module so as to reflect the current state of the (now inactive) local module. The remote copy of the module is then marked as active, control returns to the monitor, and the user's program is restarted.

3.5.3.4 Statistics, Measurements, and Debugging

For the purposes of being able to vary the module to processor assignment is the ability to measure the performance of the application program as a function of this module allocation to processor. In this way the user can be provided with such statistics as follows:-

a) The mean execution time for a given module

b) The paging rates

c) The amount of interprocessor data transfer

d) The mean delay for the interprocessor data transfer

e) The mean time between intermodule calls.

Using these figures, the user can modify module allocation so as to optimize important parameters, such as

a) The total execution cost

b) The response time, etc.

Another important facility is the ability to imbed sophisticated debugging tools into the system. Not only does the dual processor system provide extensive symbol table information at run time, but a natural module-level breakpoint system is built in. In this way, it is possible to build in a mechanism which would dynamically check offered parameter lists against those expected by the called modules. Errors of this kind are a common source of bugs in large software systems.

CHAPTER 4

COMPUTATION TIME

4.1 INTRODUCTION

As explained earlier, the view of distributed computation taken in this thesis is that a single problem is solved by a number of single-CPU general-purpose serial computers that are spatially distributed within a geographically small area. Specifically a CPU cache model of distributed computation is employed. The computer's single processor accesses a single memory, and inputs and outputs information to and from the external source. The processor's ALU (arithmetic and logic unit), which is often made up of a whole set of simple special-purpose processors, plus the CU (the control unit), and high-speed registers, make up the single CPU. In some cases, there may be several kinds of input and output (I/O) devices, and a hierarchy of progressively slower but larger memories. But all the computation is performed by the single processor, using the data and the program stored in its main memory. In some cases too, the computer may have special-purpose hardware for several types of common processes such as addition, floating-point multiplication, division, string matching, input and output. Hence, the computer's single processor is really a collection of a whole set of specific processors. Consequently, the computational power and speed performance among these single-CPU general-purpose computers, will vary as widely as their specialized hardware and their instruction set capabilities. Hence, it can be expected that the time taken to process a given computational task will also vary widely among such computers. The time taken to complete a given computational task too will depend not only on the particular

computer used but also on the nature of its underlying algorithm, (ref. 4.1, 4.2, 4.3).

This chapter briefly examines the principles of computation. It attempts to find and assign a number to a given problem or subproblem that represents the amount of computation time demanded by the problem or subproblem. It attempts to examine briefly what constitutes a computation, first with respect to the abstract Turing machine model of computation, and then examines the computer system software and hardware implementation and capabilities of practical computers. From this consideration it can be seen that, in order to assign a number representing the computation time of a certain process, it is also necessary to examine the collection many factors such as the

a) system software capabilities

b) system hardware capabilities

c) system implementation details

d) system memory capability and memory management

e) programming language efficiency and instruction set capability

f) programmer ability

g) the nature of the problem to be solved and the complexity of its underlying algorithm

All these factors and more determine the basic instruction times and the relative computational powers of the various existing computers. Furthermore, each instruction time will also be dependent on the nature of the input and output. For certain combinations of input the instruction times can be estimated by simple calculations whereas for the majority of cases this can only be done probabilistically. Hence, from the variety of the many factors that affect the instruction times, it may be seen that

probabilistic characterisation of computation is an important method of determining the computation performance measures.

4.2 PRINCIPLES OF COMPUTATION

From the point of view of a computer's ability to execute an algorithm (or a program), all modern general-purpose computers are the same (ref. 4.1). What distinguishes between them most is mainly the time taken to execute the given algorithm. What computers can and cannot do can be examined with the aid of simpler abstract models of computers.

4.2.1 Computation

Turing (ref. 4.4) and Post (ref. 4.5) independently proposed a very simple kind of a computer. The computer they proposed was a suitably powerful finite-state automaton (called a Turing Machine), possessing an infinite memory (tape) and capable of doing anything that any other computer might conceivably do, if given enough time. A number of other identical formulations of such a computer have been used (ref. 4.6, 4.7, 4.8). Fig. 4.1 shows Post's formulation of the basic structure of the Turing machine. The basic structure of such a computer model has a read-write head which is capable of looking at each tape-section of an arbitrarily long (potentially infinite) tape with symbols on each tape-section. It also has a set of internally stored instructions for reading and printing symbols onto the tape, and has the capacity too for shifting the tape. The model computer is also assumed to be capable of performing the following basic actions:

a) Marking the current tape-section (assumed empty)

b) Erasing the mark in the current tape-section (assumed marked)

c) Moving to the next tape-section on the right

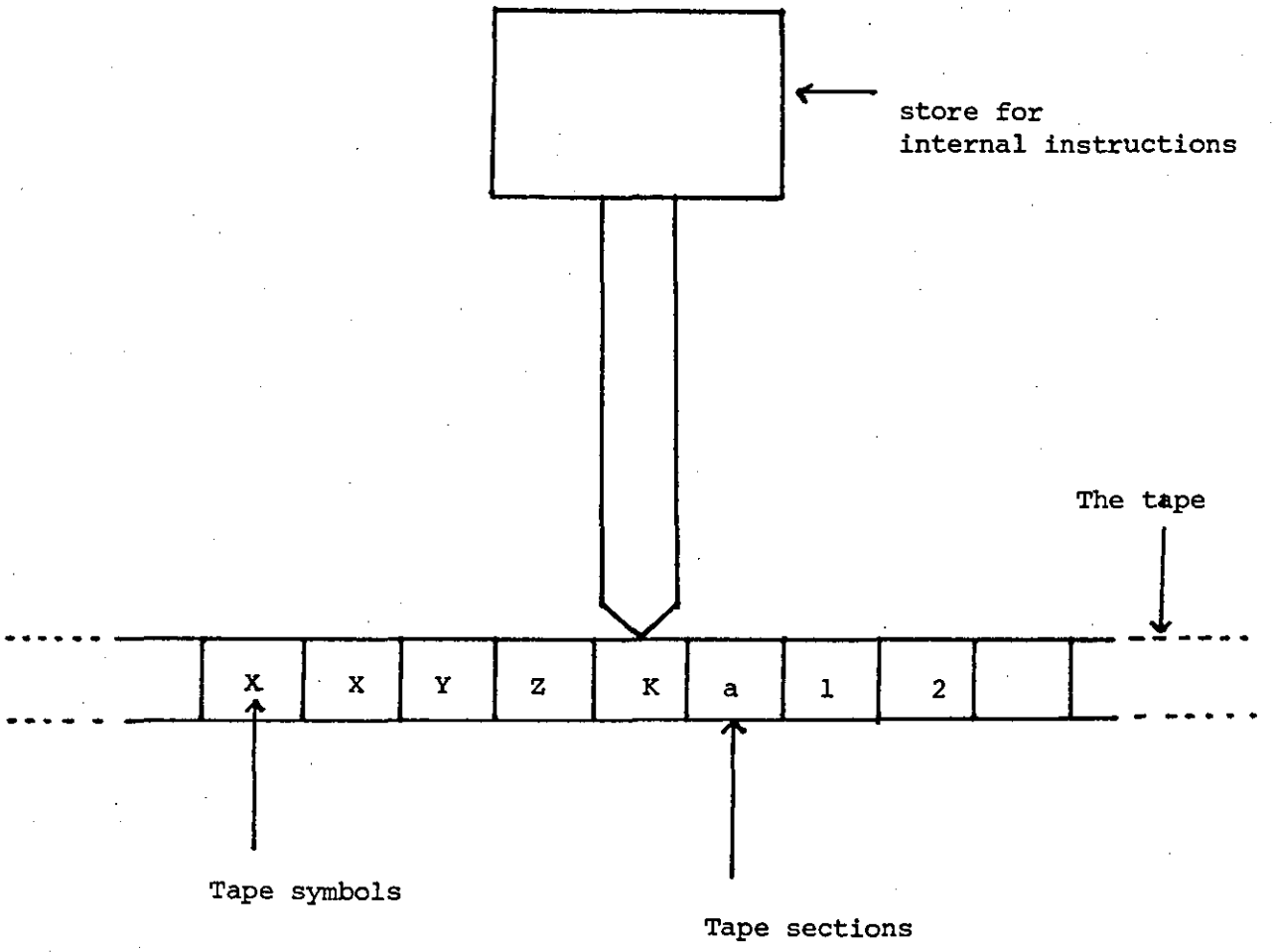


Fig. 4.1 Turing Machine Model

d) Moving to the next tape-section on the left

e) Determining whether the current tape-section is marked or is not marked

The Turing machine is a very simple logical construction of a computer that has been used to prove the generality and equivalence of all modern general-purpose computers (ref. 4.1, 4.9, 4.10), such that each can compute anything, or carry out any algorithm, that any other can, if given enough time. An infinite memory potentiality of the finite-state automaton ensures that as much memory as is needed is available for whatever program it is presented to execute. In this connection, an automaton is a machine that responds unthinkingly to a stimulus, in accordance with pre-determined rules, without any scope whatever for intuition or discretion, (ref.4.11, 4.12). Its response will depend only on the stimulus which it receives and the state in which it is in when it receives that stimulus. Such finite-state machines proceed in separate and discrete steps from one to another of a finite number of states. There is a direct relationship between their basic structure and their behaviour. Given its initial state and the input signals, it should be theoretically possible to deduce the state it will be in at any particular instant. To this end, a computer is basically a finite-state machine and

a) it has no power of its own for direction and reacts to any stimulus in exact accordance with the flow of its instructions,

b) although the number of its distinct states is very large, the ultimate number of such states is finite.

Thus, a general-purpose computer can be very simple since all that the Turing machine model of a computer does is to execute a sequence of instructions from the following repertoire:

a) READ the current symbol

b) SHIFT to the next symbol

c) WRITE the current symbol onto the current memory (tape) location

d) IF the current symbol is C, THEN do instruction S_t, ELSE do instruction S_e

In this way, information is read from and written onto the tape which serves as the system memory. The tape contains both the program data and the program code. A computer modelled in this way is a stored-program computer and the processor that executes sequences of these instructions is an example of a finite-state automaton (ref. 4.9, 4.10). One of the major sources of the computational power for such stored-program computers lies in the fact that the program is input to and stored in the same memory that contains all other kinds of data. But the Turing machine model computer executes very slowly because the instructions are weak and low-level (ref. 4.17, 4.24). The two major factors that have been used to improve computer capability in simplifying the program to a much shorter sequence of much more powerful instructions, and in reducing the time taken to execute a program are:

a) The processor can read, write and operate on a whole word

b) The processor is given the random access capability to immediately fetch any of a rather longer number of such words.

The program code runs the computer. The first instruction is loaded into the instruction register and decoded to determine the operation to be performed, and the address where

a) The operands are to be fetched

b) The results are to be stored

c) The next instruction is to be found.

The operands are then fetched and processed, the results stored, and the next instruction fetched and loaded into the instruction register.

This process continues until the "end" instruction is reached, at which point there is no next instruction. The time to perform each instruction is very short, ranging from several microseconds to a fraction of a microsecond, depending on the instruction type and the hardware characteristic of the CPU. The total execution time of the program or program module will hence be the total sum of the instruction execution times of all the various instructions constituting the program code. All modern computers are general-purpose in that each can compute anything that any other computer can compute, given enough time. But the larger computers are capable of executing certain programs very much faster than the smaller ones because they tend to have a variety of more specialized hardware which may be more suitable for certain algorithms, (ref. 4.13, 4.14, 4.15). In general, the computational power of these computers is achieved by giving the computer

a) A set of instructions that includes the basic Turing machine functions
b) Enough amount of memory sufficient to handle the program to be executed.

In this way, all modern computers will be millions of times faster than the Turing machine model computer. They are also a lot more complex, and having several kinds of I/O devices and successively faster memories. Often, the I/O functions are handled by special I/O processors working in parallel with the CPU. In some special cases, the next program instruction is often fetched at the same time that the present instruction is being executed. Several high-speed registers are usually employed to store the current instruction, its operands, and its resulting output. A larger cache memory of high-speed registers is often used to contain instructions and data that will soon be needed. The CPU too may employ

special-purpose hardware for several types of processors such as addition, floating-point multiplication, division, string matching, and I/O.

Hence, the computer as a piece of machinery is only capable of performing only a small number of simple operations, i.e.:

a) Data or information storage

b) Data movement from one location in memory to another

c) Performing simple arithmetic operations

d) Performing simple logical operations

e) Interpreting instructions

f) Data or information input

g) Data or information output

h) Starting and stopping.

The computer solves all the problems presented to it, both simple and complex, through combinations of the above primitive operations. From this point of view, its purpose is to carry out instructions, (ref. 4.16) These instructions are contained in the program statements which make up the program. In almost all programming the program statements can be classified according to the actions they perform. All such statements belong to one of six classes:

a) Input

b) Output

c) Assignment

d) Control (i.e. selection, branching, and repetition)

e) Termination

f) Comments

The two main types of data that the program acts upon are constants and variables.

4.2.2 Instruction Times

In performing its computations a computer fetches an instruction from a memory location, decodes the instruction, and executes the instruction as explained above. In high-speed computers, one of the main factors limiting the speed of operation is the performance of the memory cycle time. The time taken to complete an instruction is dependent upon

a) The type of an instruction - which is defined by the function digits
b) The exact location of the instruction and operand in the core or fixed store - since this can affect the access time
c) Whether or not the operand address is to be modified
d) In the case of the floating-point accumulator orders, the actual numbers themselves
e) Whether drum and/or tape transfers are taking place.

But obeying one instruction may be overlapped in time with some part of other instructions. In this case the single most important parameter is the performance of the processor. In the past, processor performance has been measured in instructions per second. The number of instructions per second can be estimated by using the time of a single representative instruction, or by the average instruction execution time (assuming all instructions to be equally likely). A more accurate measure is a weighted average of instruction execution time using weights derived from a general mix or from the intended application. The average instruction execution time is sometimes chosen because of its obvious relationship with the instruction stream throughput. But this method neglects such overhead factors as

a) direct memory access
b) interrupt servicing
c) dynamic memory refreshing

The average instruction execution may be obtained by either benchmarking or by calculation from instruction frequency and timing data (ref. 4.18, 4.19). The latter has the advantage that it has freedom from the extraneous factors noted above and from the normal clock rate variations found from machine to machine of a given model. This method also allows for the calculation of the change in average instruction execution time that would result from some change in the implementation. In this way the average instruction execution can be calculated from

$$t = k_1.c_1 + k_2.c_2$$

where

c_1 = the microcycle time

c_2 = the memory-read-pause time. (The memory-read-pause time is the period of time during which the CPU clock is suspended during a memory read)

k_1 = the number of microcycles expected in a canonical instruction

k_2 = the number of memory accesses expected in a canonical instruction

On the other hand, the typical instruction time for a simple operation, such as ADD, can be estimated. Such a metric is an approximation for the average instruction time and assumes that:

a) The machines have about the same Instruction Set Processor (ISP), and hence there is little difference among instructions

b) A specific data-type will be used more heavily than another

c) A typical add time will be given (e.g. the operand is in a random location in the primary memory cell rather than being cached or in a fast register)

It is possible to determine the average instruction time by executing one of every possible instruction. However, since the instruction used depends so much upon the program data they interpret, this metric is not

very accurate. A better measure is to keep statistics about the use of all programs and to give the average time based on the use of all programs. Such a measure may be used to compare two different implementations of the same architecture. Early attempts to make more accurate characterisations were based on weighing the instruction use (i.e. forming a typical utilization U) according to task (e.g. floating-point versus indexing and character handling) to give a better performance measure. Thus, instruction mixes were developed which better evaluated performance. Studies of frequency counts of instructions have been described by several authors. The best known is the Gibson mix (ref. 4.20), developed at IBM in 1959. Gibson divided the instructions of the IBM 704/650 computers into thirteen classes and counted how many instructions of each class were executed. His sample size was seventeen programs with approximately nine million instructions. Similar studies have been carried out (ref. 4.17, 4.21) as shown in Table 4.1.

For a given application, weighted average of the instruction execution times may be determined by:

a) Preparing a table of frequencies of various types of instructions based on experience in similar applications, or perhaps, on actual counts if these are available

b) Obtain the total weighted execution time for a given instruction type by multiplying the time required for the instruction by the frequency count

c) Calculate the average execution time by summing overall instruction types and dividing by the sum of the frequency counts.

Current ISPs designed for scientific applications have word lengths ranging from 24 to 64 bits; the number of different instructions varies from 70 to over 400; register structures span the area from one accumulator

	Machine					
Class	IBM 650/704, Gibson's results	CDC 3600, U. Mass.† results	PDP-10, CMU‡ results	PDP-11, DEC results	S/360, U. of Toronto results	HP 3000, HP results
Load, store	31.2	30.0	42.4	22.4	48.1	34.0
Branches	16.6	38.3	28.2	33.7	17.7	16.0
Fixpoint add, subtract	6.1	1.2	12.4	19.0	10.2	
Compares	3.8	1.2	....	12.5	7.0	
Floating add, subtract	6.9	0.5	4.9	0.0	0.0	
Floating multiply	3.8	0.5	2.6	0.0	0.0	33.0
Floating divide	1.5	0.2	1.1	0.0	0.0	
Fixpoint multiply	0.6	0.1	1.1	0.0	0.0	
Fixpoint divide	0.2	0.1	0.5	0.0	0.3	
Shifting	4.4	2.2	3.9	4.6	4.4	1.0
Logical	1.6	0.5	1.0	4.3	4.9	5.0
Miscellaneous	5.3	0.0	1.5	3.3	7.0	11.0
Indexing	18.0	13.4	....	0.0	. . .	
Fullword	. . .	6.9	....	0.0	....	
I/O control	....	0.0	0.1	....	0.0	
Interregister transfer	....	5.0	....	0.0	0.0	

†U. Mass = University of Massachusetts.
‡CMU = Carnegie-Mellon University.

Table 4.1 Percentage of Executed Instructions

plus a few index registers, through designs with 8 to 24 general or specialized registers, to designs with up to 64 registers.

4.2.3 Benchmark Programs

A benchmark program (ref. 4.22), may also be used to characterise the various performance measures of a computer. It may be the simple expression, A:= B + C, composed of one statement and two operations (:=, +). The statement execution rate is taken to be the actual computer performance and reflects the highest performance for the three address machine, whereas the conventional instructions per second measure may show different values. A more subtle performance measure is the operation rate and this is found to be more correlated with the true benchmark statement execution rate. For example, in the case of the primary memory the operation rate may be taken to be the information rate (the number of word accesses per second) and this is found to be a better performance indicator than the conventional instruction rate measure. Also, for the more unconventional vector or array computers (e.g. ILLIAC IV, CDC STAR, CRAY-1) which have to operate on at least 64 operands per instruction, instructions per second would be a poor measure, (ref. 4.23).

A carefully designed standard benchmark gives the best performance estimate because the benchmark is fairly understood and can be run on several different machines. In this way specific benchmark programs that reflect a particular type of workload can be used since whether a standard benchmark is of much value in characterising performance depends on the degree to which it is typical of the actual computer's use. A further advantage of standard benchmarks is that they may be written in the high-level language to be used by the computer and hence they reflect the application as well as characterising the language machine architecture.

One of the strongest advantages of the benchmark method is that it can handle a total problem and integrate all features of the computer. The main difficulty with benchmarks is that the result depends not only on the type of the computer, but also on the exact configuration such as:

a) The number of words in the primary memory

b) The operating system characteristics.

Thus, although the benchmark performance number perhaps comes closest to serving as an adequate single performance figure, it is weaker as a parameter characterising the structure of the computer than one characterising a contingent total system.

4.2.4. System Performance Measures

Hence, in order to measure the performance of a specific computer it is necessary to know the ISP, the hardware performance, and the frequency of use for the various instructions. The execution time T is the dot product of the fractional utilization of each instruction (U_i) and the time (t_i) to execute each instruction. As mentioned earlier, the instruction utilization (U_i) can be estimated by:

a) Defining a typical or average instruction

b) Using standard benchmarks to characterise the machine performance

c) Using a specific or unique benchmark when the actual use has not been characterised in terms of the standard benchmark.

These quantities t_i and U_i are based entirely on measures which may also be computed automatically from a computer program during compiling. Hence the compiler may be forced to give an estimate of the execution time T of a module or a program.

4.2.5 Memory Size

The memory size in bytes, for both the primary and the secondary memory give memory capability of the computer. The memory transfer rates are necessary as secondary performance measures, especially for memory interference when multiple processors are used. The primary memory transfer rate also tracks the access rate available to the CPU for secondary memory transfers and external interface transfers. For file systems, which require multiple accesses to the secondary memory for single items, the probabilistic measure of the access rate is necessary for a more accurate performance estimate. Similarly, for multiprogrammed systems, which use secondary memory to hold programs, the probabilistic measure of program swapping rate may be required.

Other secondary CPU parameters include the number of data-types and the context-switching rate. The number of data-types (such as the scientific string, character, lists, vectors, etc.) in the CPU gives an indication of performance when it is operated with a particular language. In the case of multiprogramming systems, the time to switch from job to job is important. In this case the process context-switching rate is an important attribute, since most large computer systems operate with some form of multiprogramming.

4.3 ALGORITHMS AND COMPUTATION

The concept of an algorithm is fundamental in the solution of problems, (ref. 4.1, 4.2). To determine what a computer can and cannot do, it is necessary first to consider what can be accomplished by an algorithm. An algorithm may be defined as a finite set of rules, which gives a sequence of operations for solving a specific type of a problem. From such a definition it can be seen that an algorithm possesses the following properties, (ref. 4.36):

a) Finiteness: When an algorithm is mechanically executed, it must terminate after a finite number of steps.

b) Definiteness: The actions to be carried out and the sequence of steps to be executed for each step of the algorithm are unambiguously defined.

c) Completeness: The rules of the algorithm are complete so that it can solve all the problems of a particular type, and for any input data, for which it is designed.

d) Effectiveness: All the operations used in the algorithm are basic and are capable of being performed mechanically, and without the necessity for any intuitive step

e) Input/Output: An algorithm has certain precise inputs, or initial data, and the outputs are generated in the intermediate as well as the final steps of the algorithm.

If the algorithm forms an input into a machine, then the machine should be capable of

a) Reading and interpreting the input and intermediate data

b) Carrying out primitive mechanical operations as demanded by the algorithm

c) Mechanically controlling the sequence of steps

d) Storing the data

e) Outputting the result

If a program is aimed at a particular problem, one can think of it as searching for a solution to that problem. The search itself might never end, and hence no solution can be found for that algorithm. In this way it is possible to investigate whether or not there are some problems for which there cannot be an algorithm, (ref. 4.25). Consequently,

in order to determine the computation time of a particular problem or the computational capabilities of computers it is also necessary to examine the efficiency of the underlying algorithm. Different problems or sub-problems will require different algorithms and hence different execution times. The more efficient the algorithm the shorter the time taken for its computation, for a given size of the input. The underlying implication is that algorithms and computation are inseparable. As was explained earlier, one of the major goals in distributing a computation is to speed up the given computation by using an interconnection of two or more computers within a LAN. Hence, speeding up a given computation can be achieved by designing and using better and more efficient algorithms to solve the problem on the computers, or by using faster and faster computers. But using better and more efficient algorithms can achieve a more dramatic reduction in the computation time than using faster and faster computers, (ref. 4.2).

To describe a problem as being solvable algorithmically implies, in general, that there is a computer program that will produce the correct answer for any input, if it is allowed to run long enough and if it is also allowed as much storage space as it needs. In as much as algorithms are essentially mechanical procedures requiring no intuition and capable of being executed automatically by a machine, various mechanical models of computation have been devised and investigated, (ref. 4.4, 4.9).

Perhaps the most important motivation for devising formal methods of computation is the desire to analyse the inherent computational difficulty of various problems. One would like to prove lower bounds on the computation time. In order to prove that there is no algorithm to perform a given task in less than a certain amount of time, one needs a precise and

highly stylized definition of what constitutes an algorithm. A Turing machine (ref. 4.4), model of computation is an example of such a definition. By using a Turing machine model, one can prove that a particular function requires a certain minimum amount of time to execute. Much of the emphasis in the early work in this field (called the computability theory) was on describing or characterising the nature of those problems that could be solved algorithmically and on exhibiting the nature of some problems that could not be. One of the important negative results was the proof of the unsolvability of the "halting problem", (ref. 4.4, 4.10). The halting problem is concerned with determining whether an arbitrary algorithm (or computer program) will get into an infinite loop while working on a given input. There cannot exist a computer program that solves this problem. On the other hand, the knowledge that a given problem can theoretically be solved on a computer is not sufficient to explain whether it is practical to do so. In the majority of cases it is also necessary to know more about how efficiently a given problem is solvable in a machine than merely its solvability, and also about how effectively the algorithms are converted into programs. This in turn leads to the consideration of problems from the two main areas of study: the complexity of algorithms and the correctness of programs. There are numerous problems with practical applications that can theoretically be solved (i.e. for which programs can be written), but for which the computation time and storage requirements are much too great for those programs to be of practical use. Consequently, the computation time and space requirements of a program are of great importance.

4.3.1 Computational Complexity

Computational (or algorithmic) complexity refers to the computation time and space requirements of a program. The analysis of the computation

time and space requirements of a program have become the subject of theoretical study called computational complexity, (ref. 4.25). One branch of the study of computational complexity aims at setting up a formal and somewhat abstract theory of the complexity of computable functions in which solving a problem is taken to be equivalent to computing a function from the set of inputs to the set of outputs. In this way, axioms for measures of complexity can be formulated, which are basic and general enough so that either the number of instructions executed, or the number of storage bits used by a program can be taken as the complexity measure. Using such axioms, it is possible to prove the existence of arbitrarily complex problems and of problems for which there is no best program. Thus, the study of computational complexity facilitates the writing of efficient algorithms to solve common problems and to provide the tools and principles for analysing and improving algorithms.

4.3.1.1 Average and Worst-Case Complexity

Algorithms can be evaluated by a variety of criteria. But most often, one is interested in the rate of growth of time (or space) required to solve larger and larger instances of a given problem, (ref. 4.25, 4.27). However, it is difficult to use the execution time (or word storage) as a measure of complexity because it varies with the particular computer used. One may instead count the number of instructions or statements executed by a program, but this measure too has several of the other disadvantages of the execution time measure. Such a measure is highly dependent on the programming language and on the programmer's style. Instead one needs a method that is independent of:

a) The computer used

b) The programming language

c) The programmer's style and ability

d) The implementation details.

An important observation is that the amount of work done (complexity) usually depends on the size of the input. For example, solving a system of 20 linear equations in 20 unknowns generally requires more work than solving a system of 2 linear equations in 2 unknowns. Another important observation is that even if the inputs of only one size are considered, the number of operations performed by an algorithm may depend on the particular input. For example, solving a system of 20 linear equations in 20 unknowns may not require much work if most of the coefficients are zero. Hence, the amount of work done by an algorithm cannot be described by a single number because the number of operations performed cannot be the same for all inputs. For example, the size of the input to solve a problem concerning a graph is dependent upon both the number of vertices and edges in the graph.

4.3.1.1.1 Average Time Complexity

It is not always possible to enumerate the number of operations performed by a particular algorithm on each input of size n. One possible method of solution is to determine the average behaviour of the algorithm, (ref. 4.25, 4.28), by calculating the number of operations performed for each input of size n and then take the average. Such a method may not be accurate enough in practice because some inputs may occur much more frequently than others. Instead, a weighted average is more likely to give a more meaningful estimate. Such a weighted average may be estimated as follows:

Let S_n be the set of inputs of size n for the problem under consideration. Let X be an element of S_n, and let $P[X]$ be the probability that input X occurs. Let N(X) be the number of basic operations performed by the

algorithm of input X. Then the average time complexity measure A(n) is given by

$$A(n) = \sum_{X \text{ in } S_n} P[X].N(X)$$

where

N(X) is to be estimated by careful examination and analysis of the algorithm, but P[X] cannot be estimated analytically. The function P[X] can only be determined by experience or from special information about the application for which the algorithm is to be used; it may not be easy to determine. Hence if P[X] is complicated, the estimation of A(n) is difficult. Furthermore, if P[X] depends on a particular application of the algorithm, then the function A(n) describes the average behaviour of the algorithm for only that particular application.

4.3.1.1.2 Worst-Case Time Complexity

The alternative approach to describing the implementation-independent behaviour of an algorithm is to calculate its worst case complexity, (ref. 4.29), which may be defined simply as

$$W(n) = \max_{X \text{ in } S_n} N(X)$$

where

W(n) = the maximum number of the basic operations performed by the algorithm on any input of size n.

Hence W(n) can be estimated more readily than A(n) can. W(n) is more valuable because it gives a simplified upper bound on the work done by the algorithm. Furthermore, the worst-case analysis can be used to form an estimate of the time limit for a particular implementation of an algorithm. This may be an important consideration in real-time applications.

An important observation is that the input for which an algorithm behaves worst depends on the particular algorithm, and not on the problem. Also, the concepts of worst-case and average behaviour analysis is still useful even if one chooses a different measure of work done, such as the execution time or number of passes through a loop. Hence, the observation that the amount of work done often depends on the size and properties of the input would lead to the same analysis of worst-case and average behaviour, no matter what measures were used.

The analysis of A(n) and w(n) assumes that the total number of operations performed by an algorithm is proportional to the number of the basic operations. Hence if an algorithm does N(X) basic operations for an input of size X, then the total number of operations is at most k.N(X), so that the actual execution time is k'.N(X) seconds, where k and k' are constants which depend on the algorithm and the computer on which it is implemented, but not on the input X. Consequently, if an algorithm does W(n) (or A(n)) basic operations, then it does at most k.W(n) (or k.A(n), on average) operations in total and runs in at most k'.W(n) (or, on average k'A(n)) seconds in the worst-case (or on average). In an exact comparison of two or more algorithms which are developed for a given problem and which did approximately the same amount of work, then one would need a very concise count of all the work done including book-keeping. In this case, it would be necessary to quantify not only W(n) and A(n) for the algorithms, but also the constants k and k'. However, for many problems, some algorithms have been developed that are so much better and more efficient than others that the actual values of k and k' for each of them are not very important. For example, if for one algorithm $W_1(n) = 2^n$ and for another $W_2(n) = n^2$, the latter will run

faster, in the worst case, for almost all values of the input size even if it may do more bookkeeping per basic operation than the former. Furthermore if $k_1 = 2$ and $k_2 = 15$, then so long as $n > 9$, the second algorithm does less work. This example illustrates the major difference between the combinatorial explosive nature of the enumerative scheduler algorithm and the relatively better performance of the max-flow min-cut scheduler algorithm discussed in chapter three, and plotted in fig. 3.17.

4.3.1.1.3 Space Complexity

The performance measure of an algorithm or a program in terms of its memory requirements can be analysed in a similar manner as the time complexity performance measure (ref. 4.26, 4.29). The number of memory cells used by a program, like the number of seconds required to execute the program, depends on the particular implementation. A given program will require storage space for the instructions, the constants, the variables, and the input data it uses. It may also use some work-space for manipulating the data and storing the information needed to carry out its computations. Sometimes, the input data may be representable in several abstract forms, some of which may require more space than others. In the case in which the input is represented in these various forms such as graphs, arrays, sets, or lists, then due consideration must be given for the space required for the input itself as well as any extra space needed for manipulations. For example, forming the union of two sets may require only one or two operations if the sets are represented as linked lists, but would require a larger number of operations, proportional to the number of elements in one of the sets, if they are represented as arrays and one must be copied into the other. Similarly the space requirements would be affected. If the amount of space required depends on the

particular input, then worst-case and average behaviour analysis can be employed.

4.3.2 Asymptotic Computation Complexity

With regard to the computational complexity of an algorithm, the important performance measure of interest is the rate of growth of time (or space) required to solve larger and larger instances of a given problem (ref. 4.25). One would like to associate with a problem an integer, called the size n of the problem, which is a measure of the quantity of the input data. As explained above, the time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of an algorithm. The limiting behaviour of the time complexity as the size increases is called the asymptotic time complexity. Space complexity and asymptotic space complexity are defined similarly.

It is the asymptotic complexity of an algorithm which ultimately determines the size of problems that can be solved by the algorithm. If an algorithm processes inputs of size n in time $k.n^2$ for some constant k, then the asymptotic time complexity of that algorithm is defined as $O(n^2)$, (ref. 4.30). More precisely, a function G(n) is said to be of asymptotic time complexity O(F(n)) if there exists a constant k such that

$$G(n) \leqslant k.F(n)$$

for all but a finite (possibly infinite) set of non-negative values of n. In general, the asymptotic time (or space) complexity of a given algorithm is either polynomial $O(n^k)$) or exponential ($O(k^n)$) bounded functions of the input size, n. Exponential time complexity exhibits the phenomenon of combinatorial explosion asymptotically and hence is generally inefficient. Computations based on exponentially bounded algorithms can quickly get beyond the capacity of any present or future serial computers because the

time to execute such an algorithm with a large input size can be very long indeed, even if the time to execute a single basic operation is assumed to be one microsecond. Consequently, it is the polynomial time complexity of an algorithm that is of practical interest in the solution of problems. With regard to their computational requirements, two functions $f_1(n)$ and $f_2(n)$ are said to be polynomially related if there exists polynomials $p_1(n)$ and $p_2(n)$ such that

$$f_1(n) \leq p_1(f_2(n))$$

$$f_2(n) \leq p_2(f_1(n))$$

for all values of n. The asymptotic complexity performance measure is implementation-independent.

The concept of the asymptotic time (and space) complexity of an algorithm classifies problems according to the time (and space) required to solve them. In this way "hard", i.e. very time consuming, and "not-so-hard" problems are distinguished. Polynomial bounded algorithms such as $O(n^3)$, say, can still be considered as having fairly low time requirements. However, a class of problems for which no reasonably fast algorithms have been developed exist. Many of these are optimization problems, (ref. 4.25, 4.33), that arise quite frequently in certain applications, such as

a) Job Scheduling problems

b) Graph Colouring problems

c) Bin packing problems

d) Hamiltonian circuit problems

e) Knapsack problems, etc.

None of the algorithms developed for the above problems are known to run fast, or in a reasonable time. The class P is given to those problems

that include those with reasonably efficient algorithms. An algorithm is said to be polynomial-bounded if its worst-case asymptotic complexity is bounded by a polynomial function of the input size : i.e., if there is a polynomial p such that for each input of size n the algorithm terminates after at most p(n) steps. Hence P is the class of problems that are polynomial-bounded. Although not every problem in P has an acceptably efficient algorithm, if a problem is not in P it will be very difficult and probably impossible in practice to solve. Thus, while the definition of P is too broad to provide a criterion for problems with low time requirements, it provides a useful criterion (not being in P) for problems that require too much computation time. The list of problems shown to be not in P has continued to grow over the years. To solve many of these problems, approximation or heuristic algorithms are used (ref. 4.31, 4.32). In many applications an approximate solution is good enough, especially when the time which would otherwise be required to find the optimal solution is considered. In some cases an algorithm for a complex problem may be obtained by combining several algorithms for simpler problems or subproblems (ref. 4.34). This collection of simpler algorithms may all work on the same input or some may work on the output or intermediate results of others. The complexity of the new algorithm may be bounded by addition, multiplication, and composition of the complexities of its constituent algorithms. However, since polynomials are closed under these operations, any algorithm built from a collection of several polynomial-bounded algorithms in various ways will also be polynomial-bounded.

4.4 A TIME-DELAY TRANSFORM VIEW OF COMPUTATION

In the design and analysis of computer algorithms and their programs two of the main aims are to determine whether the ultimate program code is

a) correct - i.e. does what it is supposed to do

b) up to expected performance - i.e. runs fast enough.

These two main areas of analysis can be carried out in several ways (ref. 4.35, 4.36). Correctly functioning programs can be determined by thorough testing: either mathematically or through program execution. Program performance, on the other hand, may be determined as was explained earlier, or through a transform theory of software performance (ref. 4.37). In both methods, performance estimates depend on the time-delay imposed on the CPU operations and on the frequency of execution of these operations. A model of time-delay behaviour in computer programs can be developed, and then modified to incorporate the operation execution frequencies. In this way, the time-delays and execution frequencies can be visualized as forming a network of interacting parts when used to model computer programs.

4.4.1 The Z-Domain Operations

The execution of a program segment such as a program statement or program module can be viewed as an action that imposes a time-delay on a control signal as it sequences or flows through the program. Such time-delay may be due to the time it takes the CPU to add two numbers together, or to copy a value from one memory location to another, or to compare two values and decide which operation to execute next. The total time elapsed between the beginning of execution of a program, or program module, and its termination will be the sum of the individual time-delays in each segment of the program weighted by the number of times each segment is executed. Hence, to obtain a quantitative measure of performance, a performance formula can be developed, given the estimated frequency and time-delay for each segment of the program. Such a performance formula may be derived by applying a time-delay operator to the program. This operator is related to the classical Z-transform of the feedback control theory, (ref. (4.38).

The z-transform of a time-varying function g(t) may be defined as

$$G(z) = \sum_{t=0}^{\infty} g(t).z^{t} \quad , \text{ for } |z| < 1$$

in which the time-domain is the discrete-valued semi-infinite interval $(0, \infty)$ representing the time-delays which occur during the program execution. The z-transform representation of a function is most useful when applied to arbitrary functions in order to study the behaviour of the function without knowing everything about the function in advance, such as the program performance, and represents the weighted sum of the transformed time-varying function.

As mentioned above, a program segment, such as a program statement, imposes a very small but simple unit time delay; this is just a spike at time t_o, say, in the time-domain, as shown in fig. 4.2, so that

$$G(z) = \sum_{t=0}^{\infty} g(t).z^{t} = \sum_{t=0}^{\infty} N_o.z^{t} = N_o.z^{t_o}$$

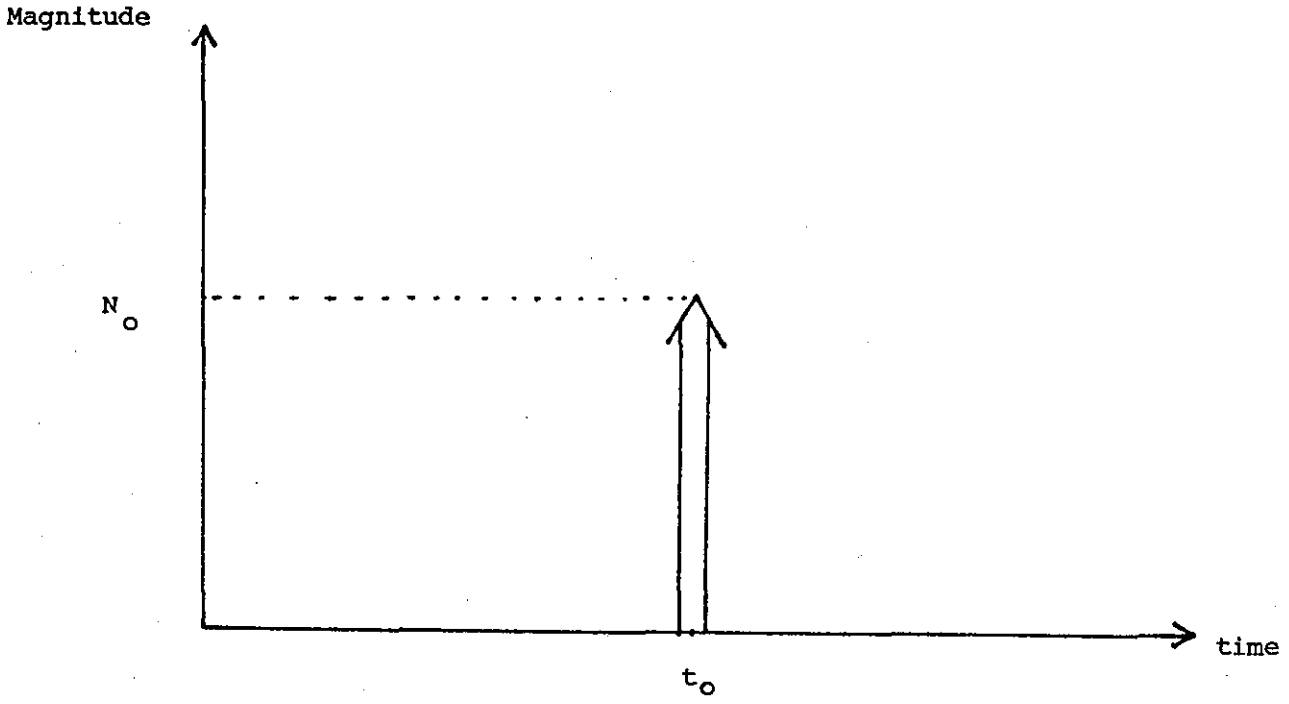
where N_o is a constant representing the magnitude or height of the spike at time $t = t_o$. For n such program segments, an ensemble of time-delays can similarly be defined, so that

$$G(z) = \sum_{t=0}^{n} g(t).z^{t} = N_o.z^{t_o} + N_1.z^{t_1} + \ldots N_{n-1}z^{t_{n-1}}$$

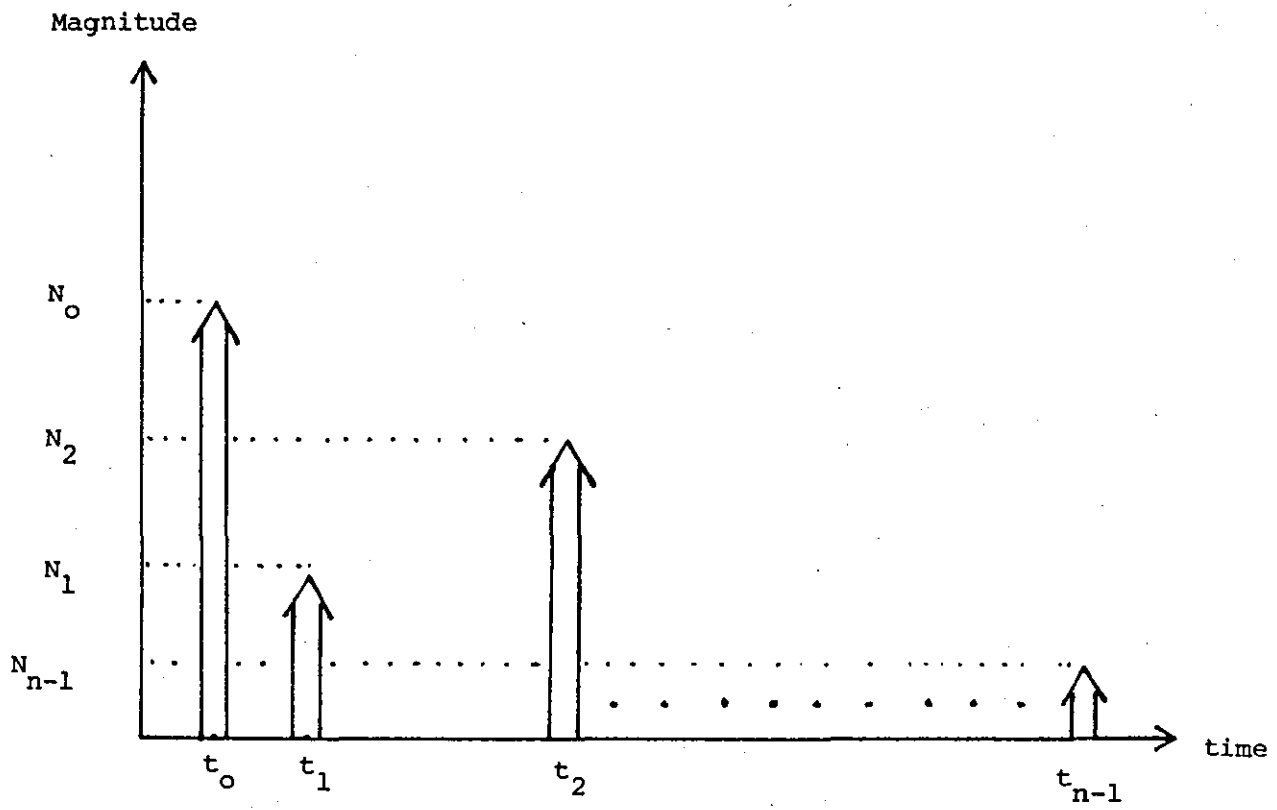
From the above, it may be deduced that

a) A time-delay in the time-domain corresponds to a multiplication by z in the z-domain.

b) The coefficients N_o, N_1, ... are magnitudes that, taken together, can be thought of as constituting the probability density function (pdf) defining the probability of a time-delay at time t.



(a) single spike



(b) An ensemble of spikes

Fig. 4.2 Unit time delay

Hence, by letting

$g(t) \equiv$ p.d.f. of the time-delay parameter t then

$$\sum_{t=0}^{\infty} g(t) = 1$$

so that, g(t) represents the instantaneous probability that a program will incur a time-delay of t time-units. From such a definition, the expected time delay τ, is given by

$$\tau = G'(1) = \sum_{t=0}^{\infty} t.g(t)$$

where $G'(1) \equiv \frac{d}{dz} G(z)\Big|_{z=1}$

In this way an ensemble of time-delays corresponding to the average execution time of the program module can be obtained. For arbitrary programs a software network model of the program can be used to determine the average execution time, (ref. 4.40)

4.4.2 Execution Time for Structured-Programs

The average execution time of a computer program can be estimated by analysing its software network (ref. 4.39, 4.40). A software network is defined as a graph G = {Nodes, Arcs, Map} containing a set of Nodes, a set of Arcs that interconnect the Nodes, and a Map function that defines the connection pattern of the Nodes via the Arcs. The labelled arcs represent the execution time of program segments, while the nodes represent the state of the program prior to the execution of a segment. The state of the program refers to the current point of control in the program's overall flow of control. Hence, a state is a place between two or more executable segments.

If t_i represents the estimated time-delay associated with a control signal propagating through arc i of a software network and p_i represents

the estimated probability of arc i being selected when the program is executed, then $G = (t_i, p_i)$ is the time-domain representation of the software network, as shown in fig. 4.3(a). The time-domain representation of the software network is transformed into the corresponding z-domain representation by relabelling the arcs from (t_i, p_i) to $(p_i.z^{t_i})$, as shown in fig. 4.3(b). Finally, the z-transform software network is further simplified and reduced into a single arc whose weight is a single expression representing the G(z) from which the overall performance of the network G'(1) can be obtained, as explained earlier. The reduction process is accomplished by a successive application of decomposition rules. Only software networks of structured programs (the structured networks) are reducible into a single final arc referred to above. Such a structured network consists of concatenated or nested structured sub-networks. In general, whether any arbitrary program which has not been written according to the structured programming rules can be converted into a structured program is yet unclear. However, such a conversion may be achieved if some redundancy is introduced in the coding, or if extra variables and control parameters are used (ref. 4.41, 4.42).

The basic structured networks are derived from the single-entry single-exit components, (ref. 4.43), and are as follows:

a) Serial execution (Concatenation)

e.g. (i) read ;

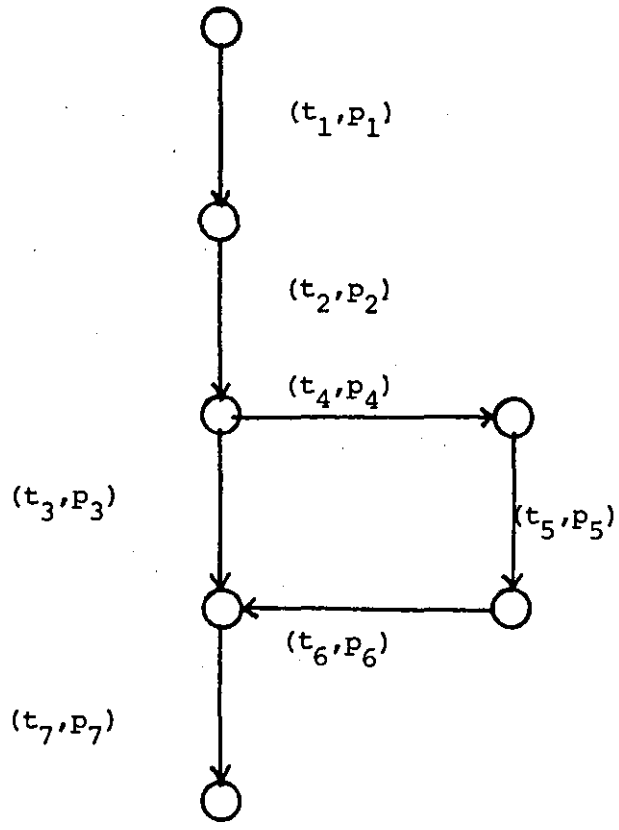
(ii) c:=a+b;

(iii) begin

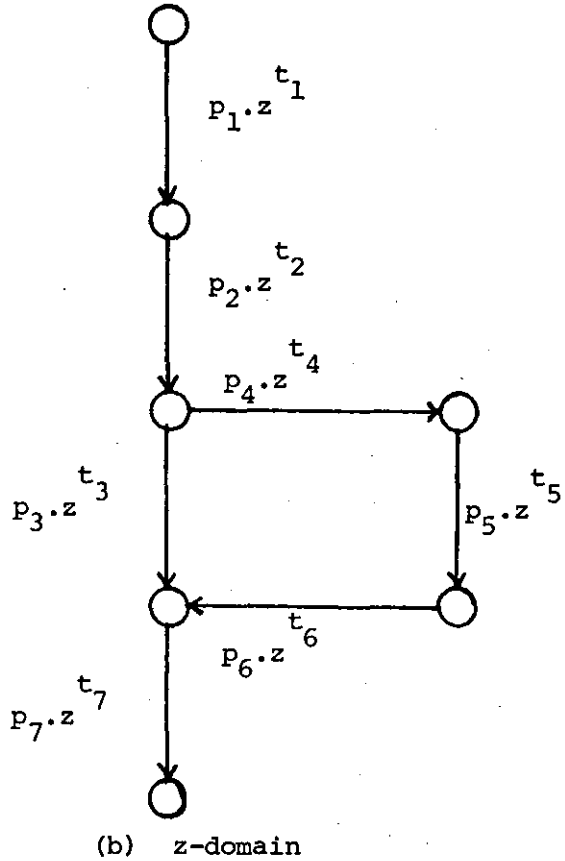
x:=a ;

y:= x^2

end;



(a) time-domain



(b) z-domain

Fig. 4.3 Software Network

b) Decision (or alternation),

e.g. (i) if x = 0

then x:= x+1;

(ii) if x>0

then y:=true

else y:=false;

(iii) case id of ... end;

c) Looping (or Iteration)

e.g. (i) repeat ..... until p;

(ii) for i:=start to finish do s

(iii) while a>b do

begin

a:= a-b ;

writeln (b)

end;

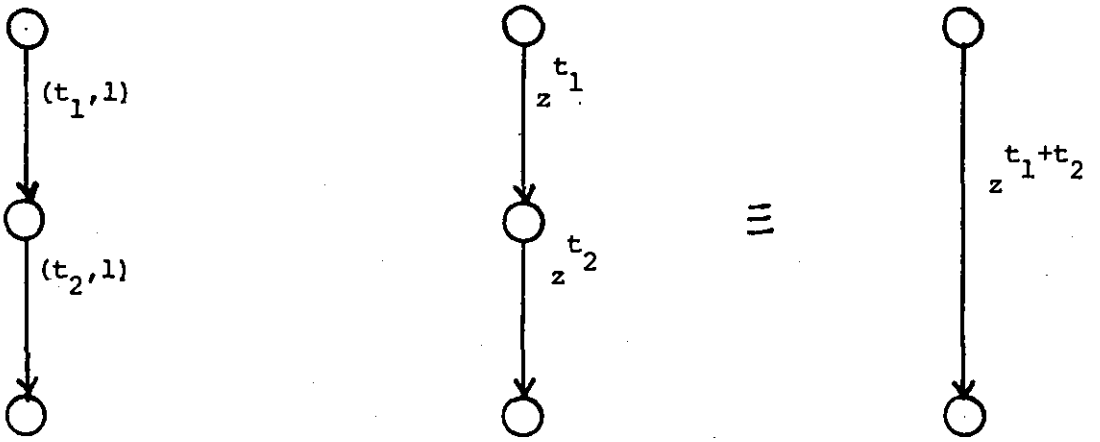
The decomposition rules for the basic structured networks is as shown in fig. 4.4. In this way the average execution time of a structured program or module may be determined.

The calculation of the average execution time of structured programs and modules can further be simplified by the use of the Petri network model of structured programs (ref. 4.40). A Petri network is a bipartite graph

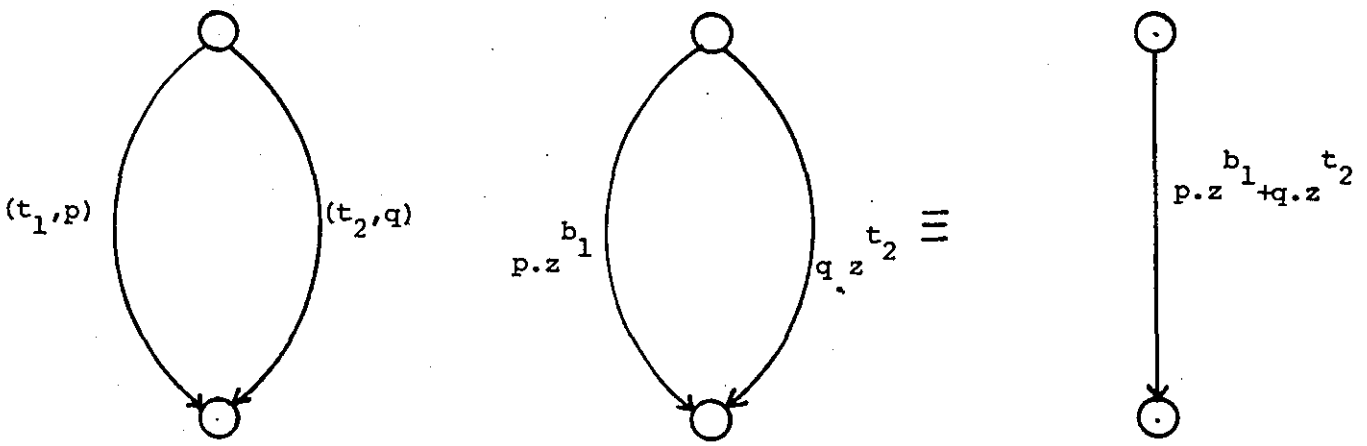
G = {Places, Transitions, Arcs, Map}

where

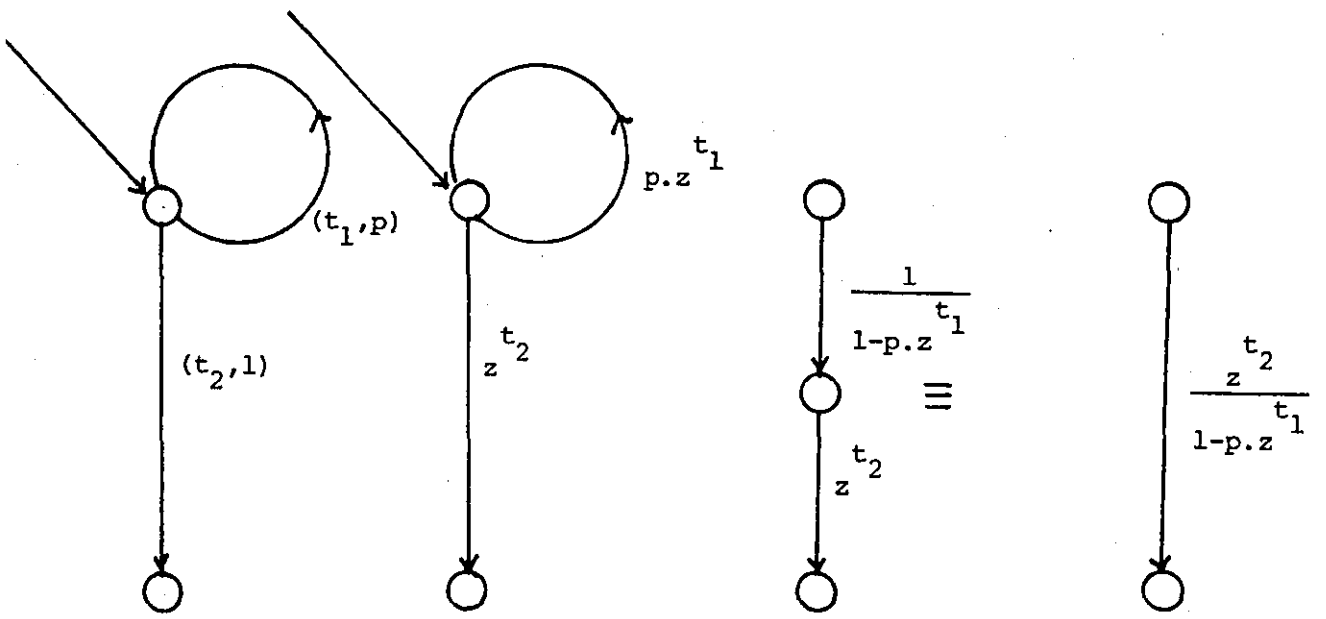
Places ≡ Set of program states representing the beginning or end of an action, e.g. begin, end, repeat, until, if, then, else, while, for, etc.



(a) Concatenation



(b) Decision



(c) Iteration

Fig.4.4 Reduction Rules

Transitions ≡ Set of program actions representing operations, statements, program segments, etc.

Arcs ≡ set of control paths representing the flow of control from places to transitions and from transitions to places.

Map ≡ The topology of the graph is determined by the connectivity of places and transitions. Arcs only connect places to transitions and transitions to places. Arcs are not allowed to connect places to places or transitions to transitions.

A graph $G = \{X, E\}$ is said to be a bipartite graph if its node set X can be partitioned into two subsets X_1 and X_2 such that every arc of G has one endpoint in X_1 and one endpoint in X_2.

A structured program is composed of concatenated and nested D-structured components. The D-structured components have single-entry single-exit control structures as explained above. The Petri networks of D-structured components simplify the analysis of structured programs. Fig. 4.5 shows the format of a Petri network and figs. 4.6, 4.7 and 4.8 show the D-structured components in Petri network notation. In fig. 4.5 it can be seen that the flow through a network can be split by a forked transition and merged by a place. These operations correspond to the flow of control in a program or module. The places correspond to control statements of a program, and the transitions correspond to executable operations performed by a program or module.

The concatenation and nesting (looping) rules are the only composition rules of structured programming (ref. 4.43). These two rules guarantee programs that are reducible to a single-entry single-exit (or single transition) structure. This reducibility facilitates the analysis of any structured programs using its network model and the z-transform

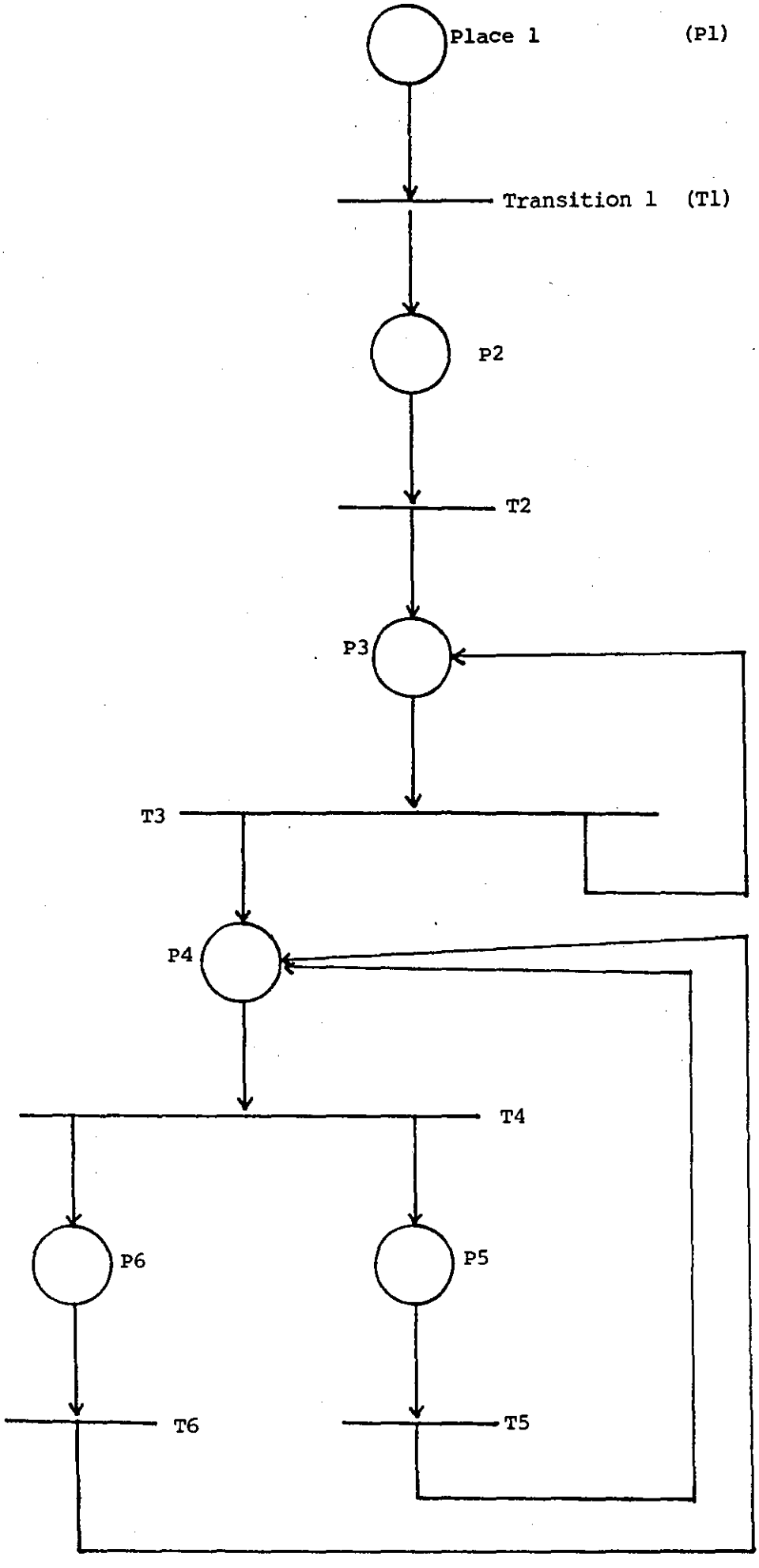


Fig. 4.5 Petri Network

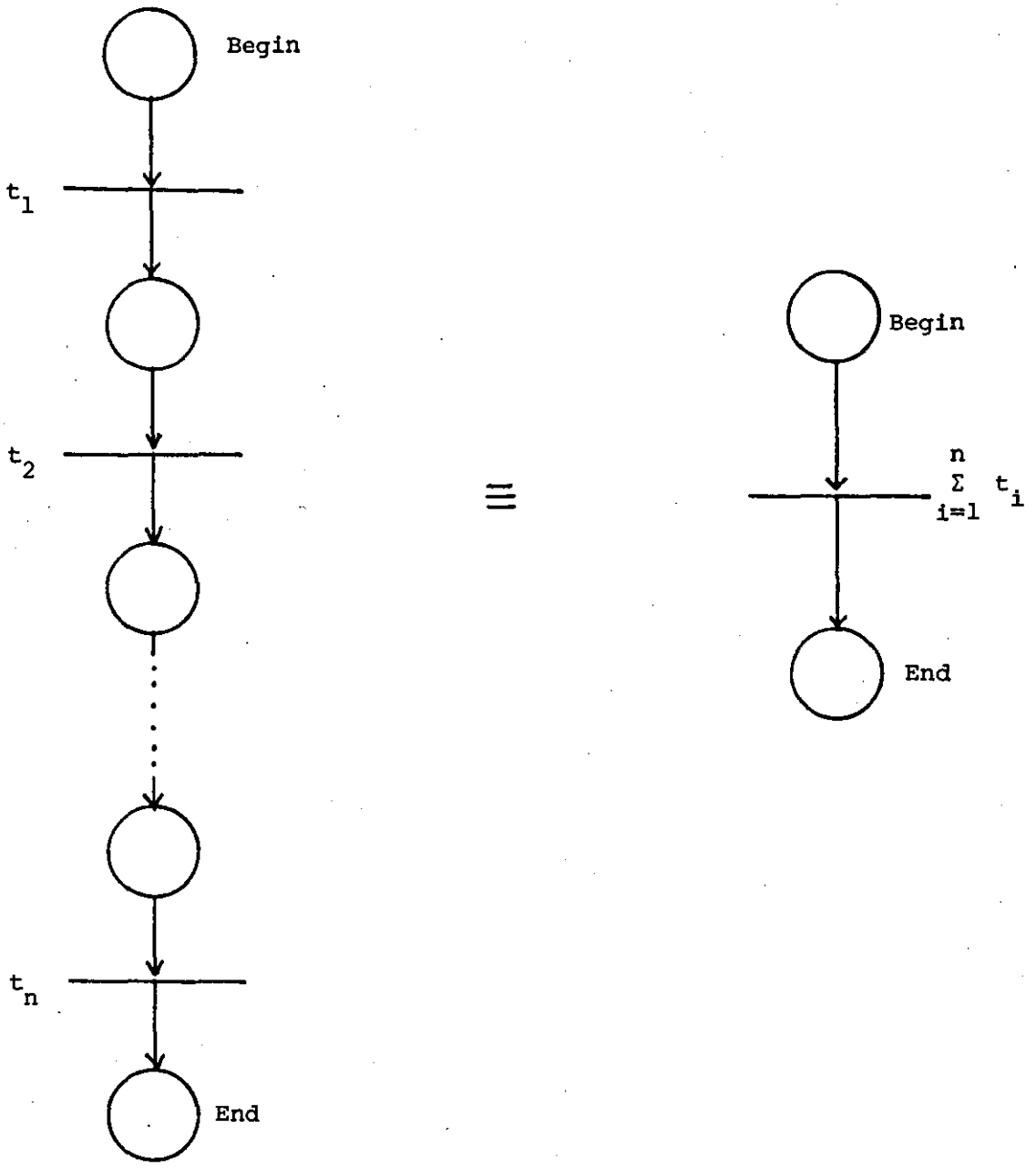
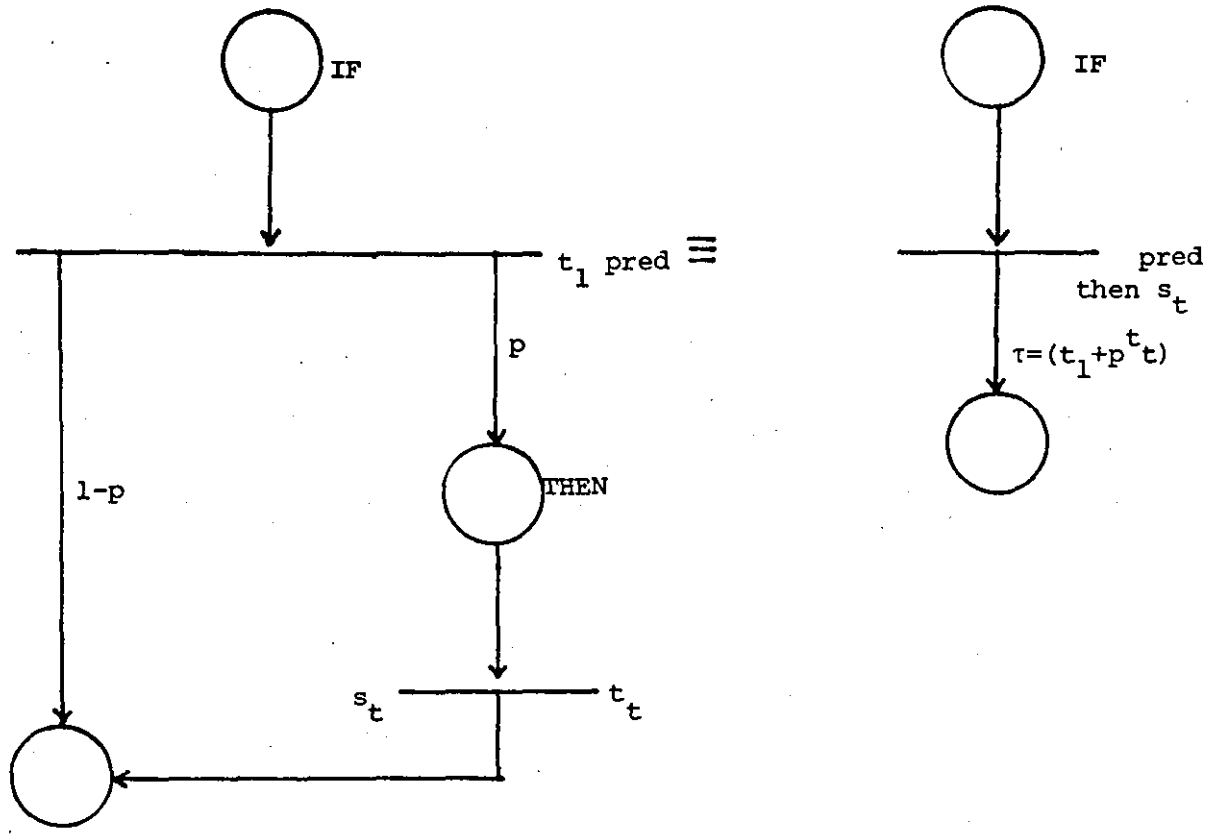
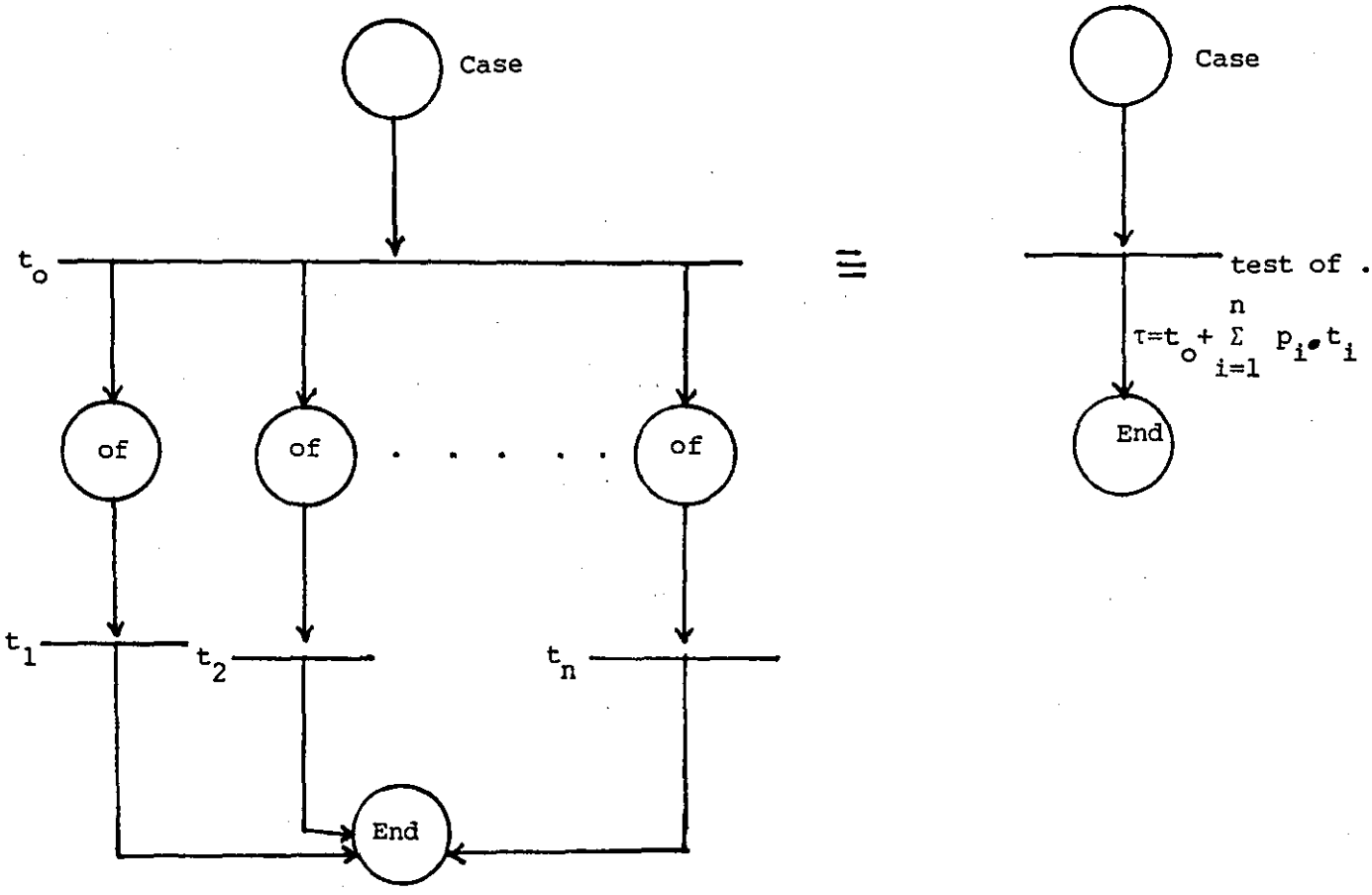


Fig. 4.6 Concatenation Decomposition Rule

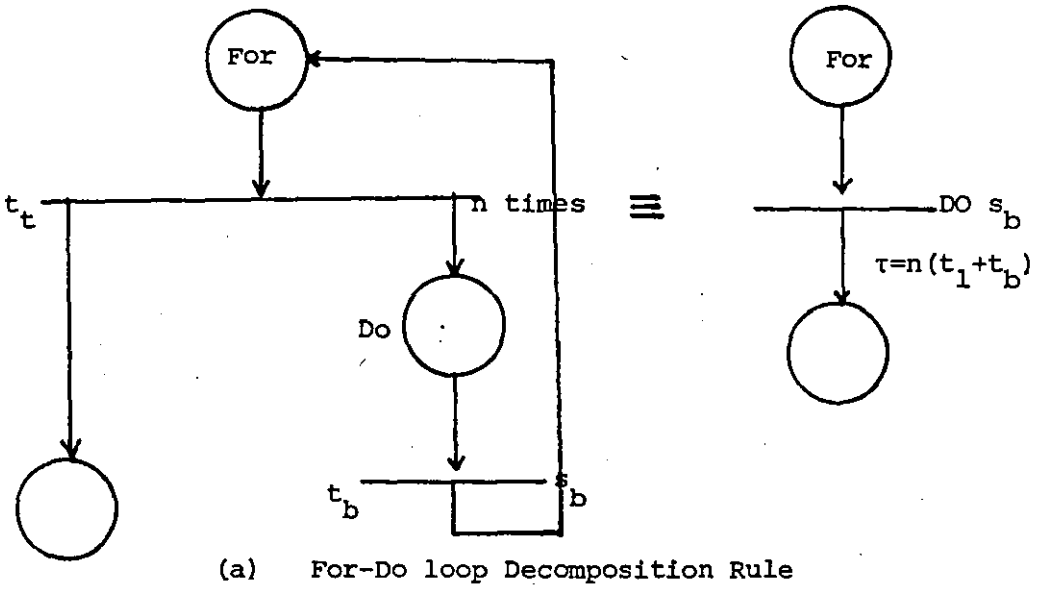


(a) If-then Decomposition Rule

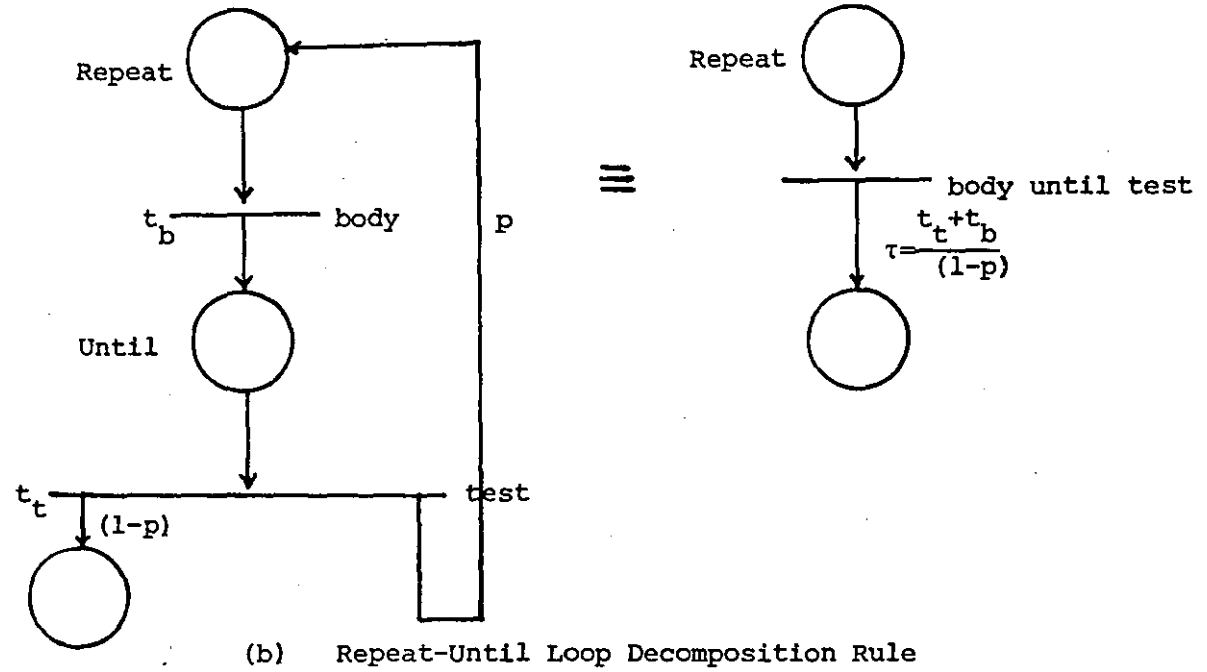


(b) Case-of Decomposition Rule

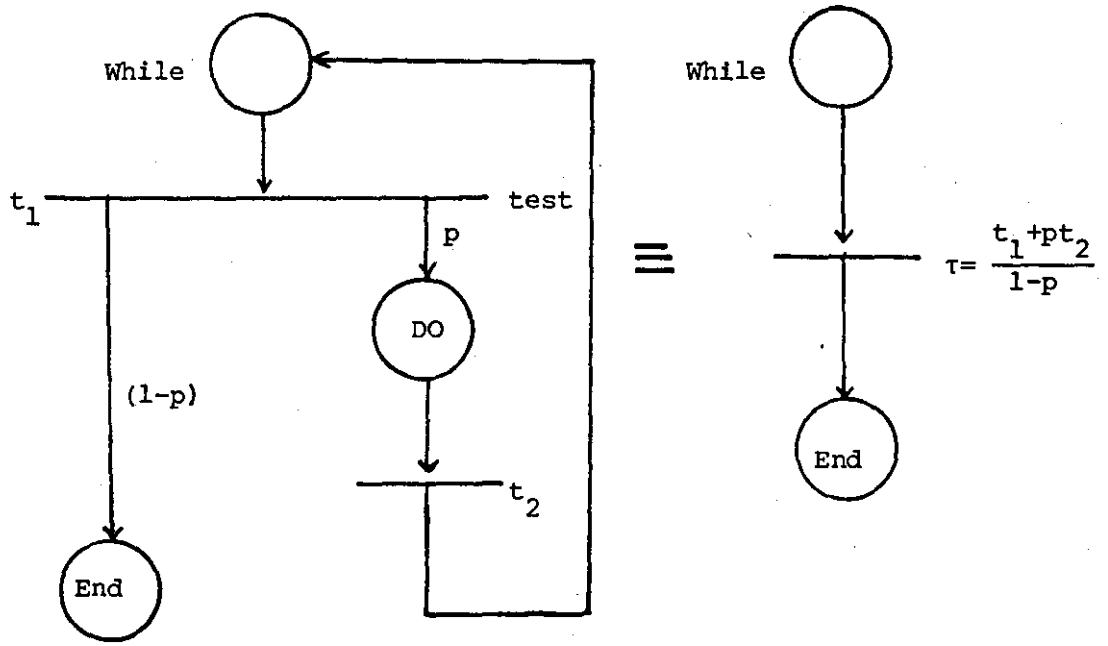
Fig.4.7 Decision Decomposition Rules



(a) For-Do loop Decomposition Rule



(b) Repeat-Until Loop Decomposition Rule



(c) While-Do Loop Decomposition Rule

Fig. 4.8 Iteration Decomposition Rules

performance formulae. The flow of control in programs and modules may be concatenated and nested to any arbitrary depth. Many of the time-consuming programs in scientific computation often involve repeated evaluation of the same function on different argument sets.

4.5 PDF CHARACTERISATION OF COMPUTATION

So far it has been explained and shown how a performance formula can be obtained and used to estimate the average or maximum processing time of a program or module. But if such a method was to be employed in a distributed computation environment for every program or module which is a candidate for distributed computation, the work and time involved in obtaining the performance formula for every program or module would be overwhelming. Hence simplified representations which will preclude tractable solutions are preferable.

In modelling many resources of a computer system, one may be primarily interested in the service times (computation times) of programs or program modules which use the resources. In this context, the program's service time will consist of the execution of the instructions and the amount of time spent will be determined largely by the particular instruction mix executed, the CPU times for those instructions, the I/O requirements and memory management, if the system has virtual memory. If the resource is a moving head disk, for example, then a program's service time will consist of a positioning time and a transfer time, and the total time used will be determined largely by the distance the arm must move, the mechanical speed, and the total amount of information transferred. All of these determining factors are measurable (or observable) and in a sense deterministic But such an approach might be impractical. It may be found more appropriate

and more practical to characterise service times as random phenomena. For CPU times, the instruction paths will usually depend heavily on unpredictable data (ref. 4.44, 4.45). In a virtual memory system, the CPU times will also depend on memory management routines which may, in turn, depend very heavily on the behaviour of the entire multiprogramming set of programs (ref. 4.45). Hence, further simplifications may be found necessary.

One simplification is to represent the processing times by probability distribution functions (PDF) - such as, the probability distribution for CPU time used between I/O operations, the probability distribution for the times between scheduler activations, or the probability distribution for the times between page faults; and even then to assume that successive timings are independent with the respective distributions (ref. 4.49, 4.50). In this way, arbitrary probability distributions may be defined and used for the CPU service times. But even when arbitrary probability distributions are defined and used for the service times, only the first and second moments are of the greatest importance (ref. 4.51). These moments are defined as follows:
Let X be a random variable and $P[X]$ be the probability of X, then the mean value, or the first moment of X, when X is defined on a discrete sample space, is defined as

$$E[X] = \sum_{x} X.P[X]$$

and the n^{th} moment is similarly defined as

$$E[X^n] = \sum_{x} X^n.P[X]$$

In general, the n^{th} central moment, (the mean value of the n^{th} power of the difference between the service time and the mean value), is defined

as

$$E[(X-E[X])^n] = \sum_x (X-E[X])^n . P[X]$$

The first central moment is identically equal to zero. The second central moment is the variance $V[X]$, and the square of the variance is the standard deviation, σ_x. The mean represents the service time and the variance gives an indication of variability. A more direct measure of this variability is called the coefficient of variation C_x, which is given by

$$C_x = \frac{\sqrt{V[X]}}{E[X]} = \frac{\sigma_x}{E[X]}$$

Processor service time distributions in general tend to be highly variable and values of C_x of ten or more are not unusual (ref. 4.45). On the other hand I/O service time distributions tend to be much less variable and values of C_x much less than one are typical, (ref. 4.49).

Similar definitions for the mean value, moments and variance apply equally well when X is defined on a cntinuous sample space in which case the corresponding PDF are continuous. If $f_x(x)$ is the pdf (probability density function) of the random variable x, then

$$E[X] = \int_{-\infty}^{\infty} x . f_x(x) . dx$$

and

$$E[X^n] = \int_{-\infty}^{\infty} x^n . f_x(x) . dx$$

and

$$V[X] = \int_{-\infty}^{\infty} (x - E[X])^2 . f_x(x) . dx$$

Arbitrary distributions can be defined thus, and from them the moments extracted.

4.6 MARKOVIAN CHARACTERISATION OF COMPUTATION

It was shown above how arbitrary PDFs may be defined and used to characterise a program's performance measure such as the average service time and service time variability. However, if arbitrary PDFs for service times are used, once is still left overwhelmed with information and the mathematical model becomes very difficult to characterise and solve. For example, suppose one wishes to represent the time until completion for a processing request arriving at the CPU when the CPU is already busy processing a previous request. If one wishes to estimate the probability distribution for this period of time, then one will need to determine the distribution for the sum of the time for the request plus the time until the CPU is given the request. This latter time will depend on the time already spent on the work ahead of the arriving request. From this, it can be seen that the solution becomes very involved and difficult so that one can only hope for a solution under very restricted conditions. Such a method of solution together with the inherent difficulties appear to result into a paradox: on the one hand one wishes to solve for the time dependent behaviour of the computing environment but at the same time, on the other hand, one is not free to consider time in the representation of the system. However, the Markov process representation of the computing system allows the time to be considered in a very controlled manner and thus overcome the apparent paradox.

Markov processes are extremely powerful tools which can be used to provide accurate, yet mathematically tractable, models of computing systems performance (ref. 4.52, 4.53). Performance models are often used to estimate the performance of computing systems over a period of time.

This time period may be explicit for some performance measures and implicit for others. The two most important performance measures, throughput and response time, represent explicit and implicit time periods, respectively. Throughput is measured in the amount of work (e.g. the number of programs) handled during a time period. Though one might wish to estimate the response time for an individual command (or the turnaround time for an individual module), usually one will have to be content with an estimate of the mean or some other measure of the response time distribution.

4.6.1 The Markov Process

A Markov process with a discrete state space is referred to as the Markov chain, (ref. 4.54). The discrete-time Markov chain is defined as follows:

> A set of random variables $\{X_n\}$ forms a Markov chain if the probability that the next value is x_{n+1} depends only upon the current value x_n and not upon any previous values. Expressed analytically, the Markov property may be written as
>
> $$P[X(t_{n+1}) = x_{n+1} | X(t_n = x_n, \ldots . x(t_1) = x_1] =$$
>
> $$P[X(t_{n+1}) = x_{n+1} | X(t_n) = x_n]$$

This represents a random sequence in which the time dependency extends backward only one unit in time. Hence, the way in which the entire past history affects the future of the process is completely summarised in the current value of the process. Because of this, one is not free to require that a specification of the random variable, that describes how long the process remains in its current state before making a transition to some other state, also be given as to how long the process has been in its current state. This imposes a heavy constraint on the distribution of time that the process may remain in a given state. In fact this

state time must be exponentially distributed. Thus the exponential distribution is the continuous distribution which is memoryless. This memorylessness is what makes the exponential distribution so important in mathematical models for analysing computer systems and communication networks. Similarly, in the discrete-time Markov chain, the process may remain in the given state for a length of time that must be geometrically distributed. The geometric distribution is the only discrete probability mass function that is memoryless. The consideration of Markov processes is central to the study of queueing theory.

4.6.2 The Exponential Distribution

The key to the Markov process representation is the negative exponential distribution (ref. 4.54). The negative exponential pdf with rate μ is defined as

$$f_x(x) = \begin{cases} 0 & , x < 0 \\ \mu . e^{-\mu x} & , x \geq 0 \end{cases}$$

and the corresponding PDF as

$$F_x(x) = \begin{cases} 0 & , x < 0 \\ 1-e^{-\mu x} & , x \geq 0 \end{cases}$$

from which $E[X] = \frac{1}{\mu}$, $E[x^2] = \frac{2}{\mu^2}$, $V[X] = \frac{1}{\mu^2}$ and $C_x = 1$.

The main importance of the negative exponential distribution comes from its memorylessness. The memoryless property is that if a random variable is known to have the exponential distribution and that the value of the random variable is at least some other value, then the distribution for the remaining value of the variable has the same exponential distribution as the total value. In this connection, if the CPU times of a

program between I/O activities are exponentially distributed and that the CPU service offered so far is k seconds, say, then the remainder of the current CPU time will have the same exponential distribution as the total CPU time.

Service time distributions are often categorised by their variability, relative to the exponential distribution. A class of distributions with greater variability than the exponential is known as the hyperexponential Similarly, the distributions with less variability than the exponential are known as the hypoexponential.

4.6.3 Exponential Service Stages

Timings in computer systems do not always follow the negative exponential distribution. But combinations of exponential service stages can be used to approximate closer the actual service time distributions. Hyperexponential and hypoexponential distributions can be thus derived, (ref. 4.54). The method of exponential stages is both general and compatible with Markov processes because the only memory introduced is the distribution stage, and this additional memory is accommodated by refining the state definition.

4.6.4 The Poisson Process

The Poisson process is important in modelling many important processes in a computer system or a communication network (ref. 4.54). If the times between events in a stream of events are independent and the durations of the inter-event times have the negative exponential distribution, the events can be shown to form a Poisson process, (ref. 4.51). Such events could be the completion of service at a CPU, when the CPU is busy processing or an arrival to the CPU for processing. The two important properties of a Poisson process are:

a) Occurrences of events during non-overlapping intervals of time are independent

b) For a sufficiently small interval of time, Δt, the probability of no events occurring during the interval is $1-\mu.\Delta t$, the probability of one event occurring during the interval is $\mu.\Delta t$, and the probability of more than one event occurring during the interval is negligibly small.

The first property gives the Poisson process a memoryless property and is equivalent to : events form a Poisson process and the inter-event times are independent with identical exponential distributions.

CHAPTER 5

LAN DELAY PERFORMANCE

5.1 INTRODUCTION

In a distributed computation environment the total system delay performance is an important design factor. One of the main components of delay in such an environment is due to the scheduling time of modules as has already been explained in earlier chapters. The other two major components of delay are mainly due to the communications within the LAN communications subnet and the computational delay at the sink processor due to an increase of the computational workload there. This chapter examines the communications component of the total delay experienced by the module and the intermodule data packets across the LAN communications subnet interface.

A major aspect of most modern computer communications systems of the type examined in this thesis is the sharing of resources. Some of the main types of resources shared in such a system are

a) Communications capacity

b) Storage capacity

c) Processing capacity

However, many of the issues involved in the consideration of these computer communication networks deal mainly with an equitable allocation of these finite-capacity resources among the competitive demands for the resources. These competitive demands for the resources almost always lead to conflicts and hence a means of allocating the resources in a manner that resolves this conflict is an important system design objective for a smooth system operation. The main result of such competitive conflicting demands for the resources is the time delay involved before the resource

is finally allocated. In most cases, these competitive conflicting demands arrive in an unpredictable fashion. Furthermore, the size of these demands made upon the finite-capacity resources is also often unpredictable. But, in a well designed system the resultant gains due to the sharing more than compensate for the losses due to the conflicts. If there are no conflicts for the resources, then performance analysis is relatively straightforward. Hence, the unpredictable contention for the resources often leads to the consideration of the system as a network of queues or of queueing networks. In this way, both the computational and communications models which principally examine these queues associated with the resources and the interaction between the resources and their queues can be formulated and examined. By the use of such models, it may be possible to examine some important basic performance measures of the resource sharing system, such as the

a) Resource utilization

b) Resource capacity

c) Resource cost

d) Resource efficiency

e) System response time and delay

Also, the important relationships and trade-off among the various performance measures can be examined. For example, the consideration of the resource efficiency (or throughput) may lead to the fomulation of better or more efficient queueing disciplines or channel access protocols which in turn may lead to an overall reduction of the system response time. However, low delay performance is an important objective in computer communication and computation environments.

In the LAN communications subnet environment, the shared resource is the common broadcast transmission channel. As explained in an earlier chapter, various channel acquisition protocols can be employed within the LAN communications subnet. But in a bursty communications environment such as is supported by computer communication networks, random channel acquisition protocols can be employed (ref. 5.1, 5.2), and CSMA-CD protocols have been employed (ref. 5.3, 5.4, 5.5, 5.6).

5.2 RANDOM CHANNEL ACCESS PERFORMANCE

Random channels are characterised by the existence of many uncoordinated network-users sharing a common communications channel such as a coaxial cable network. As was explained in an earlier chapter, ALOHA networks (ref. 5.1, 5.2), CSMA and CSMA-CD networks (ref. 5.3, 5.5), fall in this category. In ALOHA-like channel resource sharing networks there is no predictable or scheduled time for any ready-user to transmit and the ready-users broadcast their transmissions at random. Whenever a network-user has a packet ready for transmission, the user just sends it without any regard to the state of the channel. After transmitting the packet, the user must wait for an acknowledgement for a length of time equal to the maximum possible round-trip propagation delay on the network. If an acknowledgement is not forthcoming during that period of time, the packet is assumed to have been destroyed through a collision and so the user must retransmit the packet again. This process is repeated until the packets safe arrival is acknowledged. The main motivation for considering the ALOHA-like behaviour of random channels is that it gives the minimum or lower-bound performance measure of such channels. Throughput and channel delay performance for both pure and slotted ALOHA-like

channels have been investigated (ref. 5.6, 5.7), and have been the basis for further improvement of channel acquisition protocols (ref. 5.8, 5.9). On the other hand, the maximum or upper-bound channel throughput and delay performance measures are found to be dependent on the channel propagation delay.

5.2.1 Channel Propagation Delay

The value of the propagation delay plays a major role in the overall performance of the transmission channel (ref. 5.13). It is also a dominant factor in characterising or distinguishing the various network types into long-haul, local networks and multiprocessor systems. The importance of the propagation delay on channel performance can be characterised by the propagation delay parameter α which may be defined as

$$\alpha = \frac{\text{propagation delay}}{\text{data transmission time}}$$

or

$$\alpha = \frac{(\ell/V)}{(P/C)}$$

where

ℓ = maximum length of the communication channel

V = propagation speed in the communication channel,

(approx. 2×10^5 km/sec)

P = packet size in bits

C = peak transmission capacity (bit rate) in bits per second.

In slotted channels, the value of α is simply

$$\alpha = \frac{\tau}{b_1}$$

where

τ = maximum end-to-end propagation or half the slot size

b_1 = average packet transmission time.

The value of α is also important in giving an indication of an upper bound on the channel utilization U. The variation of U and α can be estimated, since

$$U = \frac{\text{throughput}}{\text{Data transmission rate}}$$

$$= \frac{P/(\text{ propagation delay + transmission time })}{\text{Data transmission rate}}$$

$$= \frac{P/(\ell/V + P/C)}{C}$$

$$= \frac{1}{1 + \alpha}$$

From this relationship it may be seen that high values of channel utilization may be achieved by using longer packets.

5.2.2 Throughput Performance of Random Channels

The lower-bound limiting throughput performance of the ALOHA-like random channels have been derived (ref.5.1, 5.2, 5.6). These results are derived by assuming that the population of the network-users collectively forms an infinite source for the packets, and that this input source forms an independent Poisson process. In a Poisson process with rate λ the probability that k packets are generated during a time period t is given by the Poisson distribution (ref. 5.10, 5.11),

$$P(k) = \frac{\lambda^k . e^{-\lambda t}}{k!}$$

Such an assumption facilitates analytically tractable equations whose results are found to agree reasonably well with the practical results. Furthermore, such an infinite population model is found to closely approximate a finite population model with about 50 or more users (ref. 5.12). Also, packets are assumed to be of constant length and that the channel is noise-free. In some cases, an exponential distribution for packet

sizes is found to give acceptable analytical results, (ref. 5.14). Hence if the total traffic input (offered load) G to the communication subnet is assumed to be an independent Poisson process generated by an infinite population of network-users, the channel throughput S in a pure ALOHA-like environment can be calculated by considering the average traffic on the channel due to both the newly generated and the previously collided packets waiting for channel reacquisition and retransmission. Hence,

Average offered traffic (packets/sec) = Average Carried Traffic (packets/sec) + Average retransmitted traffic (packets/sec)

or, normalising with respect to the average packet transmission time, then

$$G = S + G.P(\text{packet involved in collision})$$

Since G is assumed to be generated from an independent Poisson process, then

$$G = S + G.(1 - e^{-2(1+\alpha).G})$$

from which

$$S = G.e^{-2(1+\alpha)\cdot G}$$

where α = propagation parameter.

If $\alpha << 1$, then

$$S \doteq G.e^{-2G}$$

Similarly, for the slotted ALOHA-like channels (ref.5.2),

$$S = G.e^{-(1+\alpha).G}$$

or

$$S = G.e^{-G} \text{ if } \alpha << 1.$$

The improved throughput performance of the slotted channel can be attributed to the vulnerability of the packets to collision, as shown in

fig. 5.1. Fig. 5.2 shows the variation of the channel throughput S with G for α = 0.0001. From these results it can be seen that the maximum throughput attainable in both the unslotted and slotted ALOHA-like channels is $S = \frac{0.5}{e(1+\alpha)}$ and $S = \frac{1}{e(1+\alpha)}$ respectively, (ref. 5.2). It can be seen that these values for channel throughput are very low and hence better channel acquisition protocols are necessary. The CSMA and CSMA-CD protocols can be used to achieve higher values of S, (ref. 5.8, 5.9).

5.2.3 Delay Performance for Random Channels

As explained earlier, the delay performance of the LAN communications subnet plays a critical role in a distributed computation environment and low values of delay are necessary. In the simple case in which the network-users just transmit their packets in ALOHA-like fashion, the delay performance can be estimated by considering all the major factors contributing to the delay. The three main components of delay are due to the:

a) queueing delay

b) propagation delay

c) transmission delay

Even though the queueing delay for the newly generated packets is zero, the dominant component of delay in the ALOHA-like channel acquisition and transmission systems is due to the retransmission delay following a collision. This can be estimated by first calculating the average number of retransmissions per packet transmission time. This value is just $\frac{G}{S} - 1$, so that the total delay D is given by

$$D = \left[\frac{G}{S} - 1\right).\Delta+\alpha+1\right].b_1$$

where Δ = average normalized delay for one transmission.

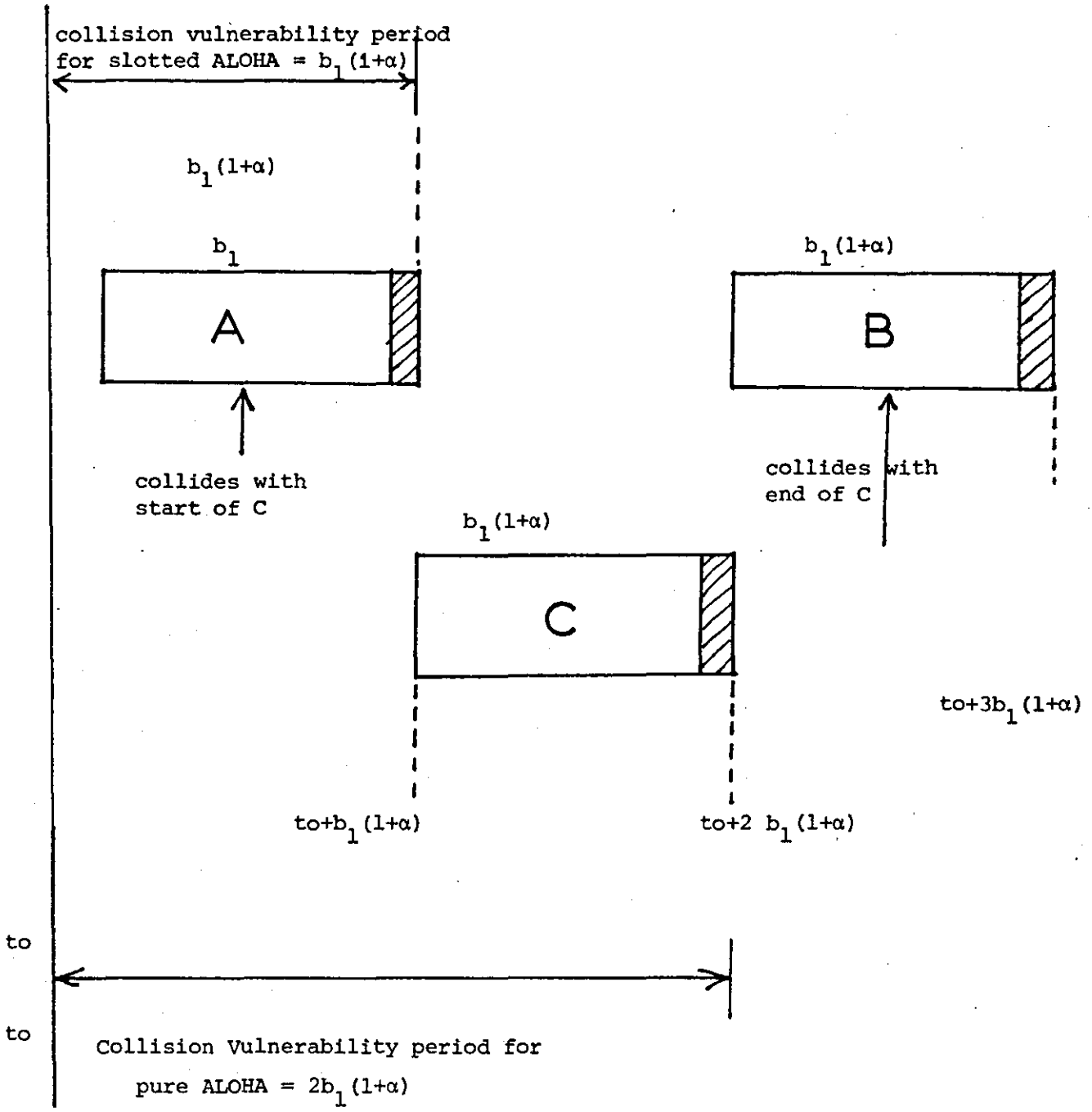


Fig. 5.1 Collision Vulnerability Period for Packet C

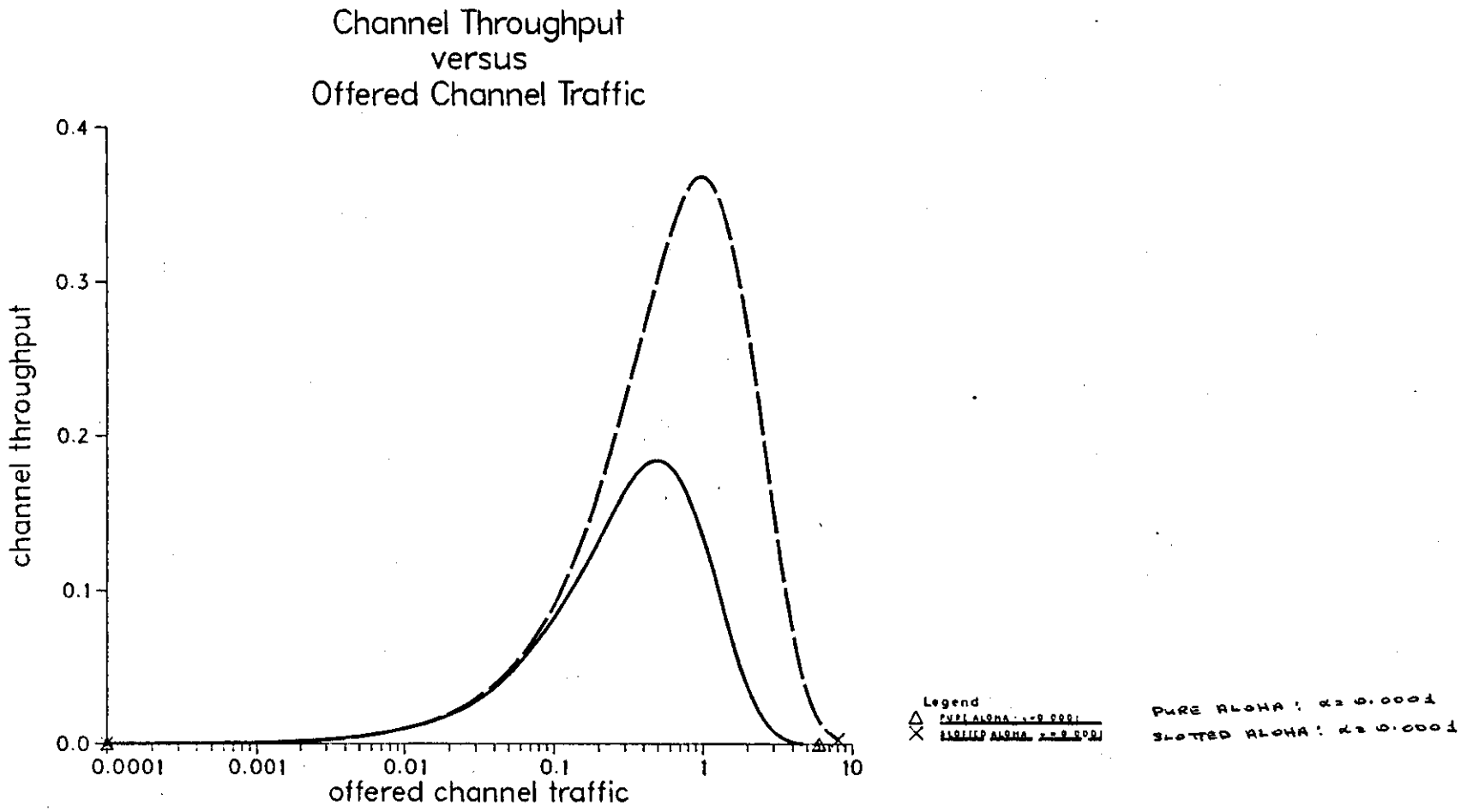


Fig. 5.2 Channel Throughput

The value of the average delay for one transmission depends on the collision arbitration and retransmission algorithm that must be observed by the colliding users and a uniform distribution of from 1 to R packet-retransmission times has been used as it is found to minimize the number of repeated collisions, (ref. 5.12). From this consideration

$$\Delta = \frac{R+1}{2} + 1 + 2\alpha + g$$

where g = time for receiver to generate the acknowledgement, so that

$$\frac{D}{b_1} = \left[e^{2(1+\alpha).G} - 1\right](1 + 2\alpha + g + \frac{R+1}{2}) + \alpha + 1$$

for the unslotted ALOHA-like channels, and

$$\frac{D}{b_1} = \left[e^{(1+\alpha).G} - 1\right](1 + 2\alpha + g + \frac{R+1}{2}) + \frac{3}{2}(\alpha+1)$$

for the slotted ALOHA-like channel. Fig. 5.3 shows how the values of $\frac{D}{b_1}$ varies with G for both types of channels and for R = 10 and R = 100. These results show that the delay performance for such channels is both poor and may exhibit instabilities, (ref. 5.3, 5.15). Furthermore, figs. 5.2 and 5.3 show that a definite trade-off exists between S and D, so that the required low delay performance cannot be achieved simultaneously with the desirable high values of throughput.

5.3 CSMA-CD BASED PERFORMANCE

A consequence of bursty transmission channel traffic in computer communication environments is that among a large population of network-users, only a small number of them have any data to transmit, at any one time. These constitute the ready-users. In such an environment,

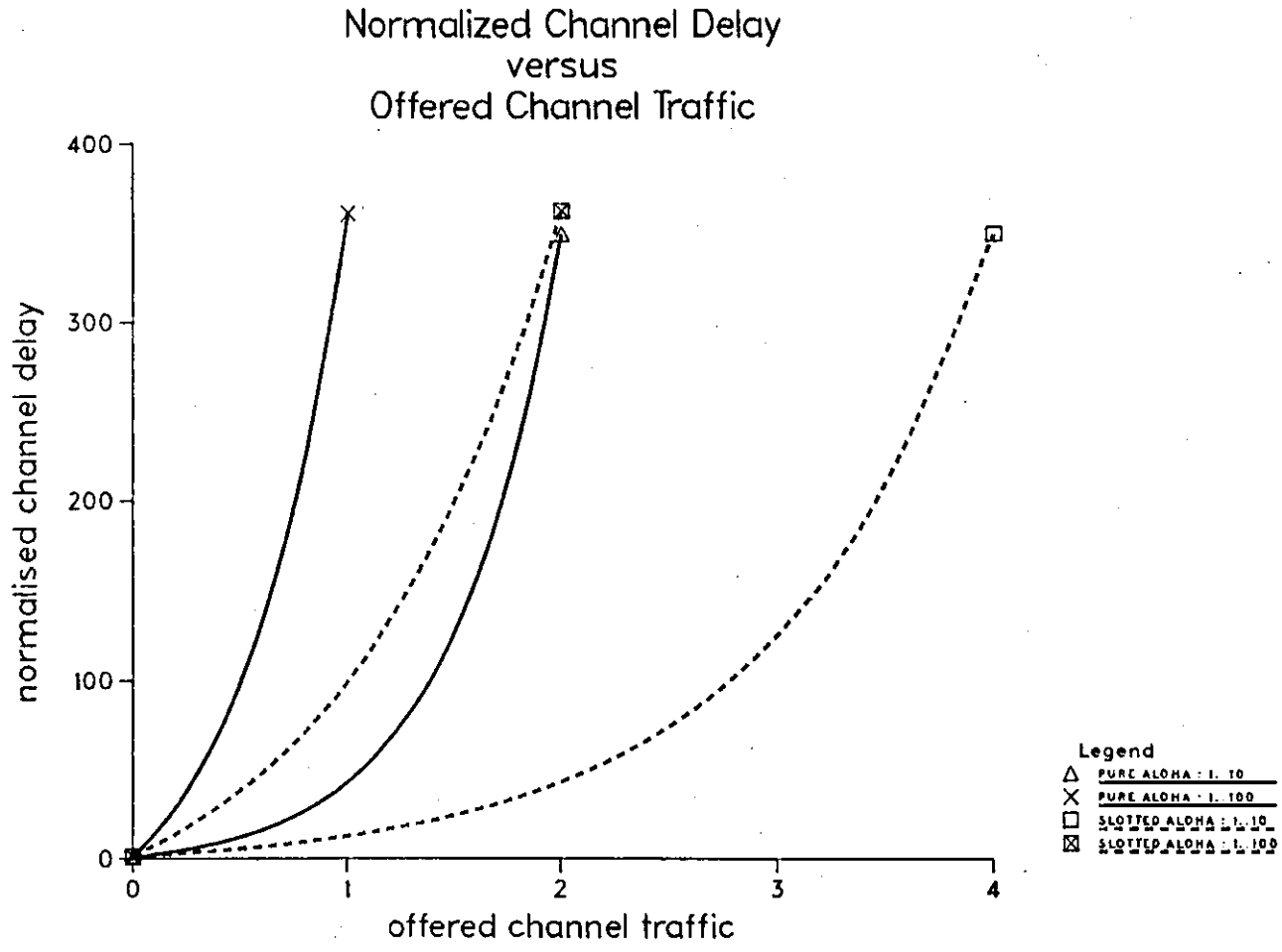


Fig. 5.3 Normalized Channel Delay

the performance of an access protocol for a broadcast-type network depends mainly on how quickly any one of the ready-users can be identified and given sole access to the multi-shared communication channel resource. A carrier sense multiple access with collision detection (CSMA-CD) channel protocol requires that collisions in the channel be detected and that all the users involved in the collision abort their transmissions quickly (ref. 5.7, 5.8, 5.9). In addition, an adaptive random retransmission algorithm is required to ensure a stable channel (ref. 5.15).

In almost all CSMA-CD environments, network-users are assumed to be time synchronised so that, following each successful transmission, the channel is slotted in time (ref. 5.2, 5.16). In addition, the users can only start transmission at the beginning of a time slot. In order to implement the collision abort and the retransmission contention algorithm, the minimum duration of a time slot is $T = 2\tau$, where τ is the maximum end-to-end propagation delay. Hence, within a time slot, if a collision is detected and the colliding transmissions are aborted immediately, the channel can be assumed to be free of any transmissions at the beginning of the next time slot.

5.3.1 Heavy Traffic Performance Channel Model

A simplified model for the performance of a loaded channel can be examined by considering alternating channel time periods (ref. 5.7, 5.16). These alternating time periods can be identified as either the transmission period during which the channel has been acquired for a successful packet transmission, or the contention period during which the ready users attempt to acquire control of the channel, as shown in fig. 5.4. As mentioned earlier, the ready-users must defer to the passing traffic before

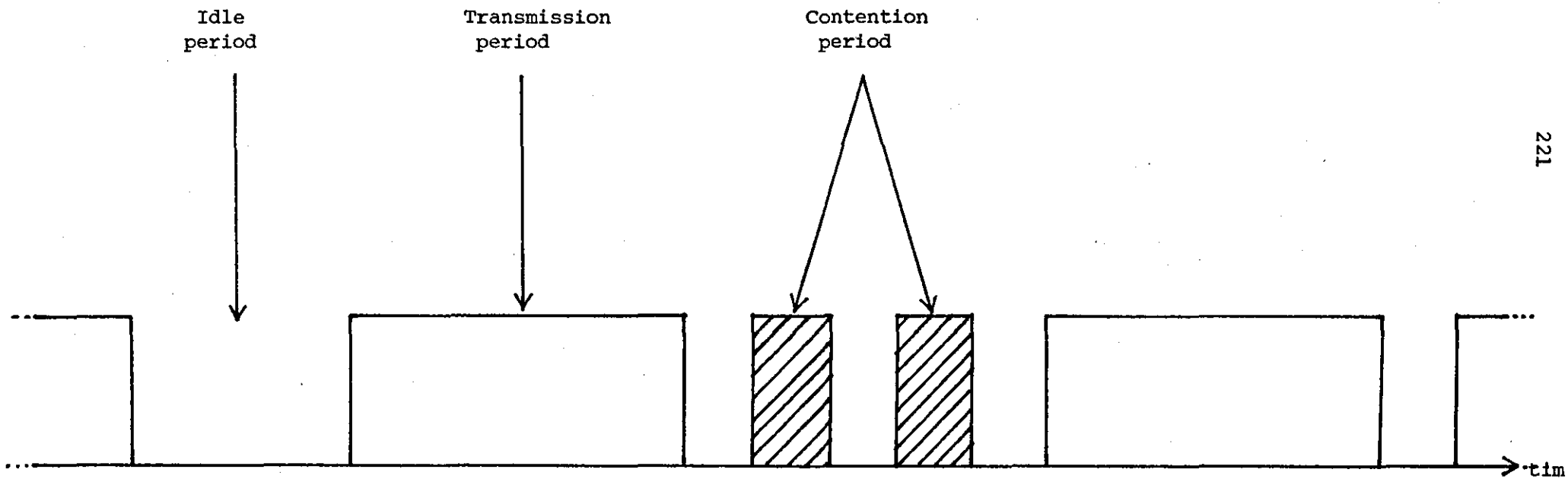


Fig. 5.4 Alternating Idle, Transmission and Contention Channel Periods

starting to transmit into the channel. Also, the channel time slots are assumed to be time synchronised by the tail of the preceding channel acquisition period. A slot will be empty when no ready-user chooses to transmit into it and it will contain a collision if more than one ready-users attempt to transmit into it. When only one ready-user transmits into a slot, then the channel has been acquired for the duration of a packet.

5.3.1.1 Channel Utilization and Throughput

Channel utilization is the fraction of time the channel is carrying uncorrupted packets. A set of formulae can be developed to characterise the performance expected of the channel:

Let

P = number of bits in the packet

C = peak capacity in bits per second carried on the channel

T = time in seconds of a slot = 2τ

τ = maximum propagation delay between two users in the network plus the carrier detection time

k = the number of ready-users who are continuously queued to transmit a packet. Either the enquiring user has a new packet immediately after a successful transmission period or another user becomes ready: k also happens to give the total offered load on the network.

Hence, a ready-user in the distributed queue attempts to transmit into the current slot with probability 1/k, or delays the transmission with probability 1 - 1/k. The maximum probability, A, that exactly one ready-user acquires the channel in the current slot is given by

$$A = k.\frac{1}{k}.(1 - \frac{1}{k})^{k-1} = (1 - \frac{1}{k})^{k-1}$$

i.e. there are k ways in which one ready-user can choose to transmit

(with probability 1/k) in the current slot while the remaining k-1 ready-users choose to defer (with probability 1 - 1/k). Hence, A is characterised by a geometric distribution.

From the value of A, the average waiting time can be calculated.

Let E(W) = the mean number of slots of waiting in a contention period before a successful acquisition of the channel

The probability of not waiting at all is = A.

The probability of waiting one slot only is = A(1-A)

The probability of waiting i slots is = $A(1-A)^i$, i.e. a geometric distribution whose mean is E(W).

When more than one user attempt to acquire the control of the channel, a collision occurs. Each of the colliding users must exercise a collision arbitration algorithm such as the binary exponential backoff (ref. 5.16). Hence

$$E(W) = \sum_{i=1}^{\infty} i\ (1-A)^i.A = \frac{1-A}{A}$$

From this, channel utilization and throughput can be estimated since the channel time is considered to be divided between the transmission and contention periods. The packet transmission time is P/C seconds and this is the actual length of the transmission period. The mean time to channel acquisition is E(W).T and this is the length of the contention period. Hence the maximum utilization S is

$$S = \frac{P/C}{P/C + E(W).T} = \frac{1}{1 + \frac{E(W).T.}{P/C}}$$

or

$$S = \frac{1}{1 + 2\alpha.E(W)} = \frac{1}{1 + 2\alpha.\ \frac{1-A}{A}}$$

where α = propagation parameter = $\frac{\tau}{b_1}$

From the above analysis it can be seen that the values of the channel utilization and throughput depend crucially on the values of T, C, P and k. High values of S can be achieved if the packet size P is large, for a given value of C, or if the value of α is small. It has been found that (ref. 5.16), the value of S approaches that of the slotted channel ALOHA throughput, (1/e), when the values of the packet sizes approach the slot size. Figs. 5.5 and 5.6 show the variation of the channel utilization with various values of k, P and α. Fig. 5.7 shows the variation of the channel utilization with various values of the propagation delay parameter α, and k.

5.3.1.2 Number Involved in a Collision

In order to exercise effectively some collision arbitration algorithms, such as the binary exponential algorithm, which require that the colliding users wait a random amount of time before attempting to retransmit, following a collision, it is often necessary to keep a running estimate of the number of ready-users in the system. One way to accomplish this is to use a logically separate subchannel for signalling the state changes. The first contention slot following each successful packet transmission may be set aside for some form of signalling. The running estimate should be updated when

a) a successful transmission occurs: estimate decremented

b) a new user becomes ready: estimate incremented.

It is easy to detect successful transmissions since all users listen to the channel and hence can easily keep track of the number of successful transmissions. The number of new users who become ready are more difficult to detect but they can be estimated from the knowledge of the alternating busy and idle periods of the channel state. When a collision occurs,

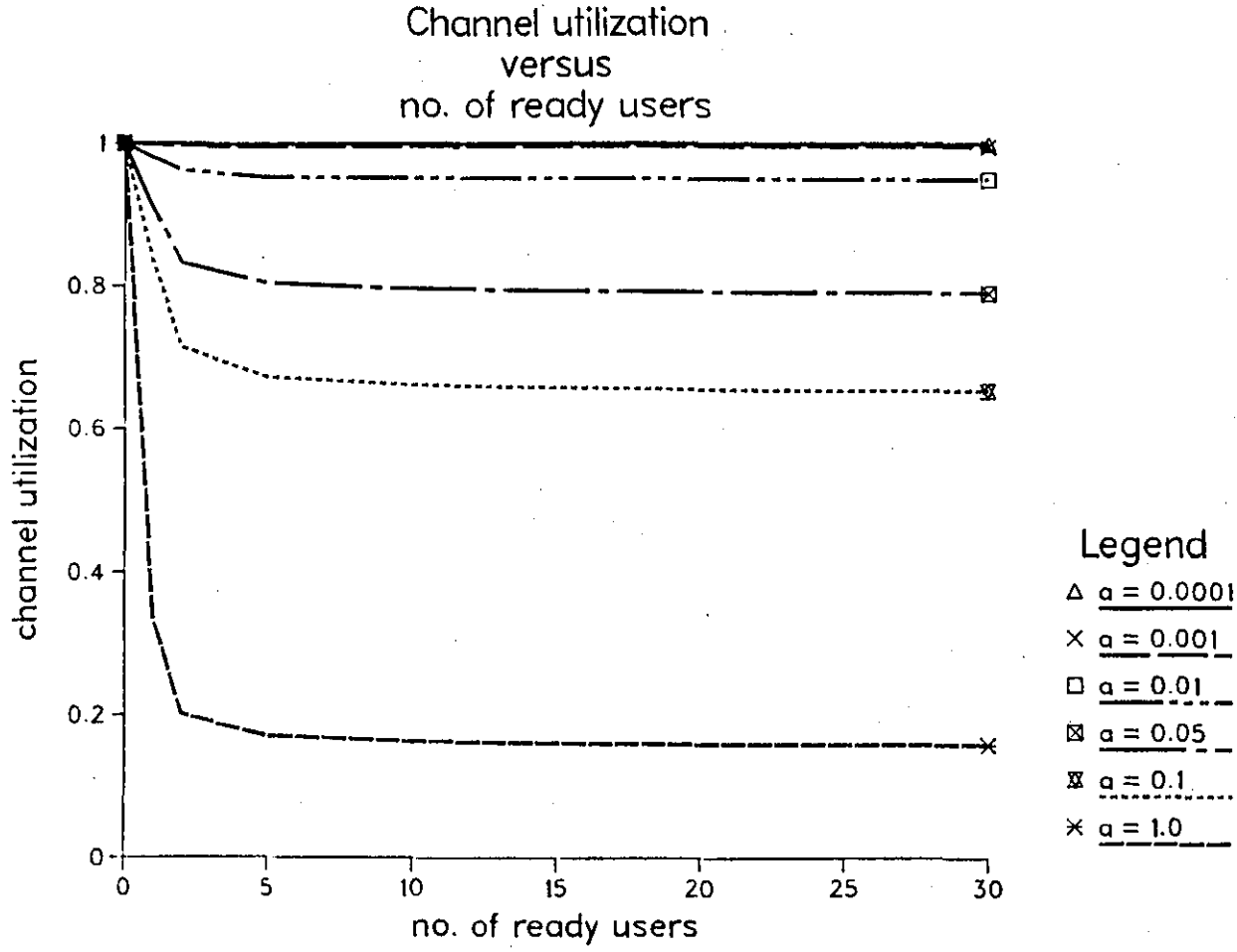


Fig. 5.5 Channel Utilization

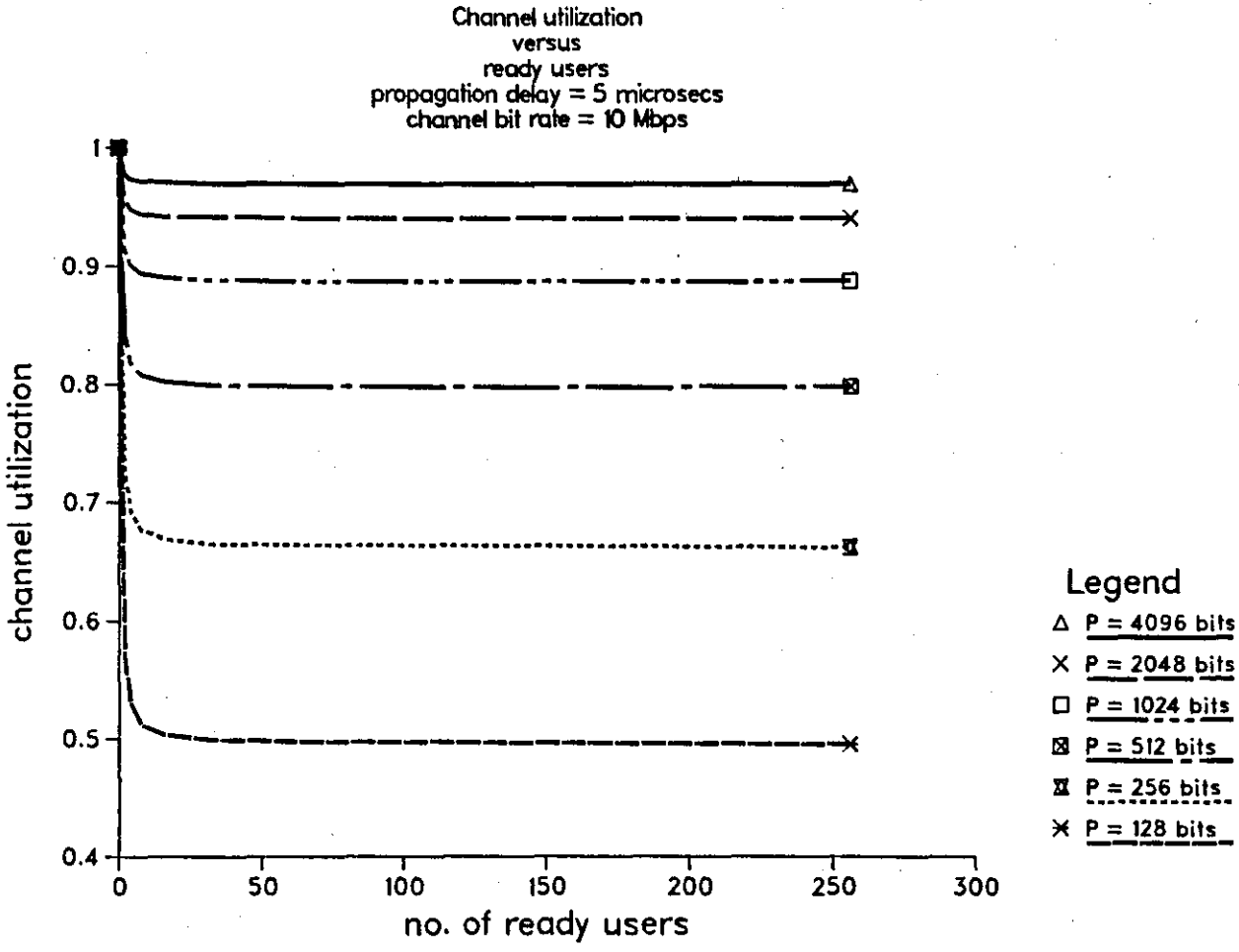


Fig. 5.6 Channel Utilization

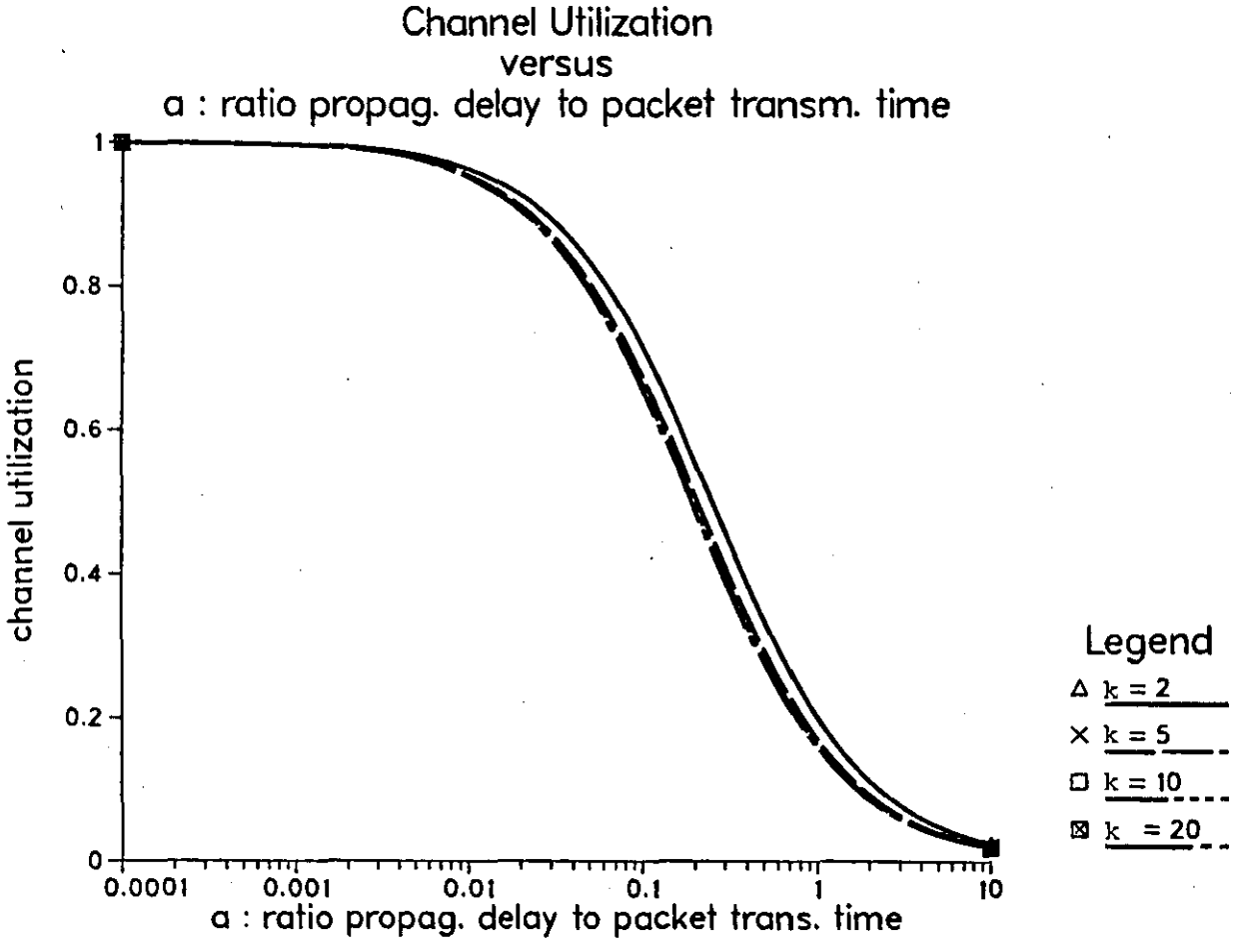


Fig. 5.7 Channel Utilization

all the users know that two or more users have collided.

To make an estimate of the average number of the new ready users represented by the occurrence of each collision in a time slot, it can be assumed that the number of users who become ready in each slot is generated by a Poisson process, with rate λ (ref. 5.11). From this assumption, the probability that exactly k users were involved in a collision P(K), conditioned on the fact that a collision occurred, can be calculated from

$$P(K) = \frac{\lambda^k . e^{-\lambda}}{k!(1 - e^{-\lambda} - \lambda . e^{-\lambda}}$$

The value of λ may be estimated from the average of two possible values obtained by recording the fraction of the channel slots corresponding to the probability of zero ($e^{-\lambda}$), or one, ($\lambda . e^{-\lambda}$), user becoming ready in a time slot. Knowing P(k), the mean value E(K) may be obtained, since

$$E(k) = \sum_{k=1}^{\infty} k.P(k)$$

$$= \frac{e^{-\lambda}}{1-e^{-\lambda}-\lambda e^{-\lambda}} \cdot \sum_{k=1}^{\infty} k.\frac{\lambda^k}{k!}$$

$$= \frac{\lambda}{1-e^{-\lambda}-\lambda . e^{-\lambda}}$$

However the mean E(k) will not be an integer, in general, but the nearest integer value may be used as the estimate.

5.3.2 Queueing Theoretic Channel Model

The performance of CSMA-CD channels can be characterised and determined by examining them using queueing models (ref. 5.8, 5.9, 5.12). In such models, the source of the traffic to the broadcast channel is again assumed to consist of an infinite population of network-users who collectively

form an independent Poisson process with an aggregate mean packet generation rate of λ packets per second. This also approximates a large but finite population in which each user generates packets infrequently and in which each packet can be transmitted in a time period much less than the average time period between successive packets generated by a given network-user. Furthermore, each user can store and attempt to transmit at most one packet at a time. By carefully specifying the collision arbitration algorithm, the throughput and channel delay performance may be calculated.

5.3.2.1 Collision Arbitration Algorithm

A suitable collision arbitration algorithm, such as the binary exponential backoff algorithm, is necessary to avoid collisions from building up and hence introducing instability (ref. 5.3, 5.15). The CSMA-CD protocol may be defined and specified by the following two possible courses of action for the ready-users:

subalgorithm one (A1):

Following a successful transmission period, each ready-user transmits with probability one into the next time slot.

subalgorithm two (A2):

Upon detection of a collision, each ready-user uses an adaptive algorithm for selecting its transmission probability in the next time slot.

In subalgorithm A2, a suitable protocol is used so that the probability of a successful transmission into the next time slot is equal to the slotted ALOHA-like channel throughput S, (ref. 5.2).

5.3.2.2 Throughput and Delay Performance

The ready-users can be considered to form a distributed queue with random order of service in the broadcast channel. The method of imbedded

Markov chain analysis (ref. 5.11, 5.17), can be used to derive the equilibrium moment generating function of the distributed queue size. From the moment generating function for the distributed queue size, the important performance measure characterising the mean packet delay experienced by the packet can be obtained. Under the assumptions of Poisson arrivals and that packets arrive and depart one at a time, the moment generating function of the queue size obtained for the imbedded points is valid for all points in time (ref. 5.17).

The transmission time of each packet is assumed to be an independent, identically distributed random variable with the probability distribution function (PDF) b(x) with mean b_1, second moment b_2, and Laplace Transform b(s), (ref. 5.18, 5.19).

$$b(s) = \int_0^\infty e^{-sx}.b(x).dx$$

The snapshot of the channel, representing the busy and idle periods, is shown in Fig. 5.8, from which the fundamental equation for the imbedded Markov chain queueing system is

$$q_{n+1} = q_n + U_{n+1} + V_{n+1} - 1$$

where

q_n = the number of the ready-users left behind by the departure of the n^{th} successful transmission, C_n

y_{n+1} = the time from the departure of C_n to the beginning of the next successful transmission.

U_{n+1} = the number of new Poisson arrivals during the time period y_{n+1}

X_{n+1} = the channel transmission time of C_{n+1}

V_{n+1} = the number of new Poisson arrivals during the channel transmission time, X_{n+1}

But X_{n+1} has PDF b(x).

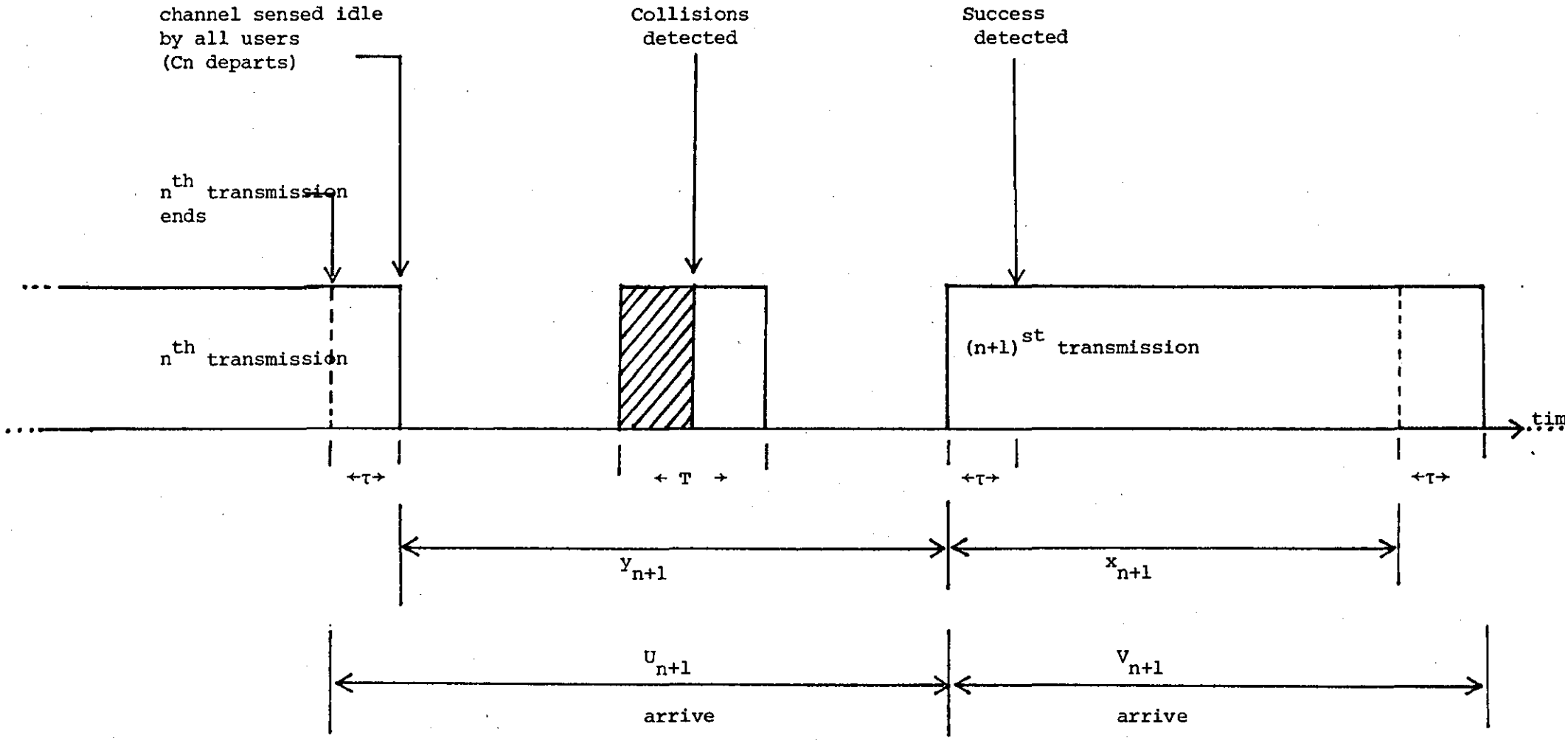


Fig. 5.8 Snapshot of the Channel

Let B(x) be the PDF for the period $X_{n+1} + \tau$. Hence, the Laplace Transform B(s) of B(x) is given by

$$B(s) = b(s).e^{-s\tau}$$

y_{n+1} is a random variable which is the sum of two independent time intervals, so that

$$y_{n+1} = (I_{n+1} + r_{n+1}).T$$

where:

T = duration of the time slot, (2τ)

I_{n+1} = the number of slots in an idle period immediately following the departure of C_n

r_{n+1} = the number of slots in the contention period following a collision until the next successful transmission

The slot containing the initial collision is included in r_{n+1}. According to the specification of subalgorithms A1 and A2, it can be seen that I_{n+1} is non-zero only if $q_n = 0$. Also, if there has been no collision when the transmission period of C_{n+1} begins, then $r_{n+1} = 0$.

Let p_j = the probability of j new ready-users arriving in a time slot then

$$p_j = \frac{(\lambda T)^j}{j!} .e^{-\lambda T}, \text{ for } j = 0, 1, 2, ...$$

At the beginning of the next time slot each new arrival executes A1 or A2 in exactly the same manner as all the other ready-users.

Hence,

$$P(I_{n+1} = k | q_n = 0) = (1-p_0).p_0^{k-1}, \text{ for } k = 1, 2, ...$$

and

$$P(r_{n+1} = k | \text{Collision occurred}) = S.(1-S)^{k-1}, \text{ for } k = 1, 2, ...$$

It can be observed that the above conditioned probabilities are geometrically distributed. The Laplace Transform, C(s), of the probability density function (pdf) of a contention period, given that a collision occurred, can be obtained. The z-Transform of a geometric series for the contention slots is given by

$$\sum_{k=1}^{\infty} S.(1-S)^{k-1}.z^{k} \qquad \text{for } k = 1, 2, \ldots$$

or

$$\sum_{k=0}^{\infty} S(1-S)^{k-1}.z^{k} - S(1-S)^{-1}, \quad k = 0, 1, \ldots$$

or

$$\frac{S}{1-S}\left[\frac{1}{1-(1-S).z} - 1\right]$$

$$= \frac{S.z}{1-(1-S).z}$$

Hence, the Laplace Transform, C(s), is given by

$$C(s) = \frac{S.e^{-sT}}{1-(1-S).e^{-sT}}$$

from which the first and second moments may be calculated. The mean value of C(s) is given by

$$-\frac{d}{ds} C(s)\Big|_{s=0} = \frac{T}{S}$$

and the second moment by

$$\frac{d^2}{ds^2} C(s)\Big|_{s=0} = T^2\left[1 + \frac{2(1-S)}{S^2}\right]$$

As was mentioned above, the imbedded Markov chain characterisation of the snapshot of the channel is given by the fundamental relation

$$q_{n+1} = q_n + U_{n+1} + V_{n+1} - 1 \qquad (1)$$

in which V_{n+1} is an independent random variable with the z-Transform $B(\lambda-\lambda z)$, (ref. 5.17, 5.20), while U_{n+1} depends upon q_n in the following manner as a consequence of A1 and A2:

given

(i) $q_n = 0$, then

$$U_{n+1} = \begin{cases} 1 & \text{:with probability } \dfrac{p_1}{1-p_o} \\ j + \text{number of arrivals during the contention period} & \text{:with probability } \dfrac{pj}{1-p_o},\ j = 2, 3, \ldots. \end{cases}$$

ii) $q_n = 1$, then $U_{n+1} = 0$

iii) $q_n \geq 2$, then U_{n+1} = number of arrivals during a contention period

(2)

Furthermore, given the occurrence of a collision, the number of new arrivals during a contention period is an independent random variable with the z-Transform $C(\lambda-\lambda z)$, (ref. 5.17, 5.20).

$$C(\lambda-\lambda z) = \frac{S.e^{-(\lambda-\lambda z).T}}{1-(1-S).e^{-(\lambda-\lambda z).T}}$$

The equilibrium queue probabilities are given by

$$Q_k = \lim_{n\to\infty} P(q_n=k),\ k=0, 1, 2, \ldots$$

The equilibrium queue probabilities Q_k exist as long as the service rate of the packets by the transmission channel exceeds the packet generation rate by the network-users, so that

$$\lambda(b_1+\tau+\frac{T}{S}) < 1$$

Defining the Z-transform $Q(z)$ of the equilibrium queue size

$$Q(z) = \sum_{k=0}^{\infty} Q_k.z^k \qquad (3)$$

and considering equations (1) and (2) above, and taking limit as $n \to \infty$, then the equilibrium moment generating function of the queue size can be calculated. It can be noted from (3) that, (ref. 5.17, 5.9)

$$Q(z) = E\left[z^{q_{n+1}}\right]$$

From (1)

$$z^{q_{n+1}} = z^{\left[q_n + U_{n+1} + V_{n+1} - 1\right]}$$

so that

$$E\left[z^{q_{n+1}}\right] = E\left[z^{(q_n + U_{n+1} + V_{n+1} - 1)}\right]$$

Since the number of new arrivals during a contention period is an independent random variable, then

$$E\left[z^{q_{n+1}}\right] = E\left[z^{V_{n+1}}\right].E\left[z^{q_n + U_{n+1} - 1}\right]$$

in which, (ref. 5.17)

$$E\left[z^{V_{n+1}}\right] = V(z)$$

and

$$V(z) = B(\lambda - \lambda z)$$

Furthermore

$$E\left[z^{(q_n + U_{n+1} - 1)}\right] = \sum_{(\text{all } j)} \sum_{(\text{all } k)} P\left[U_{n+1} = j,\ q_n = k\right].z^{j+k-1} \quad (4)$$

After some algebraic manipulation of (4), a form of the Pollaczek-Khinchin transform equation, (ref. 5.9, 5.17), for the equilibrium moment generating function expressing the Z-transform for the number of the "customers" in the system may be obtained as

$$Q(z) = B(\lambda - \lambda z).\left[Q_1.z.\left[1 - C(\lambda - \lambda z)\right] + \frac{Q_o}{1 - p_o}\left[p_1.z.(1 - C(\lambda - \lambda z)) - C(\lambda - \lambda z).(1 - e^{-\lambda T(1-z)})\right]\right] \div \left[Z - B(\lambda - \lambda z).C(\lambda - \lambda z)\right] \quad (5)$$

where

$$Q_o = \frac{1 - \lambda(b_1 + \tau + T/S)}{\lambda T\left[\frac{1}{1-p_o} - \frac{1}{B(\lambda).S}\right]} \qquad (6)$$

and

$$Q1 = \left[\frac{1}{B(\lambda)} - \frac{p_1}{1-p_o}\right].Q_o \qquad (7)$$

Q(z) also represents the total distribution for the total time spent in the system for customer C_n (ref. 5.17), so that after further algebraic manipulation of (5) and the application of Little's result (ref. 5.21), the mean packet delay - i.e. the time since arrival to the time of departure, D, may be shown to be given by

$$D = \bar{x} + \frac{T}{S} + \frac{T}{2} - \frac{1 - p_o}{2\left|B(\lambda).S-(1-p_o)\right|}.\left(\frac{2}{\lambda} + ST-3T\right)$$

$$+ \frac{\lambda\left[\overline{x^2} + 2\bar{x}(T/S) + T^2(1+2(1-S)/S^2)\right]}{2\left[1-\lambda(\bar{x} + T/S)\right]} \qquad (8)$$

where

$\bar{x} = b_1 + \tau$

and

$\overline{x^2} = b_2 + 2b_1.\tau + \tau^2$

From the above analysis the channel assignment delay may also be calculated - i.e. given that the channel is free and that there is at least one ready user, the pdf of the time the above conditions are satisfied to the start of the next successful transmission.

Let

d_n = random variable representing the channel assignment delay immediately prior to the n^{th} transmission,

and

$d = \lim_{n \to \infty} d_n$

then

$$P[d=kT] = \begin{cases} Q_o \cdot p_1/(1-p_o) + Q_1, & \text{for } k = 0 \\ \left[Q_o \cdot \left[1-p_1/(1-p_o)\right] + \sum_{i=2}^{\infty} Q_i\right] \cdot S(1-S)^{k-1}, & \text{for } k = 1,2,3, \ldots \end{cases} \quad (9)$$

from which the mean channel assignment delay, $\bar{d}$, is given by

$$\bar{d} = \frac{1}{S}\left[1 - Q_o \cdot p_1/(1-p_o) - Q_1\right] \cdot T \quad (10)$$

From (10) it can be seen that

$$Q_o \cdot \frac{p_1}{1-p_o} + Q_1 \quad (11)$$

represents the fraction of transmissions that incur zero assignment delay in gaining channel access - i.e. fraction of transmissions that do not encounter any collisions. Hence (11) represents the probability of zero channel assignment delay.

5.3.2.3 Performance Observations

Let

$\alpha = \frac{\tau}{b_1}$ = ratio of carrier sense time to the mean packet transmission time

and

$\rho = \lambda b_1$

= channel throughput

= fraction of the channel time utilized by the packets under equilibrium conditions

The requirement for $\lambda(\bar{x} + T/S) < 1$ gives rise to the following upper bound on the channel throughput.

since

$$\rho = \lambda b_1$$

and

$$\lambda(\overline{x} + \frac{T}{S}) < 1$$

or

$$\lambda b_1 (\overline{x} + \frac{T}{S}) < b_1,$$

then

$$\lambda b_1 < \frac{b_1}{\overline{x} + T/S}$$

or

$$\rho < \frac{S.b_1}{S\overline{x} + T} = \frac{S}{\left(\frac{\overline{x}}{b_1}\right) + \left(\frac{T}{b_1}\right)}$$

and using $\overline{x} = b_1 + \tau$ and $T = 2\tau$, then the upper bound on the channel throughput is given by

$$\rho < \frac{S}{2\alpha + (1+\alpha).S} \qquad (12)$$

from which $\rho \doteq 1$ if $\alpha <<< 1$

The performance of the queueing theoretic channel model may be examined by considering specific channel service time distributions.

5.3.2.3.1 Constant Packet Time

In the case in which the service time is a constant and equal to $\overline{x}$ the Laplace transform of the service time distribution (packet transmission time) is given by

$$B(s) = e^{-s\overline{x}}, \text{ and } b_2 = 0$$

so that

$$\overline{x} = b_1 + \tau \quad \text{and} \quad \overline{x^2} = 2b_1.\tau + \tau^2$$

Substituting these into (8) gives

$$\frac{D}{b_1} = 1 + 7.44\alpha - \frac{5.44\alpha(1-6.23\alpha\rho)}{1 - \rho(1+6.44\alpha) + \rho^2(\alpha+0.5)} +$$

$$\frac{\rho(1 + 12.87\alpha + 53.37\alpha^2)}{2(1 - \rho - 6.44\alpha\rho)} \qquad (13)$$

(13) can be used to examine the performance of the channel model for the normalized mean delay, D/b_1, with the channel throughput ρ for various values of α. Hence the delay performance of the channel is principally governed by the values of α and ρ and that for some values of ρ, the delay experienced by the packets can be very high. Also, from (13) it can be seen that if $\alpha=0$, then, as expected

$$\frac{D}{b_1} = 1 + \frac{\rho}{2(1-\rho)} = \frac{2-\rho}{2(1-\rho)} \qquad (14)$$

which is the classical $M|D|1$ result. If ρ is very low, then

$$D \simeq b_1$$

so that the only delay experienced by the packets, from the time of arrival to the time of departure, consists only of the packet transmission and no waiting delay is experienced. Figs. 5.9, 5.10 and 5.11 show the theoretical results expressed by equations (13), (10) and (11) respectively for the case of the constant service time. The dashed curve in fig. 5.9 shows the result for the pure M/D/1 system in which $\alpha=0$, as shown in equation (14).

In the case in which the packet sizes P, or the transmission time b_1 for the packets is very large compared with the slot time, the condition $\alpha=0$ can be approached. But in practice, $\alpha>0$. The constant service time distribution is quite appropriate in the case in which the traffic generated

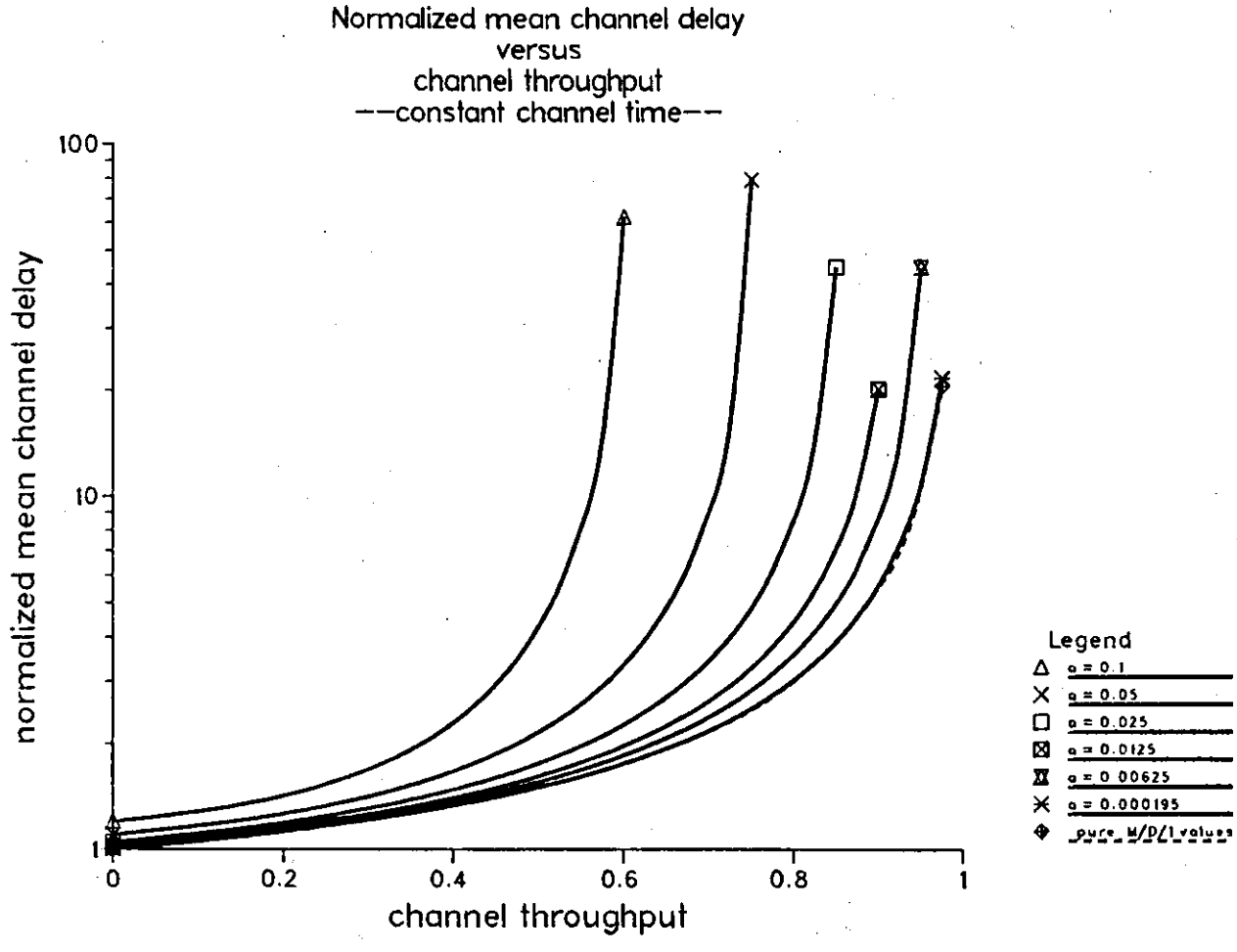


Fig. 5.9 Normalized Mean Channel Delay

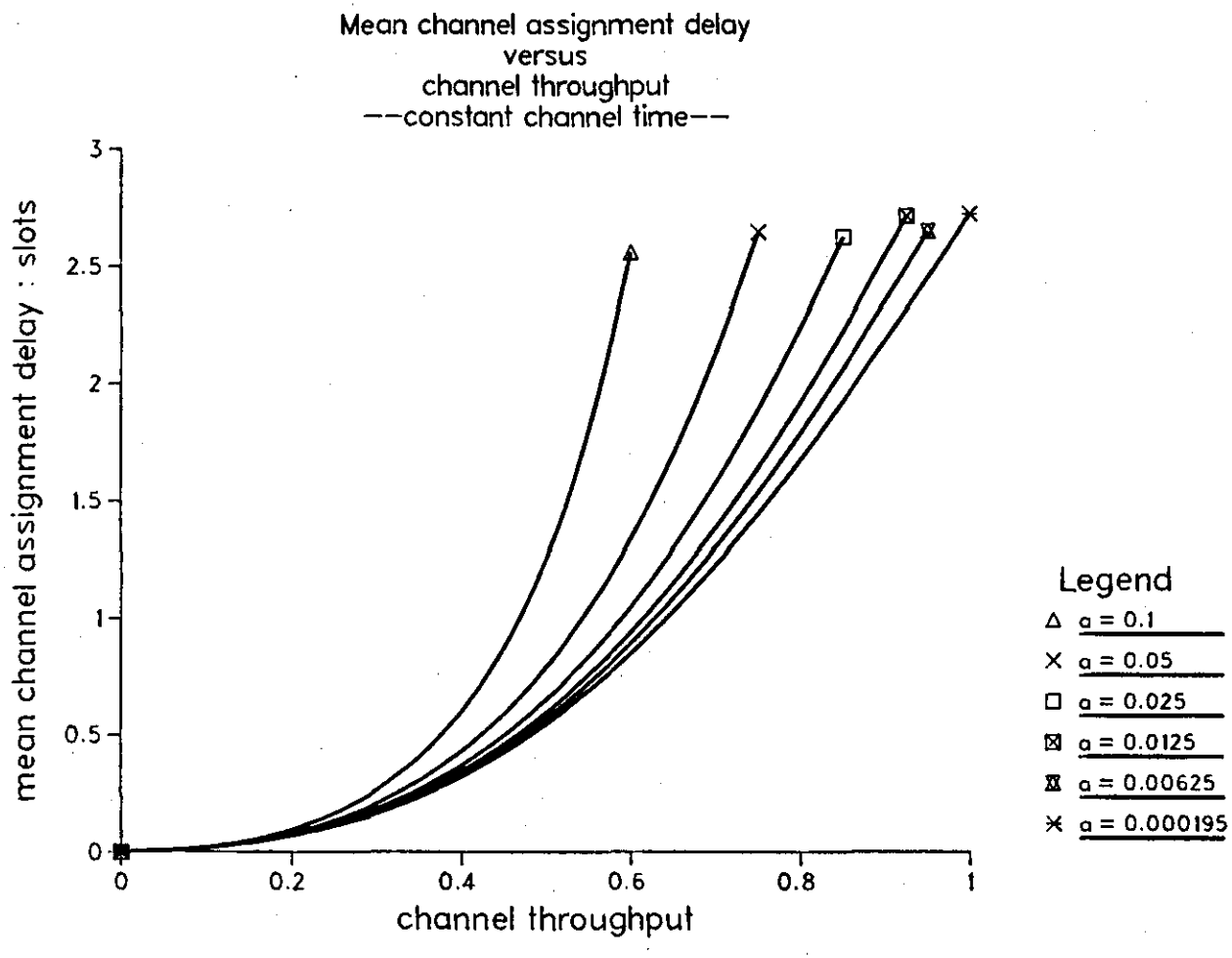


Fig. 5.10 Mean Channel Assignment Delay

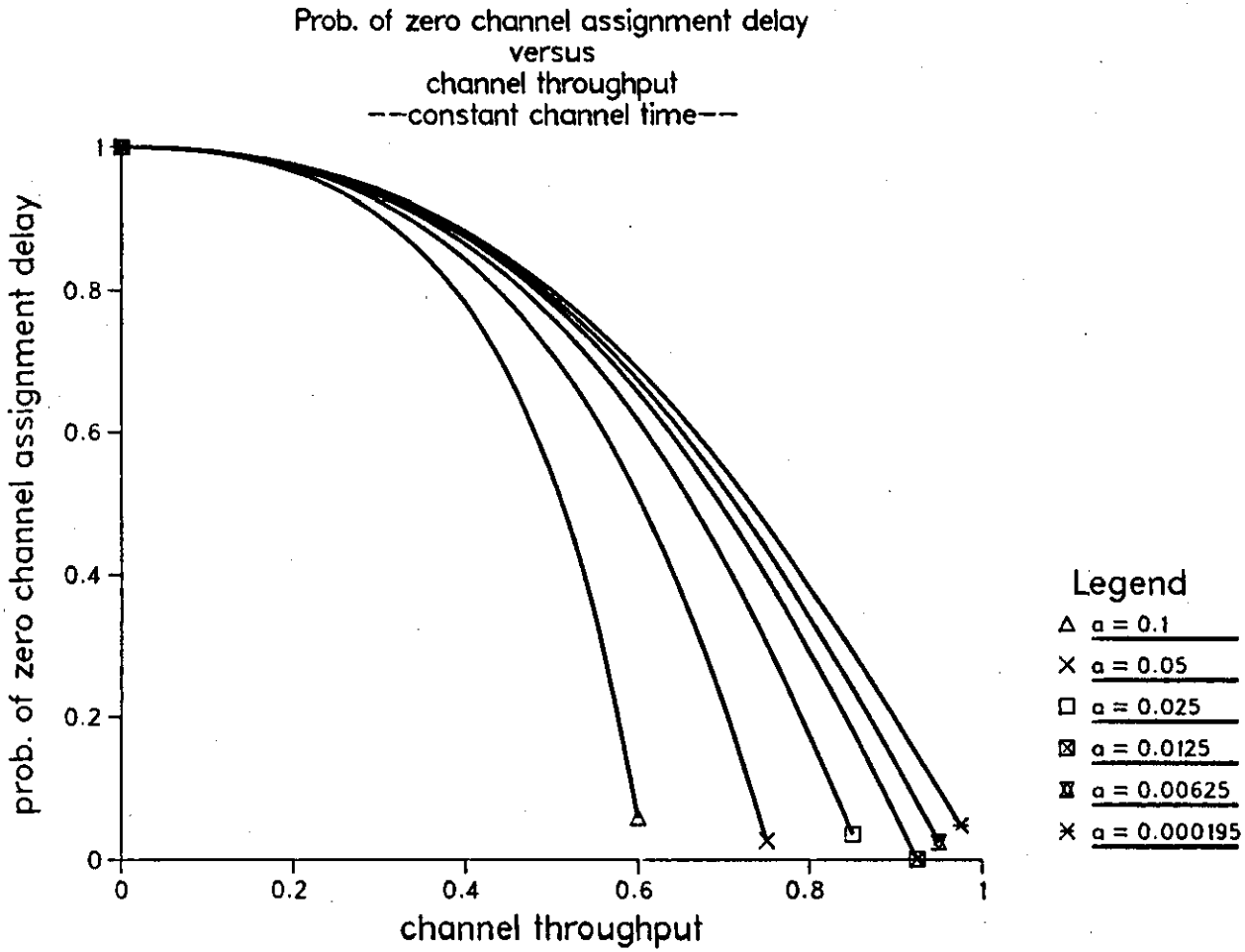


Fig. 5.11 Probability of Zero Channel Assignment Delay

by the various network users appear to be connected to the channel at a centralized location.

5.3.2.3.2 Random Packet Time

In the situation where the various network-users may be considered to be scattered widely in a completely random manner within LAN, the module and intermodule packet transmission time will also be completely random. In this case the pdf for the packet transmission time may be described by the negative exponential distribution, (ref. 5.14), with Laplace transform

$$B(s) = \frac{1}{1+s\bar{x}}$$

in which $\bar{x} = b_1+\tau$ and $b_2=2(\bar{x})^2$

Substituting these values in (8), the expression for the delay becomes

$$\frac{D}{b_1} = 1 + 7.44\alpha - \frac{5.44\alpha(1+\rho+\alpha\rho)(1-2.632\alpha\rho)}{1-5.44\alpha\rho(1+\rho+\alpha\rho)} + \rho\frac{(1+6.44\alpha+27.62\alpha^2)}{1-\rho(1+6.44\alpha)} \qquad (15)$$

As before (15) can be used to examine the performance of the channel model for the variation of the normalized mean delay, D/b_1, for various values of α and ρ. If $\alpha=0$, in (15), then, as expected

$$D/b_1 \simeq \frac{1}{1-\rho} \qquad (16)$$

which gives the classical M/M/1 performance. In a high-speed channel environment employing long packets the condition for small values of α can be achieved. Figs. 5.12, 5.13 and 5.14 show corresponding results to those of figs. 5.9, 5.10 and 5.11.

Shoch and Hupp (ref. 5.22) have carried out an analysis of the measured performance of an Ethernet LAN employing 120 directly connected network-users. In this study it was found that under normal load the

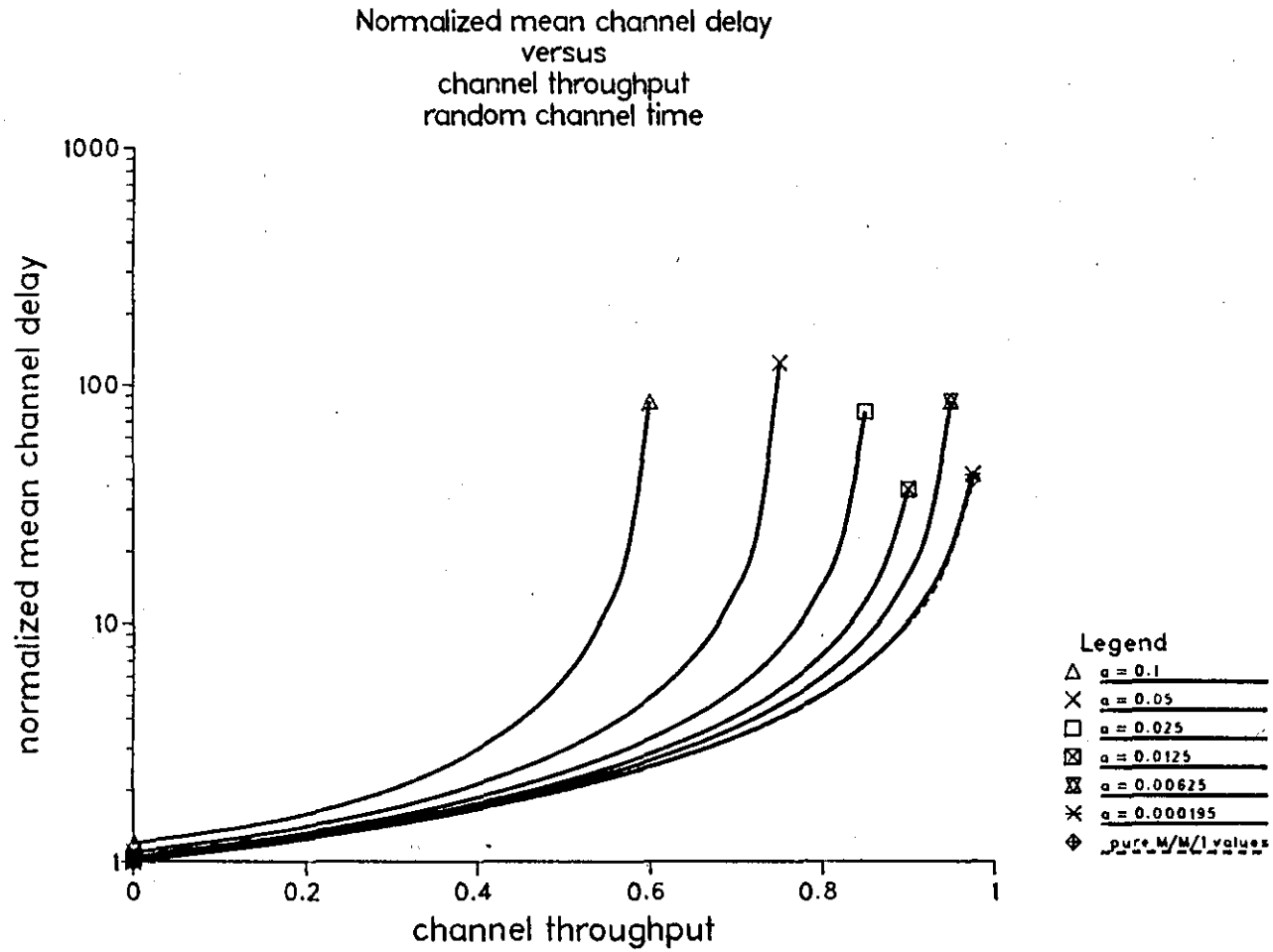


Fig. 5.12 Normalized Mean Channel Delay

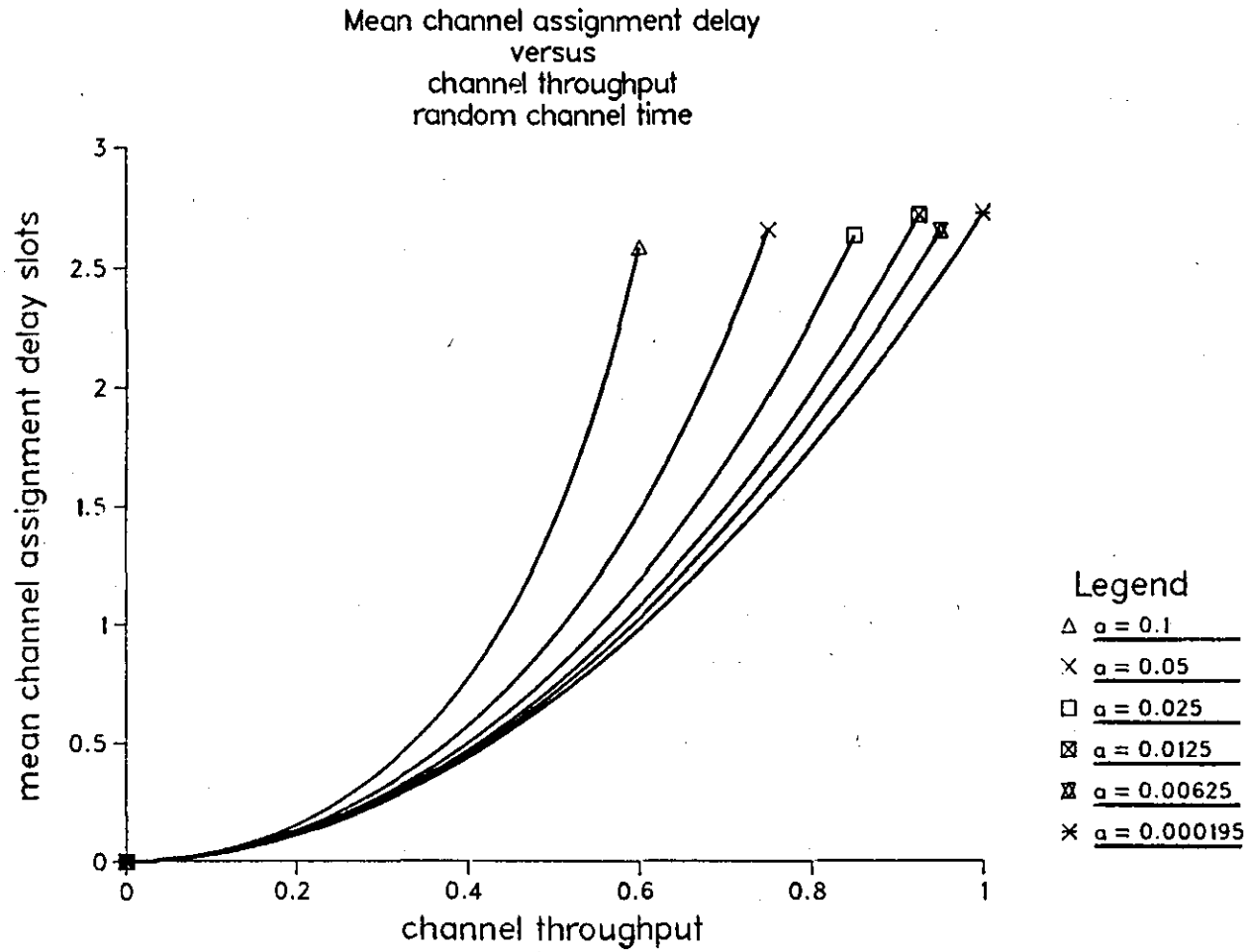


Fig. 5.13 Mean Channel Assignment Delay

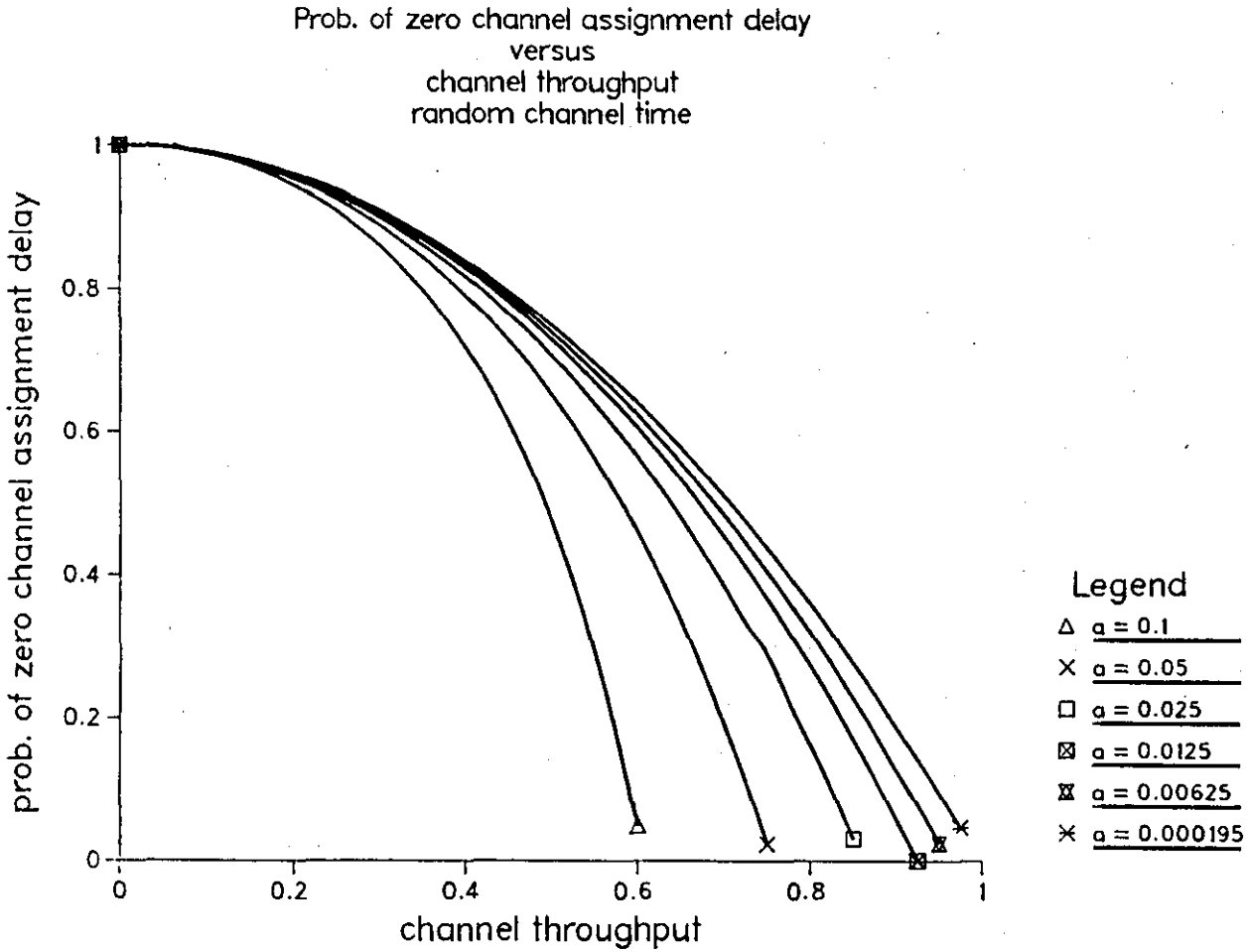


Fig. 5.14 Probability of Zero Channel Assignment Delay

system performs very well and with extremely low error rate (1 packet in 2 million) and also that the number of collisions in the channel are very few (less than 0.03% of the packets were involved in collisions) while the channel utilization and throughput remained high (99.18%). Also only about 0.79% of the packets were delayed due to deference. From this it can be seen that the channel delay performance, for low to medium traffic, remains low and that collisions in the channel are negligibly low. This low channel delay performance can also be seen from the analytical results of figs. 5.9 and 5.12, which also show that the value of α plays a dominant role in the overall performance of the channel.

Fig. 5.15 shows both the experimental (simulation) and analytical results of the channel delay performance. This comparative performance is based on equation (16) in which the channel transmission time distribution used is the negative exponential distribution.

5.3.2.3.3 More General Packet Time Distributions

The results for the performance measures of the channel derived above were based on the constant and completely random packet transmission times in the channel, respectively. These correspond to the values of the coefficient of variation, C_x, of 0 and 1, respectively. Packet transmission time distributions having the coefficient of variation less than or greater than 1 which are better approximations may also be used to model the channel performance. This can be done by employing exponential service stages to model hypoexponential or hyperexponential service time distributions.

Fig. 5.16 (a) shows how an arrangement of k exponential stages can be connected in series (a k-stage Erlangian server) can be designed to match a desired mean and standard deviation of a service time distribution

whose value of C_x is less than 1. The Laplace Transform of such a hypoexponential service time distribution is given by (ref. 5.17)

$$b(s) = \left[\frac{k\mu}{s+\mu} \right]^k$$

and the pdf by

$$b(x) = \frac{k\mu(k\mu x)^{k-1}}{(k-1)!} \cdot e^{-\mu k x}, \text{ for } x \geq 0$$

from which, the mean service time is

$$E(X) = \frac{1}{\mu}$$

and the variance

$$V(X) = \frac{1}{k.\mu^2}$$

and the coefficient of variation

$$C_x = \frac{1}{\sqrt{k}}$$

and in which μ is the rate of the exponential distribution, or $b_1 = \frac{1}{\mu}$.

Similarly fig. 5.16 (b) shows a k-stage parallel server to match a desired mean and standard deviation of a service time distribution whose value of C_x is greater than 1.

Assuming that

$$\sum_{i=1}^{k} p_{oi} = 1$$

the service time pdf is given by

$$b(x) = \sum_{i=1}^{k} p_{oi} \cdot \mu_i \cdot e^{-\mu_i \cdot x}, \quad \text{for } x \geq 0$$

whose Laplace transform is

$$b(s) = \sum_{i=1}^{k} p_{oi} \cdot \frac{\mu_i}{s+\mu_i}$$

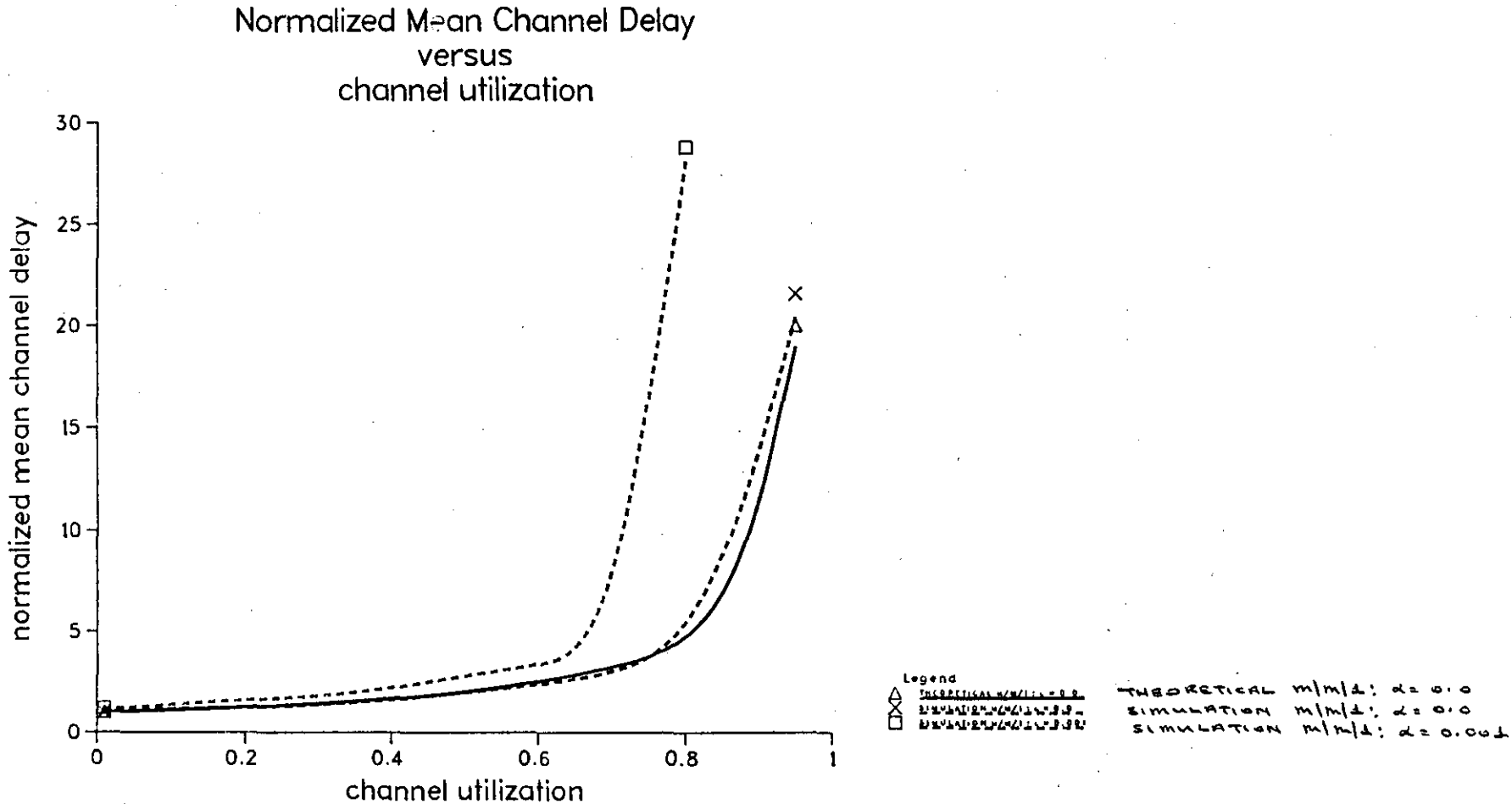
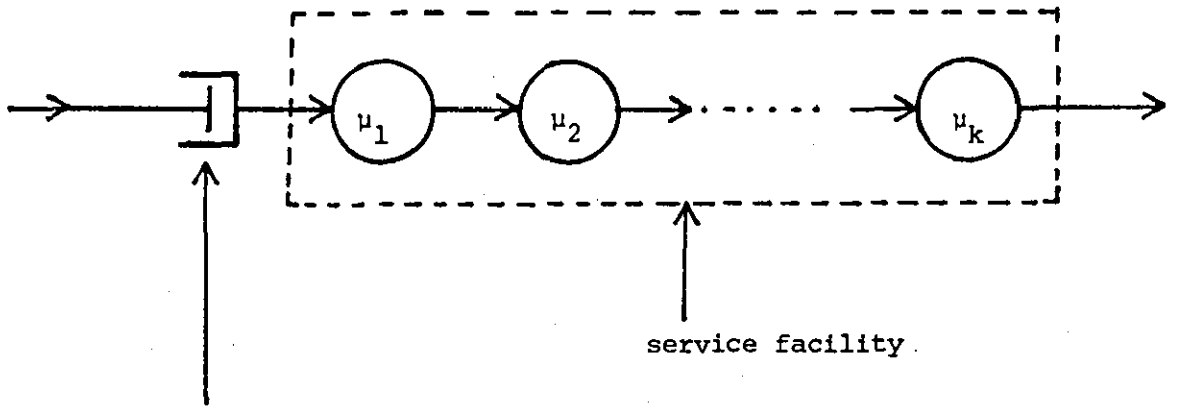
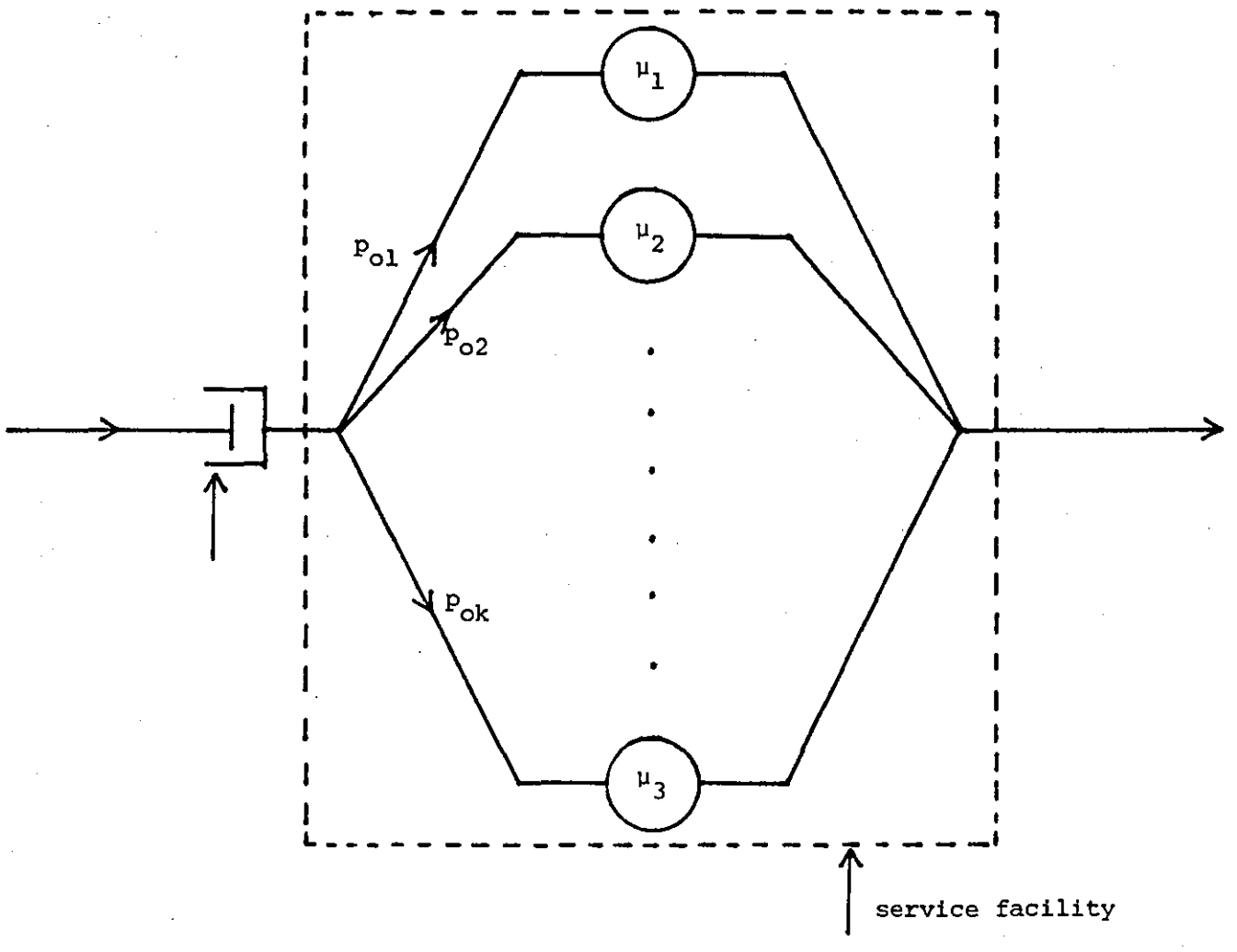


Fig. 5.15 Normalized Mean Channel Delay



(a)



k-stage parallel server

(b)

Fig. 5.16 Stage Servers

so that the mean service time is

$$E(X) = \sum_{i=1}^{k} p_{oi}/\mu_i$$

and second moment

$$E(x^2) = 2 \sum_{i=1}^{k} p_{oi}/\mu_i^2$$

so that

$$C_x^2 = \frac{2 \sum_{i=1}^{k} \frac{p_{oi}}{\mu_i^2}}{\left[\sum_{i=1}^{k} \frac{p_{oi}}{\mu_i} \right]^2} - 1$$

in which

$$C_x^2 \geq 1$$

Often the k-stage series-parallel branching Erlang server is used to model the hypoexponential and hyperexponential service time distributions, as shown in fig. 5.17. The Laplace transform of such a server may be given by, (ref. 5.17)

$$b(s) = p_{1O} + \sum_{i=1}^{k} b_1.b_2 \ldots b_i.p_{i+1,O} \prod_{j=1}^{i} \left[\frac{\mu j}{s+\mu j} \right]$$

Having determined E(X), and C_x, the minimum mean delay D may be estimated from

$$D = \left[1 + \frac{\rho(1+C_x^2)}{2(1-\rho)} \right].E(X) \qquad (17)$$

where, for λ = Poisson arrival rate of packets

$$\rho = \frac{\lambda}{\mu} = \lambda b_1$$

and assuming that α=O.

Fig. 5.18 shows a graphical performance for the normalized mean delay for a system performance that may be characterised by equation (17) for

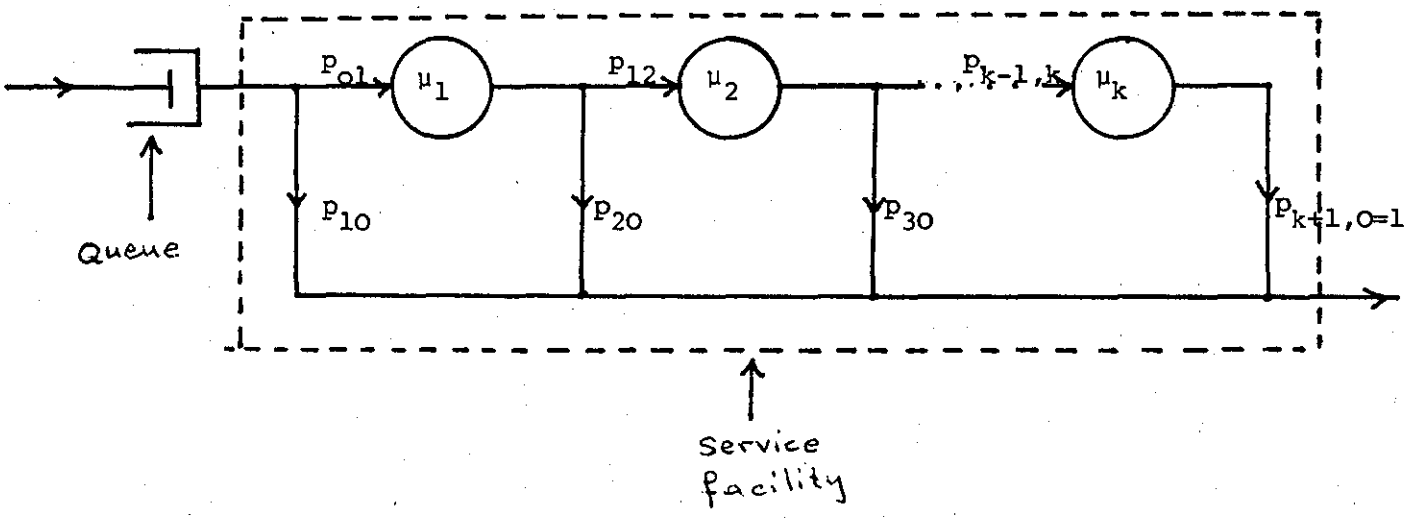


Fig. 5.17 Branching Erlang Server

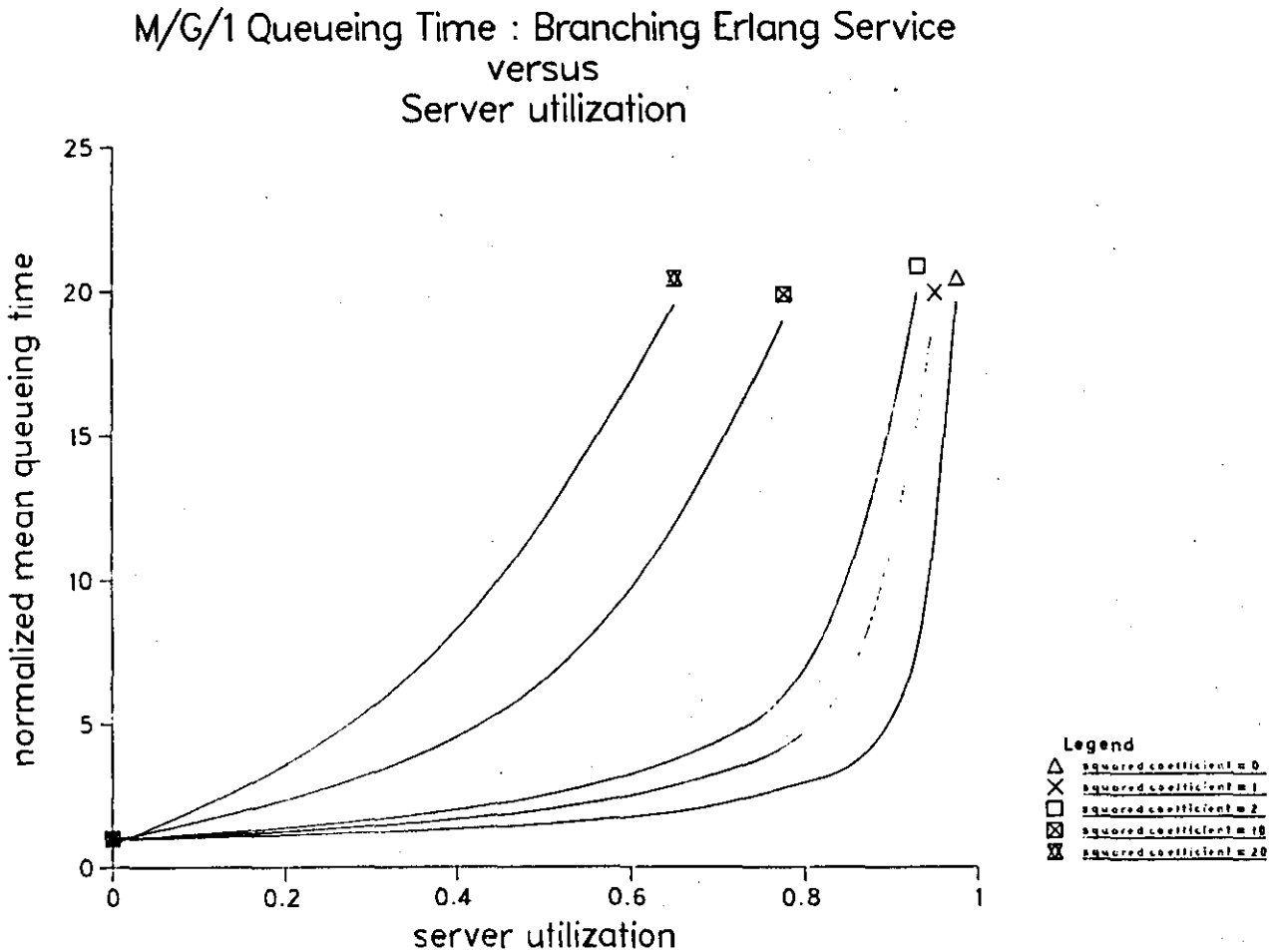


Fig. 5.18 Branching Erlang Service

various values of the squared coefficient of variation C_x^2. Values of $C_x^2 = 0$ correspond to the constant packet transmission time expressed by equation (14) while $C_x^2 = 1$ corresponds to the randomized packet transmission time expressed by equation (16). The higher the value of C_x^2, the more variable the packet transmission time and the service time distribution for the channel.

However, service time distributions that characterise the channel transmission time for packets in the channel are generally found to be less variable and hence $C_x^2 \leq 1$, (ref. 5.14).

CHAPTER 6

DISTRIBUTED COMPUTATION MODEL

6.1 INTRODUCTION

As was explained earlier, the total delay performance of a dual processor CPU cache distributed computation system is an important design objective. In such a distributed computation environment, the decision to partition a given computation and run, the various program modules on either of the two computers strictly depends on the values of the module run times and intermodule communication times. Whereas the module processing time at each of the source processors can be estimated reasonably accurately, it is much more difficult to estimate the module processing time at the sink processor. The module processing time at the sink processor is dependent on many factors, but the main difficulty in trying to estimate the running time there is because the sink is primarily a heavily shared resource system. The operating system of such a heavily shared resource attempts to provide high performance to the population of the network-udrtd (customers, jobs, modules) who attempt to share some of the following service facilities:

a) Terminals and other I/O devices

b) The secondary memory

c) The primary memory

d) The CPU

e) The printers

f) The plotters

g) The readers

h) The punches

Hence the sink processor resource system is often time-shared, multi-accessed and multi-programmed and hence efficient resource management capability is necessary if high performance is to be provided. In a dual processor CPU cache distributed computation environment incorporating such a sink processor resource system, it is necessary that the system response time is reasonably low so that the benefits of distributing the computation thus can be realised. The sink processor system response time will depend primarily on the number and the characteristics of the particular resources needed by an arriving customer. It will also depend on the amount of service demanded by the arriving customer. In the dual processor CPU cache distributed computation system of the type examined here, only the CPU and I/O service facilities are assumed to be dominant in contributing to the sink processor system response time.

If the number of the system users is small, then it may be expected that the system response time will be small. But, as the number of the system users increases, the number of the competitive conflicting demands will also tend to increase. As the number of these competitive and conflicting demands for the system resources rises, the overall system response time performance will also continue to degrade to a point whereby the system can be considered to have saturated so that it can no longer provide much benefit to the network-users in a distributed computation CPU cache arrangement in which the overall aim is to attempt to reduce the total computation time of a given computation. Hence a principal objective in such a distributed computation arrangement is to characterise and examine how the computational workload of the sink processor resource system varies with the number of the competitive demands. As this workload increases it may be expected that, with the aid of some form of an inherent

feedback mechanism, the number of users will be discouraged and decide against having any of their computation process at the sink processor system, whenever possible. Also, beyond certain computational workloads at the sink processor, the source processors already with some portion of their computation at the sink processor may be expected to withdraw some or all of their scheduled modules, whenever possible.

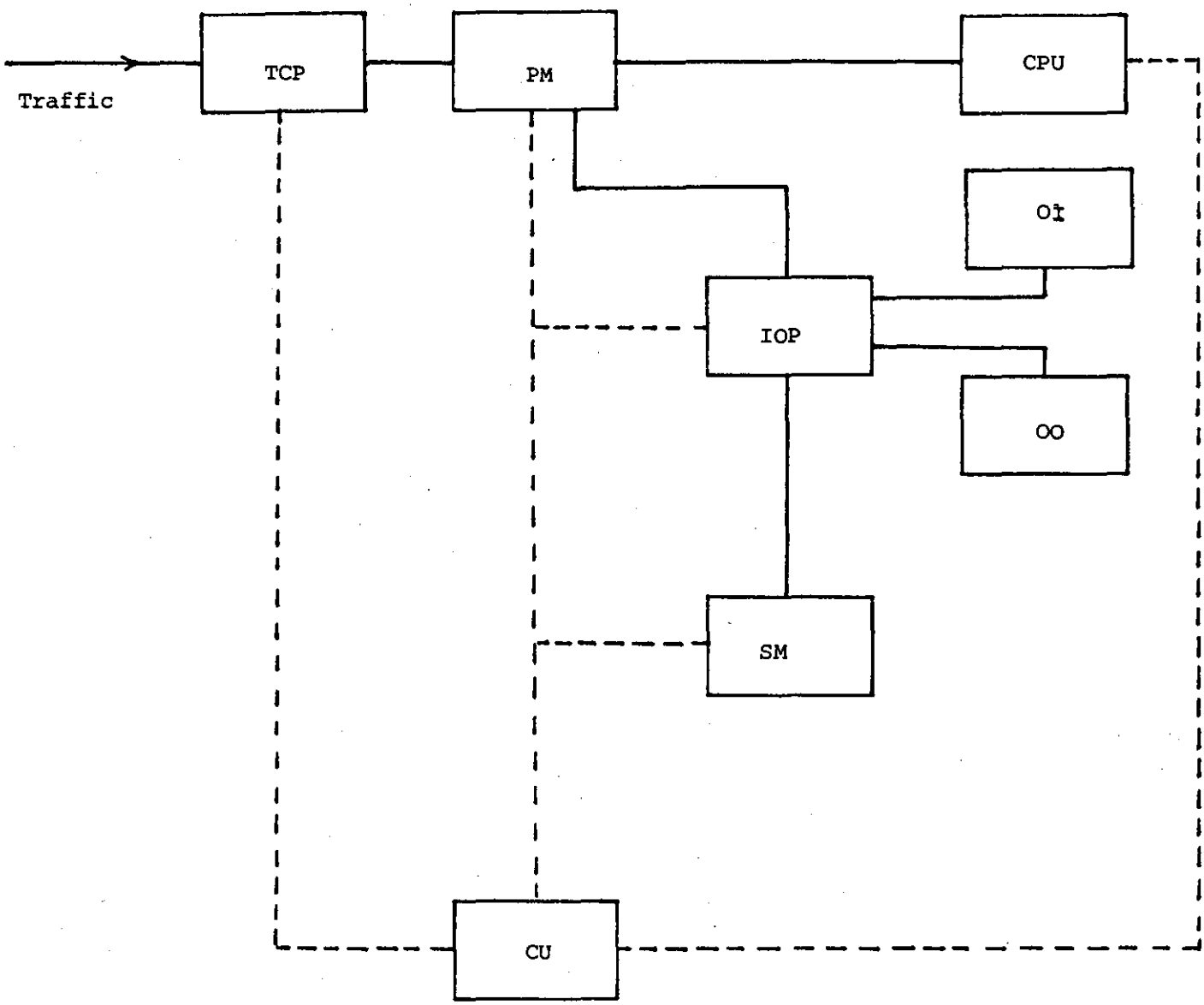
This chapter examines the issues raised above. It attempts to examine the computational component of the total system delay experienced by modules processed in a dual processor CPU cache distributed computation environment. It characterises the system performance in terms of the system response time or load factor and it also examines the effect of the increasing load factor on the decision by the source processors to partition and schedule the modules for computation by the sink processor. Both the analytical and the experimental (simulation) methods of system analysis are used employing queueing-theoretic concepts.

6.2 ANALYTICAL MODEL

The analytical values for the overall performance of the distributed computation system can be estimated by first examining and characterising the sink processor.

6.2.1 The Sink Processor

As explained earlier, the sink processor in the dual processor CPU cache distributed computation environment examined here is a multi-accessed, time-shared and multi-programmed resource system (ref. 6.1), as shown in fig. 6.1. The arriving modules and intermodule packets are first pre-processed at the sink processor resource by the communications preprocessor (packet disassembly, checked for transmission errors, packet identification,



TCP = Traffic Communications Processor

PM = Primary Memory

IOP = Input/Output Processor

OI = Other Inputs

OO = Other Outputs

SM = Secondary Memory

CU = Control Unit

_____ = dab flow path

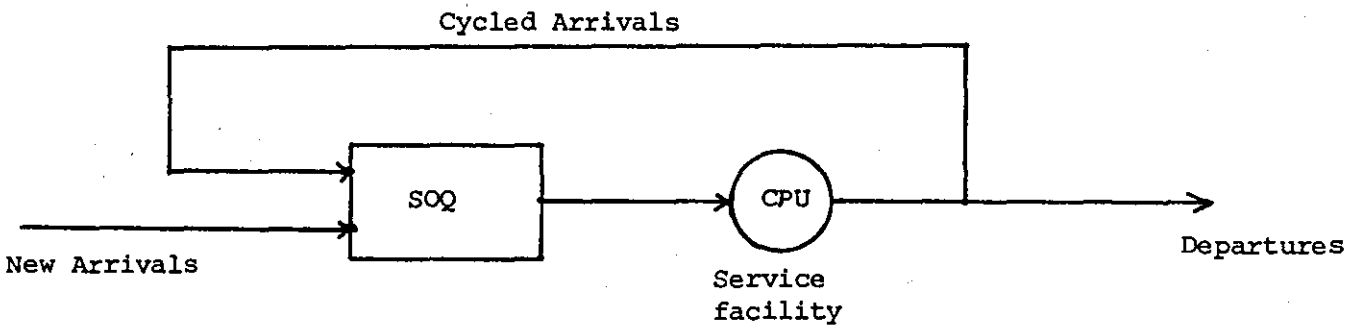
----- -= control flow path

Fig. 6.1 The Sink Processor System

book-keeping) before being loaded into the memory for processing. In this way this channel traffic communications preprocessor is an I/O device that handles the arriving and departing channel traffic. Other I/O devices process and handle the flow of data between the main memory and the secondary devices. The operating system (or the supervisory program) provides the control function by governing the activity and assignment of the various resources of the sink processor system. It is the duty of the supervisory program to resolve the numerous conflicting demands which must arise when the many user programs and modules attempt to access and use these various resources. Often the supervisory program gives each user the impression of having the whole sink system to themselves.

6.2.2 The Sink Processor Model

A time-shared computer system can be viewed as a collection of resources and a population of users who compete at various times for the allocation of these resources. The resulting competitive and conflicting demands placed upon these resources are resolved by the resource scheduler. In this study, the CPU is the most central resource in demand for allocation. One general model for characterising such a central resource in a time-shared system is the feedback queueing model, (ref. 6.2,6.3), shown in fig. 6.2, and consists of a single resource (the CPU) and a system of queues that holds the customers (service requests) waiting for attention by the CPU. These queued requests are serviced by the CPU according to an operational scheduling algorithm or queueing discipline. In this way, a newly arriving request is placed in the system of queues and, when the CPU scheduling algorithm finally permits, is given a turn in the CPU. The request spends a period of time (the service quantum or time-slice) in the CPU. This offered service quantum may or may not be enough to satisfy the original service request. If sufficient, the fully processed request departs from



SOQ = system of queues

Fig. 6.2 The feedback queueing model

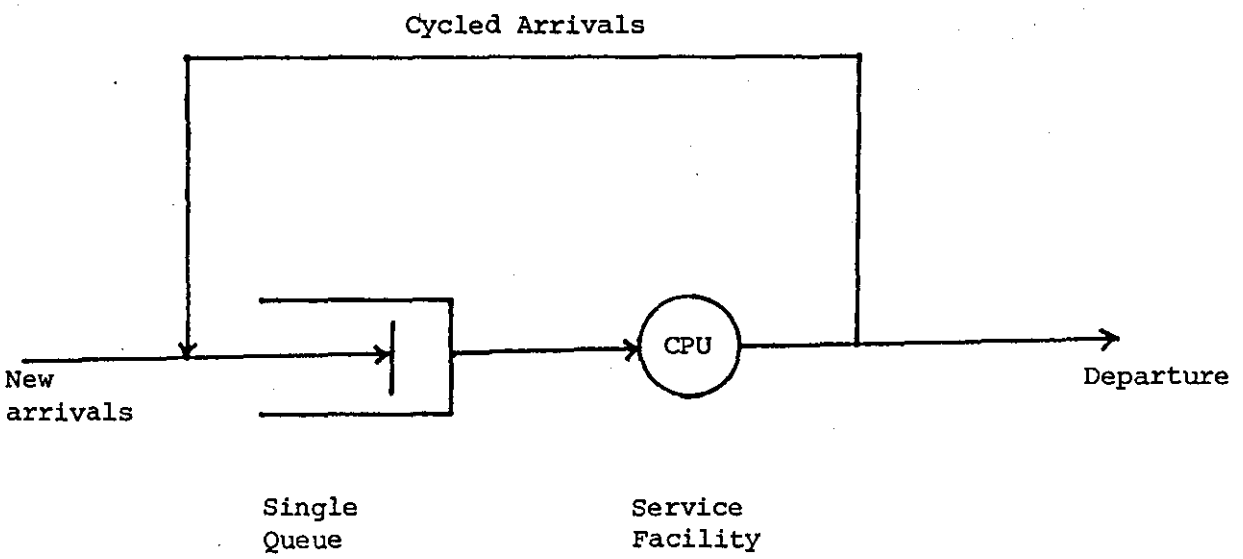


Fig. 6.3 The Round Robin System

the system; else, the partially processed request re-enters the system of queues and waits within this system of queues until the CPU scheduling algorithm decides to offer a second quantum, and so on. Eventually, after a sufficient number of visits to the CPU service facility, the request will have gained enough service and will depart as a fully processed request. In this way preferential treatment is fairly given to short request than to long requests in that a time consuming request will require many visits to the CPU service facility while a small request may require just a few visits to satisfy the request fully. Hence the feedback model is a highly preemptive resume priority queueing discipline (ref. 6.4, 6.5, 6.8).

As mentioned earlier, the allocation of resources to the unpredictable competitive and conflicting demands, which form a queue in front of the resources, often leads to the characterisation of the system in terms of a network of queues or queueing networks. In such queueing networks, the three main parameters necessary to characterise the system performance are the arrival rate, the service rate and the resource scheduling algorithm or the queueing discipline.

6.2.2.1 The System Arrival Rate

The average arrival rate λ, or the inter-arrival time distribution for the module and intermodule data to the sink processor CPU service resource is an important system parameter. The input population to many service facilities, especially the communication facilities, is often taken to be an infinite source and that the arrival process is Poisson (ref. 6.6). The Poisson arrival process provides a good approximation when the nature of the arrival process depends only in a negligible way upon the number of

customers already in the system. The Poisson arrival assumption is often made to simplify calculations but it is also found to produce results that closely agree with measured values. However, in many cases, the system performance can also be estimated in the case in which the input population is finite (ref. 6.7). An inter-arrival time distribution for characterising the average time taken by the users in generating the inter-module data requests during the processing time of the modules at both the source and the sink processors is important in the calculation of the basic system performance measures. In well designed or logically distinct modules, the intermodule reference times may be expected to be few and far between.

6.2.2.2 The System Service Rate

The average service rate μ, or the service time distribution of the CPU service facility is also an important system performance parameter. Factors pertaining to the service facility will in general be important in determining when service may be available, how many customers may be served at a time, and how long the service lasts. Statistical service-time distributions may be specified for the service-times and the negative exponential distribution gives a good approximation in many cases, although branching Erlang distribution may be used if the service is highly variable (ref. 6.9, 6.10, 6.12). In many cases, it is the system equilibrium behaviour that is needed, so that $\mu > \lambda$.

6.2.2.3 The System Queueing Discipline

The queueing discipline or the scheduling algorithm is basically a set of decision rules which relate to how a customer is selected for service from the network of queues. The simplest and most obvious queueing discipline consists of serving customers in order of arrivals, but there

are many other possibilities such as; (ref. 6.4, 6.11).

a) First Come First Serve (FCFS)

b) Last Come First Serve (LCFS)

c) Shortest Processing Time First (SPF)

d) Shortest Remaining Processing Time (SRPT)

e) Shortest Expected Processing Time (SEPT)

f) Shortest Expected Remaining Processing Time (SERPT)

g) Shortest Latency Time First (SLTF)

h) Shortest Seek Time First (SSTF)

The scheduling algorithm may also distinguish the arriving customers according to priority groups so that preferential service is given according to this pre-established internal (or external) priority among groups. Such internal priority may be based on the order of arrival (e.g. LCFS) or on the amount of processing required (Shortest Processing Time). In the case of such priority queueing disciplines, preemptive or non-preemptive priority may be employed depending on whether or not a customer in the process of being served is liable to be ejected from service and returned to the queue whenever a customer with a higher priority arrives in the queue. When the queueing discipline involves preemption, three modes of service may be distinguished depending on how the ejected customer resumes service after having been preempted, (ref. 6.4, 6.11):

a) Preemptive Resume, (PR), in wich service resumes from where it left off.

b) Preemptive Repeat without Resampling, in which service is assumed to start from scratch with the same total service time requirement as the customer had upon his earlier visit.

c) Preemptive Repeat with Resampling, in which service is assumed to start from scratch but with a new service time chosen on reentering service.

In this way, prioritized service may be offered at the sink service facility according to the order of arrival (e.g. LCFSPR) or according to the

amount of service demanded (e.g. The Round Robin). The Round-Robin (RR) processor scheduling algorithm is one of the most widely used algorithms in many multi-accessed time-shared computer systems, (ref. 6.4, 6.11). In the RR scheduling algorithm, fig. 6.3, newly arriving customers join the single queue and work their way up to the head of the queue for service in a FCFS fashion, and then finally receive their quantum of service. When that service quantum expires and if they need more service, then they return to the tail of that same queue and repeat the cycle. The size of the service quantum is important in that a very large quantum in the RR system will make the queueing discipline approach the FCFS while an infinitesimally small service quantum will make the RR performance approach the processor-sharing (PS) system (ref. 6.11). In the PS system a customer makes an infinite number of visits to the CPU service facility, each visit infinitely quickly and each visit receiving infinitesimal service, until finally his attained service equals his required service, at which time he departs. Hence, the PS discipline cannot be actually implemented, but is valuable in modelling RR scheduling. The PS is a reasonable representation of the RR when the quantum is large with respect to the swapping overhead, and small with respect to the average service time (ref. 6.11). Hence the PS is the limiting case of the RR with zero overhead as the quantum goes to zero. The FCFS discipline is often used in many situations in which the service time distribution is exponentially distributed, while the PS (or RR) is often used in those situations where the service time distribution is arbitrary because the effects of high variability (C_x>1) in service times is much less noticeable (ref. 6.10). The PS is often insensitive to all distribution characteristics other than the mean and it gives the same performance as the FCFS with exponential service times.

For I/O devices, service is often offered on a FCFS basis, SLTF (for drum-like devices), or SSTF (for moving head disks), (ref. 6.13, 6.14), but preemption cannot be used.

Hence, depending on the complexity and nature of the operating system at the sink processor system, the CPU service facility provides service to the modules and intermodule data requests in a highly complicated manner, and the system response time (or load factor) can be expected to be a function of many variables. In many cases too, external priority may be imposed on the system by the system administration.

6.2.3 The System Performance Model

From the point of view of the dual processor CPU cache distributed computation environment a composite Markovian queueing network model incorporating the characteristics of both the communication channel and the sink processor system may be used to determine the system response time. With respect to the scheduling and computation of the system modules in such an environment, three main situations may prevail:

a) all modules run at the source processor and none runs at the sink processor

b) all modules run at the sink processor and none runs at the source processor

c) some modules run at either processor

By virtue of the nature of the computation in situation (a), no distribution is involved and hence this does not contribute to the system response time analysis. However, situations (b) and (c) are relevant to the analysis of the distributed computation model because the overall system performance depends on the sink processor system. In situation (b), the effects of the communication delay in the channel are not very

significant, if the communication channel is not very heavily loaded, because the main component of channel delay is only apparent during the sending of the modules to the sink processor system and there is no intermodule communication time incurred during the processing time of the modules. However, the processing of the modules there will increase the prevailing computational workload there and hence the system response time will increase. Situation (c) represents the true arrangement for the CPU cache distributed computation system. In this case the communication channel delay plays a substantial role through the intermodule communication interactions between the two sets of modules at the two processors. Hence, only the source processors corresponding to situations (b) and (c) need be taken into account in the modelling of the dual processor CPU cache distributed computation system for the system response time. This system response time may be defined as the time interval since the sending of a computational request by the source (sink) processor to when that request is received from the sink (source) processor in the system. Hence the total system response time will involve both the communication delay, as explained in the previous chapter, and the computation delay. These two main components of the system delay can be considered to be independent. The computation component of delay can be obtained by characterising the sink processor system computation workload or load factor.

The sink processor system can be modelled as a Markovian queueing network, (ref. 6.7). Many Markovian queueing networks may be modelled as open, closed or mixed networks, (ref. 6.15).

6.2.3.1 Open Networks

In open network models of queueing systems it is usually found reasonable to assume that there is an infinite external source for the customers (module packets, requests) arriving to the network (ref. 6.16, 6.17). Exponential or other interarrival time distributions and various queueing disciplines may then be employed to model the system performance, (ref. 6.12). As well as having a source for the arriving customers, there is also a sink for the departing customers. The assumption of the infinite population of customers is a valid assumption in many communication system models where the number of sources for the customers (module packets) may be very large (ref. 6.16, 6.17), as was done in the analysis of the communication channel delay. In such open networks the value of the queue length and delay can range from zero to infinity and no restriction is imposed on the network job (module) population. However, the infinite source assumption is not usually reasonable in many computer system models because, in this case, there is usually some finite resource which acts as a bottleneck and thus limiting the total population of the customers (modules) in the network. In a dual processor CPU cache distributed computation system, the total population of modules may be limited by the contention for the sink processor system memory. Furthermore, the total population in such a system will be discouraged by a large computational workload at the sink processor.

6.2.3.2 Closed Networks

In a closed network model, there is a constant number of jobs (modules) at all times, (ref. 6.7, 6.9). The jobs (modules) neither enter nor leave the closed network. The number of these jobs in a closed network is called the job population. Hence, the total number of jobs in the network must always equal the network population, the sum of the queue lengths of all jobs in the network, (ref. 6.9).

A dual processor CPU cache distributed computation system can be modelled as a multi-accessed, time-shared system as shown in fig. 6.4. The system model shows a closed network model with job population N. In this model the job population N refers to the number of source processors that are involved in the distributed computation system as described by the situations (b) and (c) above. During processing, a module processing at the sink processor may be considered to interact with the relevant source processor module during the intermodule communication. In this way, the job population N may be taken to refer to the number of inter-module pairs involved in the intermodule communication during the computation of one module at either processor. From time to time, intermodule parameters and data are demanded by either processor in the dual processor CPU cache distributed computation environment during the computation process life-time. As explained earlier, the intermodule data traverses the LAN communications subnet interface in the form of packets. It may be assumed that the average time interval between the intermodule data exchange and transfer is long enough to guarantee equilibrium system performance, (ref. 6.7). Fig. 6.4 models the closed network with two main queues which form in front of the two main resources, the communication channel facility, and the computation (CPU) facility at the sink processor. This is modelled as a feedback queueing system in which each intermodule request can be considered to cycle through the various resources until the request is satisfied and finally returns to its origin. The system response time may be considered to be measured at point X. Fig. 6.5 models the system with the sink processor system modelled as a central server, (ref. 6.18), in which a request leaving the CPU may proceed to one of

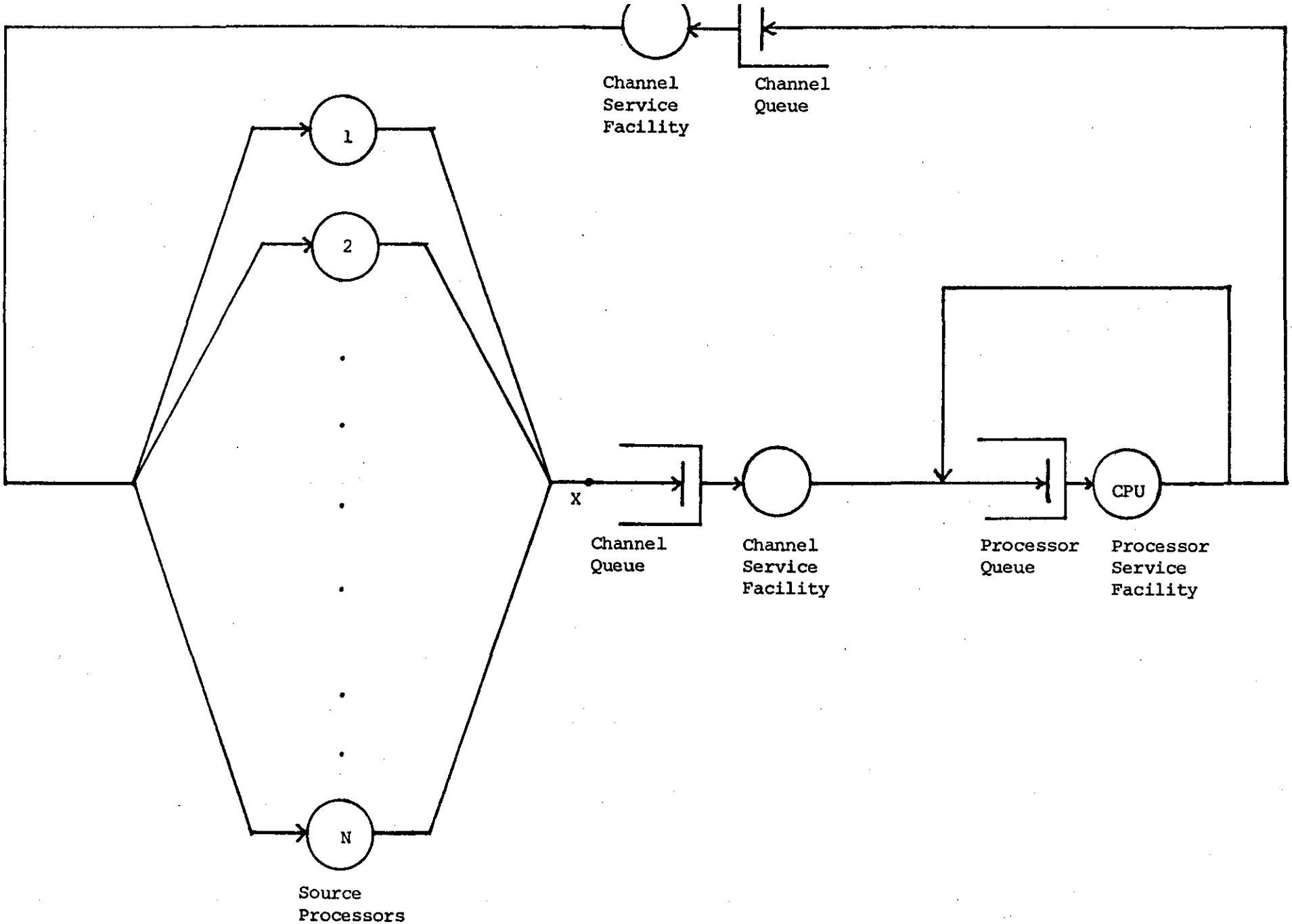


Fig. 6.4 System Model

two I/O devices, and the request leaving the I/O device may, in turn, either cycle back and return to the CPU for further processing, or it may return back to the relevant source processor. Fig. 6.5 illustrates also the concept of branching probabilities; the probability of a request making a transition from queue i to queue j in a routing chain (ref. 6.18, 6.19, 6.20).

Due to the nature of the dual processor CPU cache distributed computation system, the network job population N is not truly constant over all time. In general N will fluctuate with time as some of the users complete their computations while some of the other users start their computations at arbitrary times. However, it is preferable to model the system as a closed network with a fixed job population N, and analyse the behaviour of the model for different values of N (ref. 6.18). In this way, the system performance measures such as the system response time, load factor, throughput, and utilization may be determined as a function of N. Such a method of analysis is equivalent to characterising the system in terms of several equilibrium models: one for each value of N. The alternative is to analyse just a single model which incorporates the transient behaviour of the system with N. Such a method may be difficult to analyse or it may yield results which are too cumbersome to be directly or easily usable.

Figs. 6.4 and 6.5 show how the overall system performance may be modelled as a chain network in which a job of any class i can become a job of any other class j, possibly after making intermediate transitions to other classes k_1, k_2, ..., with non-zero probability (ref. 6.19, 6.20). Such queueing networks have product form solutions which render the

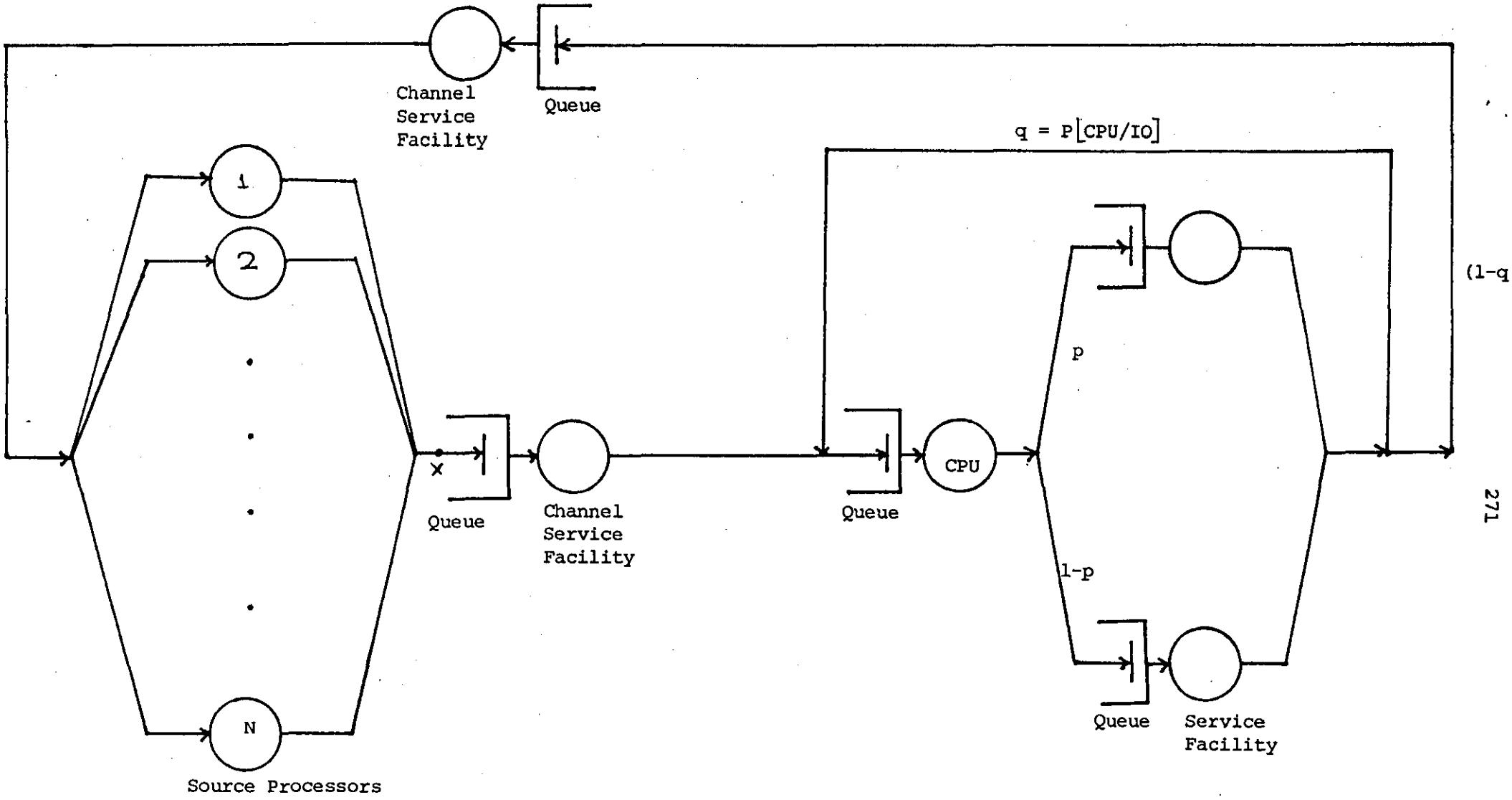


Fig. 6.5 System Model

queueing networks to be mathematically tractable (ref. 6.18, 6.21). Most useful mathematically tractable queueing network models have product form solution. Some open, closed and mixed networks have product form solutions (ref. 6.15). But for closed networks, the product form solution is more difficult conceptually and computationally than the one for open networks, because of the stronger interactions between the various queues due to the fixed number of jobs (ref. 6.9, 6.18). In a network of N queues, the basic product form is expressed as (ref. 6.21, 6.22).

$$P\left[\vec{K}_1, \vec{K}_2, \ldots \vec{K}_M\right] = \frac{X\left[\vec{K}_1\right] . X\left[\vec{K}_2\right] \ldots X\left[\vec{K}_N\right]}{G\left[\vec{N}\right]}$$

where the vectors have elements corresponding to the different routing chains, and

$P\left[\vec{K}_1, \vec{K}_2 \ldots \vec{K}_N\right]$ = the probability of $\vec{K}_1$ jobs at queue 1, $\vec{K}_2$ jobs at queue 2, ... $\vec{K}_N$ jobs at queue N,

$X\left[\vec{K}_i\right]$, for i = 1, 2, ... N, is a factor determined from the probability of $\vec{K}_i$ jobs at queue i in isolation (in an M/M/1 queue),

$\vec{N}$ = vector of number of jobs in the routing chains,

and $G\left[\vec{N}\right]$ = normalization constant

For an N-node closed network with population n, the joint distribution of finding customers in the various nodes is given by the product form solution, (ref. 6.22)

$$P\left[k_1, k_2, \ldots k_N\right] = \frac{1}{G[n]} \cdot \prod_{i=1}^{N} \frac{r_i^{k_i}}{Q\left[k_i\right]}$$

where the set of numbers $\{r_i\}$ satisfy the set of linear equations given by

$$\mu_i . r_i = \sum_{j=1}^{N} \mu_j . r_j . p_{ij}, \text{ for } i = 1, 2, \ldots N$$

where

p_{ij} = probability of a customer next proceeds to node j when he completes service at node i

μ_i = the mean of the exponentially distributed service time of the i^{th} node which consists of a single queue served by m_i servers, (each with μ_i)

and the normalization constant

$$G[n] = \sum_{\vec{k} \in A} \prod_{i=1}^{N} \frac{r_i^{k_i}}{Q[k_i]}$$

where $\vec{k} = (k_1, k_2, \ldots k_N)$ and A is the set of vectors $\vec{k}$ for which

$$k_1 + k_2 + \ldots k_N = n$$

and where

$$Q[k_i] = \begin{cases} k_i! & , \text{ for } k_i \leq m_i \\ m_i! \cdot m_i^{k_i - m_i} & , \text{ for } k_i \geq m_i \end{cases}$$

In closed networks, the queues must be considered collectively because of the interactions between them. However, it is not necessary for a computational algorithm to recognise the explicit product form, (ref. 6.23, 6.24).

6.2.3.3 The Time-Shared Model Performance

The system performance may be analysed by considering fig. 6.4 in which the N users (the source processors in the system) make unpredictable demands upon the time-shared sink processor system. The model performance can be described as follows: whenever a ready source processor user makes a computational request for service at the sink processor, the request

proceeds to receive service according to the operational scheduling algorithm there. During this time the source processor user is busy processing own computational workload, wherever possible, and does not generate any new requests. This request may be a demand for intermodule data and parameters from the remote sink processor. Conversely, the remote sink processor may demand intermodule data and parameters from the source processor, in which case the source processor must suspend own computation and service the interrupt for service. When finally that request is completely serviced at the sink processor, the response is retransmitted back over the communication interface to the respective source processor. The time interval taken by each source processor in generating each new request may be referred to the "intermodule data request time". In this way alternating periods of intermodule data request time and processing take place throughout the life-time of the scheduled modules.

The performance of the model may be examined by assuming that the intermodule data request time for each source processor is exponentially distributed with mean b_1 seconds. If $n \to \infty$ as $1/b_1 \to 0$ so that $n/b_1 =$ a constant, then the system may be modelled as a Poisson arrival process, with average rate $1/b_1$, to characterise this finite population model with n users. At the sink processor system, the processor scheduling algorithm may be assumed to be PS, FCFS or LCFSPR. When the service time at the sink processor is assumed to be exponentially distributed (rate μ), and with FCFS scheduling algorithm, then this is a finite M/M/1 system in which the probability of finding k customers (module requests) is given by, (ref. 6.7, 6.25)

$$P[k] = \begin{cases} P[0].(\lambda_1/\mu)^k.n!/(n-k)!, & \text{for } 0 \leq k \leq n \\ 0 & \text{, for } k > n \end{cases}$$

where

$$P[0] = \left[\sum_{i=0}^{n} \left(\frac{\lambda_1}{\mu} \right)^i . \, n!/(n-j)! \right]^{-1}$$

and

$$\lambda_1 = 1/b_1$$

and $1/\mu = b_2$ = the mean value of the exponentially distributed service time at the sink processor. From the memoryless property of the exponential distribution, it may be seen that the distribution for number in the system must be independent of the scheduling algorithm of the sink processor. By considering this model with exponential service time and examining the rate at which jobs enter into and depart from the sink processor system (CPU), the mean system response time T(n) of the system is given by, (ref. 6.25)

$$T(n) = \frac{n.b_2}{1-P[0]} - b_1 \qquad (1)$$

where

$$P[0] = \frac{1}{\sum_{i=1}^{n} (\lambda_1/\mu)^i.n!/(n-i)!}$$

$$= \frac{1}{\sum_{i=1}^{n} \left(\frac{b_2}{b_1} \right)^i \cdot \frac{n!}{(n-i)!}}$$

From (1) it may be seen that the minimum value of $T(n) = b_2$, and this occurs when $n = 1$, so that $T(1) = b_2$. Hence, normalizing (1) with respect to this minimum value

$$\frac{T(n)}{T(1)} = F(n) = \frac{n}{1 - P[0]} - \frac{b_1}{b_2} \qquad (2)$$

then the behaviour of F(n) may be examined as a function of n. Fig. 6.6 shows this variation of F(n) with n and it may be seen that F(n) rises very slowly with n, at first, until a critical value of n is reached when F(n) rises more rapidly and linearly with n, (ref. 6.25). In the region where F(n) rises slowly with n, the number of the interacting source processor users is small so that the periods when a customer needs service may be thought of as the periods when the other system users are doing their own processing and therefore not interfering with the one who is being served at the sink processor system. On the other hand, after the critical value of n, F(n) rises linearly with unity slope, and the system behaviour may be considered to show some form of saturation, (ref. 6.25, 6.26). The critical value of n, n_s, is given by (ref.6.25).

$$n_s = \frac{b_1 + b_2}{b_2} = 1 + \frac{b_1}{b_2} \qquad (3)$$

so that each user beyond n_s causes all other users to be delayed by an amount of time equal to his entire processing time of b_2 seconds. The linear relationship of F(n), beyond n_s, is given by

$$F(n) = n - \frac{b_1}{b_2} \qquad (4)$$

since, for large n, $P[0] = 0$

By extrapolating this linear asymptote for F(n), in fig. 6.6, back to meet the horizontal asymptote defined by

$$F(n) = 1 \qquad \text{or} \qquad F(1) = 1$$

it may be seen that the two asymptotes meet at the critical point defined by the value of n_s, so that the linear asymptotic rise may be expressed

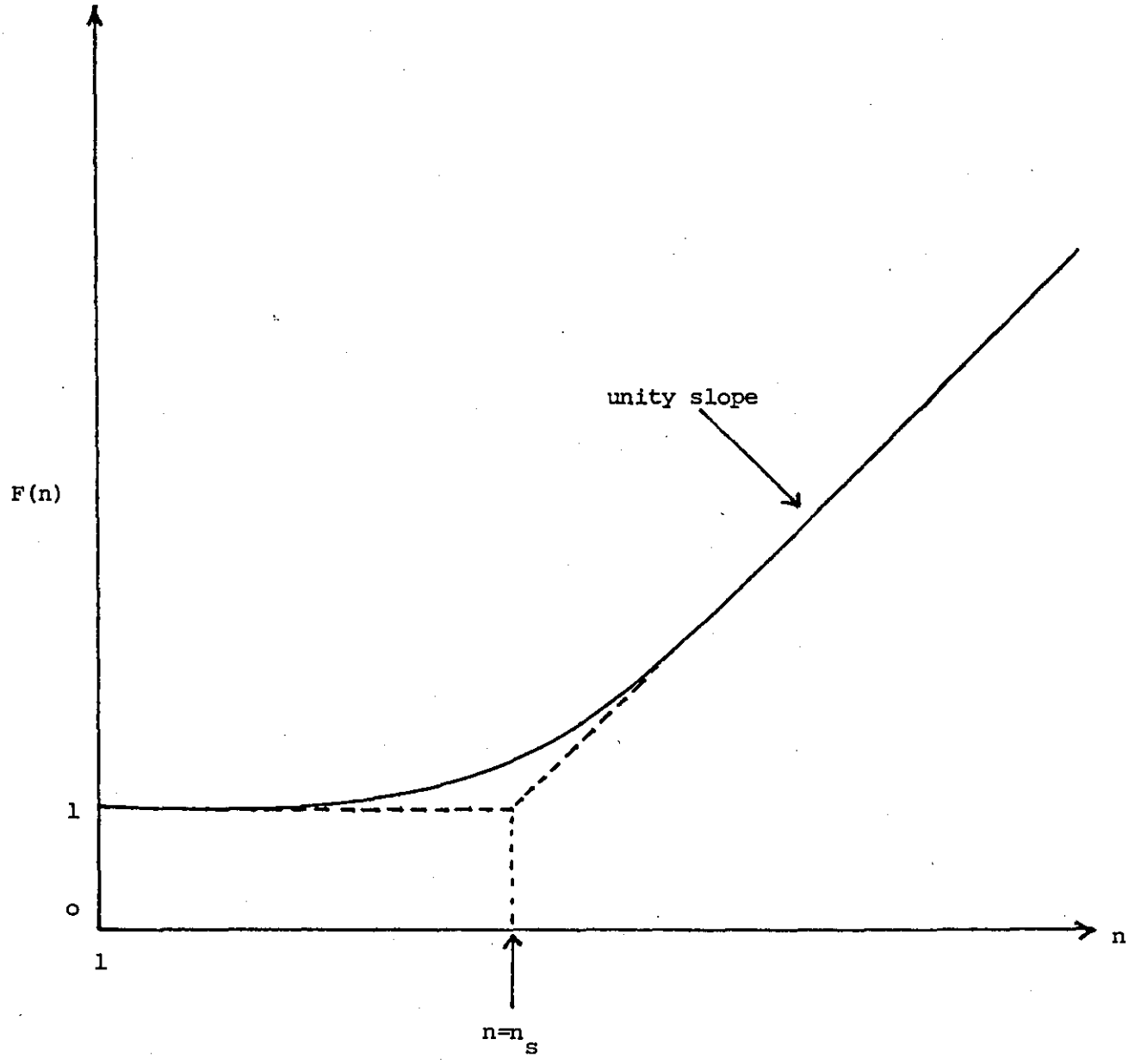


Fig. 6.6 Asymptotic behaviour of F(n) and T(n)

as, (ref. 6.25, 6.26)

$$F(n) = n - n_s + 1 \qquad (5)$$

As mentioned earlier, fig. 6.4 models the performance of just a single CPU resource at the sink processor system. The analysis and performance of both general Markovian open queueing networks (ref. 6.16, 6.17, 6.22), and general Markovian closed networks (ref. 6.9, 6.24), have been extensively studied. The analysis of the closed Markovian queueing networks suitable for modelling the performance of multiple resource computer systems in which each resource is modelled as a network node has also been carried out and studied extensively (ref. 6.27). However, when such analysis for the general Markovian open and closed network models is used for characterising the multiple resource computer system and if the product form solution must be retained, three basic limitations are generally encountered (ref. 6.23).

a) the queue discipline is FCFS

b) all service time distributions are exponential

c) all customers are assumed to behave identically so that service times and transition probabilities are drawn from the same distribution for each.

However, the PS, and the LCFSPR scheduling algorithms at the CPU have been shown to remove some of these limitations in that they have a mean conditional response time that is independent of the service time distribution in both the open and closed networks (ref. 6.11, 6.24). Furthermore, the limitations have also been removed in the case in which different classes of customers in a closed system are used, (ref. 6.18, 6.28), and also in closed networks in which the different customers are allowed

different transition probabilities, as well as their own set of exponentially distributed service times, (ref. 6.27).

Hence, by considering the time-shared system with n users sharing a multiple resource sink processor system with N resources, the asymptotic behaviour of such a closed system may be examined (ref. 6.25, 6.26). Arbitrary distributions of service at each node are permitted (ref. 6.25). Let

$b_i = \frac{1}{\mu_i}$ = mean service time at node i

m_i = the number of servers at node i

$\lambda_1 = \frac{1}{b_1}$ = rate at which jobs are generated by each of the n source processor users. The mean intermodule data request time, b_1, may be drawn from an arbitrary distribution.

$T(n)$ = average response time to pass the sink processor multiple resource system when there are n users in the system

Hence, the average cycle time, the sum of the average intermodule data request time plus the average service time, is

$$T(n) + b_1, \text{ seconds} \qquad (6)$$

so that the system throughput, $R(n)$, is given by

$$R(n) = \frac{n}{T(n) + b_1}, \text{ customers/second} \qquad (7)$$

If

$\overline{N}$ = average number of jobs in the multiple resource sink processor system

$\overline{n}$ = average number of intermodule data requests at the source processors

then, by the application of Little's result (ref. 6.29),

$$T(n) = \frac{\overline{N}}{R(n)} \qquad (8)$$

or, since

$$n = \bar{n} + \bar{N}$$

then

$$T(n) = \frac{n}{R(n)} - \frac{\bar{n}}{R(n)} \qquad (9)$$

Applying Little's result to the source processor yields

$$b_1 = \frac{1}{\lambda_1} = \frac{\bar{n}}{R(n)} \qquad (10)$$

so that

$$T(n) = \frac{n}{R(n)} - b_1 \qquad (11)$$

If the relative utilization of the i^{th} node is defined as r_i/m_i, and by considering the limit as $n \to \infty$, then an infinite queue will form at the bottleneck node and saturate it (so that saturation value of the largest relative utilization of the node becomes r_s/m_s) while only finite queues exist at the other nodes, (ref. 6.27, 6.25). The extent to which the i^{th} node creates this bottleneck effect (or saturation) is defined (ref. 6.24), as being proportional to the rate of change of throughput with respect to an increase in the service rate of that node: the throughput being defined as the average number of jobs processed per unit time. Hence, by using these arguments it can be shown that the asymptotic behaviour of the system response time, T(n), is given by, (ref. 6.24, 6.25, 6.26. 6.30)

$$T(n) \simeq \frac{n.r_s}{m_s.\mu_N.r_N} - b_1, \text{ for } n >> n_s, \qquad (12)$$

in which $\mu_N.r_N$ is the relative number of visits a job makes to the N^{th} node in passing through the rest of the network (the sink processor system). Hence the subscript N corresponds to the source processor node (node 1). Furthermore, the average number of times a bottleneck node is visited for each entry into the sink processor system, i.e. between each visit to the

source processor node, is given by, (ref. 6.25, 6.30)

$$\frac{\mu_s . r_s}{\mu_N . r_N} \tag{13}$$

When the bottleneck node is deeply saturated, so that $n >> n_s$, the output rate, $R_s(n)$, from the saturated node is given by

$$\mu_s . m_s \tag{14}$$

so that the output rate (throughput) of customers from the sink processor system, is obtained from (13) and (14), as

$$R(n) = \frac{\mu_s . m_s}{\left[\frac{\mu_s . r_s}{\mu_N . r_N}\right]}$$

$$= \frac{m_s . \mu_N . r_N}{r_s} \tag{15}$$

From (12), the asymptotic behaviour of T(n) may be examined, fig. 6.7, and the linear asymptote of T(n) beyond the saturation value $n = n_s$ has slope, given by

$$\frac{r_s}{m_s . \mu_N . r_N} \quad , \quad \text{for } n >> n_s \tag{16}$$

The minimum value of T(n) is

$$\frac{n_s . r_s - m_s . r_N}{m_s . \mu_N . r_N} \tag{17}$$

Since, for $n >> n_s$, each additional user causes all other users to be delayed by his entire average service time, it may be seen that the saturated system behaves like a deterministic system (ref. 6.25, 6.30). Hence, by considering that

n_s = maximum number of perfectly scheduled jobs, in a deterministic system, that cause no mutual interference

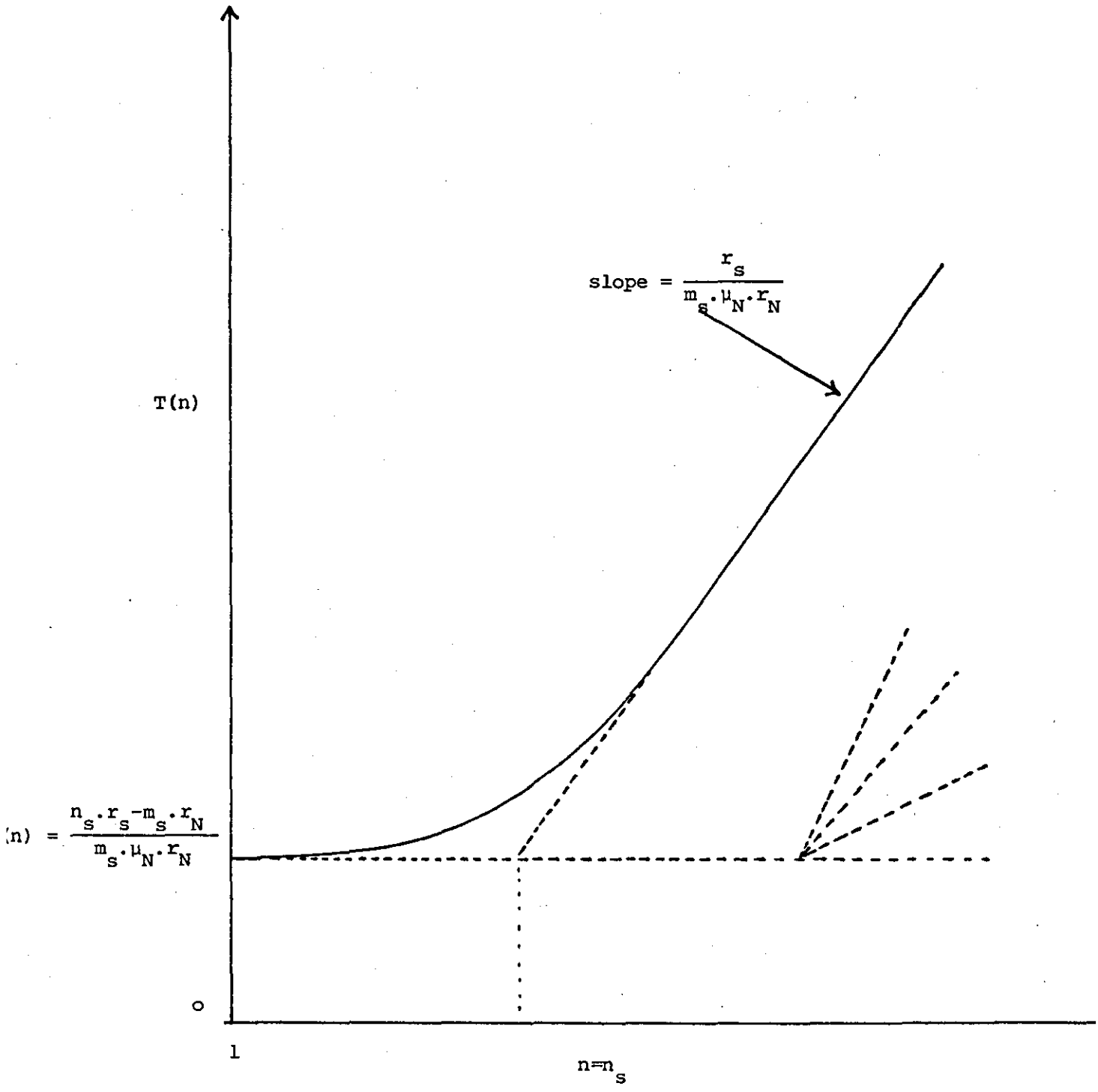


Fig. 6.7 Asymptotic behaviour of T(n) and F(n)

then, the horizontal asymptotic behaviour of T(n) may be estimated. The value of n_s may be estimated since, for each of the m_s servers in the saturated node, the maximum number of jobs that can be scheduled is equal to the service required by a job in each cycle divided by the service time spent by a job in the saturated node per cycle. Hence, considering all the m_s servers, then

$$n_s = \frac{m_s \cdot \sum_{i=1}^{N} \frac{\mu_i \cdot r_i}{\mu_N \cdot r_N} \cdot \frac{1}{\mu_i}}{\frac{\mu_s \cdot r_s}{\mu_N \cdot r_N} \cdot \frac{1}{\mu_s}} \qquad (18)$$

so that,

$$n_s = \frac{m_s}{r_s} \cdot \sum_{i=1}^{N} r_i \qquad (19)$$

From (18) the average cycle time (or the service time in a cycle, no queueing since n=1), is given by

$$\sum_{i=1}^{N} \frac{\mu_i \cdot r_i}{\mu_N \cdot r_N} \cdot \frac{1}{\mu_i}$$

or

$$\frac{n_s \cdot r_s}{\mu_N \cdot r_N \cdot m_s} \qquad (20)$$

Equating (6) and (20), then the horizontal asymptotic behaviour of T(n) may be given by

$$T(n) = \frac{n_s \cdot r_s}{\mu_N \cdot r_N \cdot m_s} - b_1 \qquad (21)$$

In this way several linear asymptotes with different values of n_s and slope may be obtained depending on which node saturates first (ref. 6.30, 6.25), as shown in fig. 6.7.

The performance analysis of more general open, closed and mixed multiple resource networks of queues that permit different classes of

customers, routing chains, scheduling disciplines and general service time distributions by the method of local balance in computer systems have been extensively studied (ref. 6.22, 6.31, 6.10, 6.15). Most of this study has concentrated on networks with product form solutions. In the use of the local balance methods in the solution of these generalised models, the Markovian characterisation of the service time distribution of these networks is preserved by the use of stage-type servers (ref. 6.32). Generalized queueing models for the multiprogrammed computer systems have been solved similarly (ref. 6.7, 6.24, 6.32). The major goals of the multiprogrammed computer systems are similar to that of the time-shared multi-accessed systems in that a number of jobs are permitted to gain simultaneous access to the resources of the system in such a way that the CPU is allowed to be busy processing one job while various I/O peripheral devices are processing some of the others concurrently. A central-server model of a computer system permits the inclusion of a number of peripheral devices. Fig. 6.5 shows a time-shared central-server model with two peripheral I/O devices.

6.2.3.4 Computational Algorithms

The traditional approach to the solution of the general Markovian queueing networks was to formulate a system of algebraic equations (balance equations) for the joint probability distribution of the vector-valued system state, as explained earlier. But it was later found that for certain types of networks, the solution of these balance equations is in the form of a product of simple terms, and that these products could then be normalized numerically to form a proper probability distribution, (ref. 6.21, 6.22, 6.23). However, in the case of networks with closed

routing chains, this normalization was found to be computationally limited, (ref. 6.24). But this difficulty was overcome by the use of computational algorithms as it is not necessary for the computational algorithm to recognize the explicit product form (ref. 6.23, 6.24). The four main types of computational algorithms are

a) The Convolution Algorithm, (ref. 6.24, 6.33)

b) The Mean Value Analysis Algorithm (ref. 6.34)

c) The Local Balance Algorithm for Normalizing Constants (ref. 6.23)

d) The Algorithm to Coalesce Computation of Normalizing Constants, (ref. 6.23).

A number of criteria may then be employed in choosing a computational algorithm for queueing network models. Such criteria include

a) generality

b) asymptotic computational complexity

c) asymptotic space complexity

d) numerical stability

e) implementation effort

All the four computational algorithm exhibit various advantages and disadvantages with respect to the above performance criteria. While the performance of the Mean Value Analysis Algorithm (MVAA) may be shown to be asymptotically equivalent to the others, its program implementation is often simpler (ref. 6.34). The MVAA starts off by recognizing that the joint distribution contains far too much detail even in situations in which much simpler quantities such as the mean queue sizes, mean queueing times, mean resource utilization and throughput only are needed. Hence the major goal in the application of MVAA computation algorithm

is to obtain the mean value performance measures associated with the queueing system. The MVAA for a closed network such as the one modelled by fig. 6.5 with a single routing chain and allowing a number of job classes, can be sketched (ref. 6.23, 6.34).

Let

M = number of queues (nodes) in the network

C = number of job classes. The classes are partitioned among the queues, with at least one class per queue

S_m = the set of classes belonging to queue m. Queue m has FCFS scheduling algorithm and exponential service time distribution, with mean b_m at each of the classes

p_{ij} = probability a job departing from class i next joins class j

The value of the relative throughputs at each of the classes is given by the set of linearly dependent equations

$$r_j = \sum_{i=1}^{C} r_j \cdot p_{ij}, \text{ for } j = 1, 2, \ldots C \qquad (22)$$

Hence, if R_j is the throughput at class j, then the throughput at class k is given by

$$R_k = \frac{r_k}{r_j} \cdot R_j \qquad (23)$$

Let

r_m = the relative throughput of queue m

$$r_m = \sum_{\text{all } j} r_j \qquad (24)$$

N = job population in the network

$L_m(n)$ = the mean queue length at queue m when there are n jobs in the network

$Q_m(n)$ = the mean queueing time at queue m when there are n jobs in the network

From fig. 6.5 it may be seen that the network may be modelled as single server queues and infinite server queues. According to MVAA, for networks with single server and infinite server queues, the various system performance measures may be determined from the mean values, and without the need for considering the probabilities of the network states or the marginal probabilities, (ref. 6.33, 6.34). For single server queues, it may be shown that the mean queueing time $Q_m(n)$ can be defined recursively as follows, (ref. 6.34)

$$Q_m(n) = b_m(1 + L_m(n-1)) \tag{25}$$

while for the infinite server queues

$$Q_m(n) = b_m \tag{26}$$

Given the mean queueing time, the mean queue length may be obtained by the application of Little's result, (ref. 6.29), and throughput. The value of the throughput may be obtained by applying Little's result to the mean cycle time. The mean cycle time defines the mean time between visits to a queue, and is given by, (ref. 6.34)

$$\sum_{i=1}^{M} \frac{r_i}{r_m} \cdot Q_i(n) \tag{27}$$

so that

$$n = R_m(n) \cdot \sum_{i=1}^{M} \frac{r_i}{r_m} \cdot Q_i(n) \tag{28}$$

in which the job population n of the network is used as the queue length in Little's result. From (28) the throughput is given by

$$R_m(n) = \frac{n}{\sum_{i=1}^{M} \frac{r_i}{r_m} \cdot Q_i(n)} \qquad (29)$$

By using these recursive equations and the initial condition that $L_m(0) = 0$, then for m = 1, 2, ... M, the mean value system performance measures may be calculated from the MVAA computation algorithm, which may be sketched as follows, (ref. 6.33, 6.34)

For n = 1 to N

For m = 1 to M

If Queue m is single server

Then (* queue m is single server *)

$$Q_m(n) = b_m \ (1 + L_m(n-1))$$

Else (* queue m is infinite server *)

$$Q_m(n) = b_m$$

(* end loop on m*)

For m = 1 to M

$$R_m(n) = \frac{n}{\sum_{i=1}^{M} \frac{r_i}{r_m} \cdot Q_i(n)}$$

$$L_m(n) = R_m(n) \cdot Q_m(n)$$

(*end loop on m*)

(* end loop on n *)

From such a computational algorithm the queueing model performance measures such as the mean throughput, queue lengths and queueing time may be determined for each queue. Fig. 6.5 is modelled with six queues and the above computational algorithm may be used to determine the mean performance measures. From the values of the mean queueing time $Q_m(n)$

and the mean cycle time, the system response time, T(n) may be obtained. If q is the probability that a job returns to the CPU for more service after leaving an I/O device, (i.e. $P[CPU/IO]$), then the number of the CPU-I/O cycles has a geometric distribution, starting at one, with a mean r_{CPU} given by

$$r_{CPU} = \frac{1}{1-q} \text{ cycles}$$

The computational algorithm may be applied to the network model of fig. 6.5 by letting

queue 1 = the infinite server queue

queue 2 = the single server transmission channel queue

queue 3 = the single server sink system CPU queue

queue 4 = the single server sink system I/O (floppy) disk queue

queue 5 = the single server sink system I/O (hard) disk queue

queue 6 = the single server transmission channel queue

Hence the mean system response time, T(n), measured at point x, may be calculated from

$$T(n) = r_1 \cdot Q_1(n) + r_2 \cdot Q_2(n) + r_3 \cdot Q_3(n) + r_4 \cdot Q_4(n) + r_5 \cdot Q_5(n) + r_6 \cdot Q_6(n)$$

$$= \sum_{i=1}^{6} r_i \, Q_i(n) \qquad (30)$$

If $r_1 = 1$, then

$r_2 = 1$

$r_3 = r_{CPU} = \frac{1}{1-q}$

$r_4 = \frac{P}{1-q}$

$r_5 = \frac{1-P}{1-q}$

$r_6 = 1$

T(n) is the mean time taken since the issue of a request by a source processor to the time the response is finally obtained from the sink processor system when there are n jobs in the network. An alternative characterisation of T(n) is the sink processor system load factor F(n) which may be defined similarly to (2) as a dimensionless quantity as

$$F(n) = \frac{T(n)}{T(1)} \tag{31}$$

where

$$T(1) = \sum_{i=1}^{6} r_i . Q_i(1)$$

so that the minimum value of F(n) is F(1) = 1.

Hence both T(n) and F(n) may be used to characterise the system performance as the system workload varies. In a dual processor CPU cache distributed computation system, it is necessary that T(n), (or F(n)) characterises the variation of this workload. A small value of T(n) or F(n) means that distributed computation is feasible because the reserve capacity of the crunching power of the sink processor system is still available and also the effects of the intermodule communication times are not too high. Hence the values of T(n) and F(n) contain both the communications and the computational delay components of the system delay performance. The communications component of delay was dealt with in Chapter 5 and can be omitted from the model. Furthermore, these two components of delay are largely independent. It was seen in Chapter 5 too that under low traffic conditions, the communications component of delay is very small. Hence, the contributions to F(n) and T(n) from queues 2 and 6 may be omitted in the computational algorithm so that F(n) and T(n) characterise the computational workload only at the sink

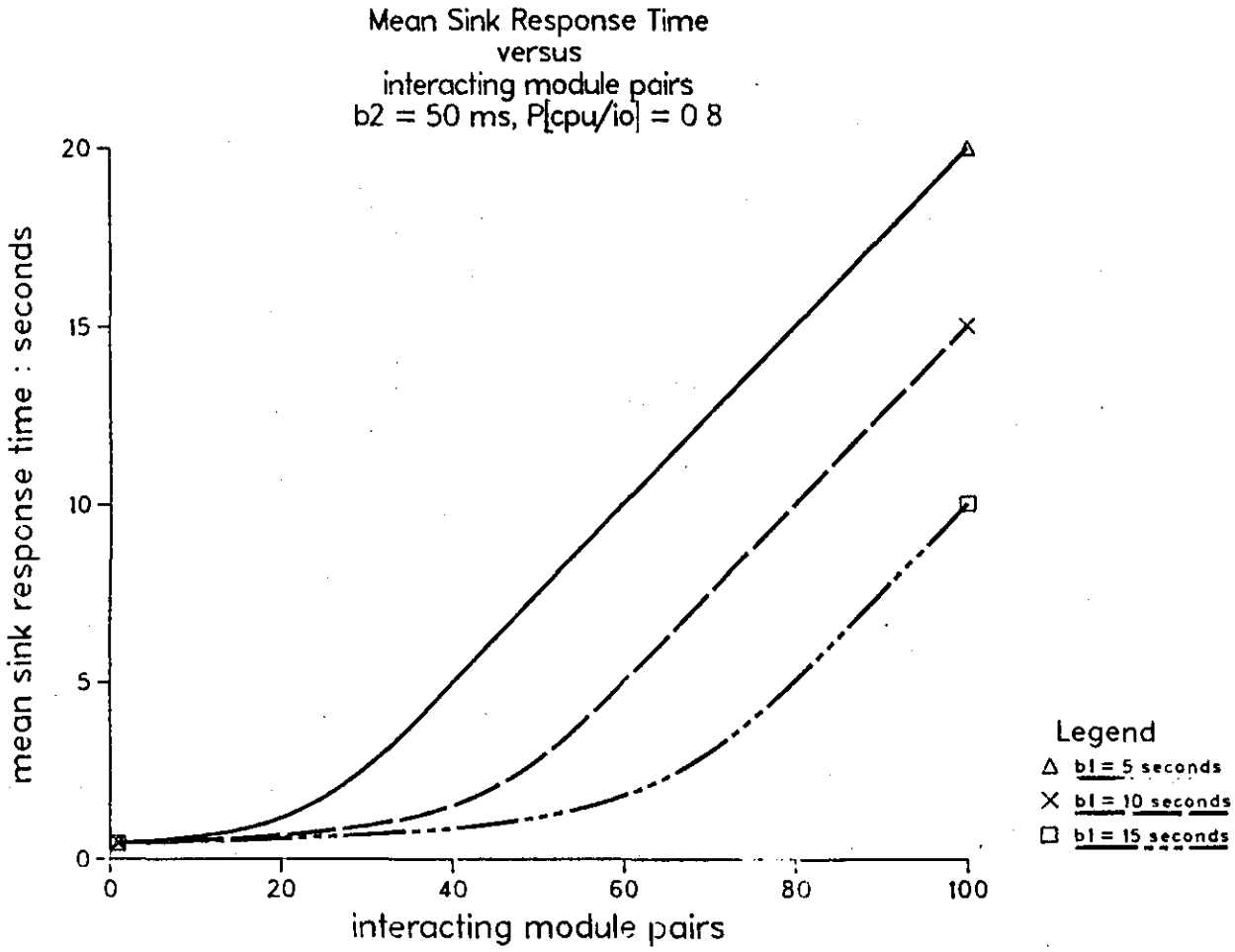


Fig. 6.8 Mean Sink Response Time

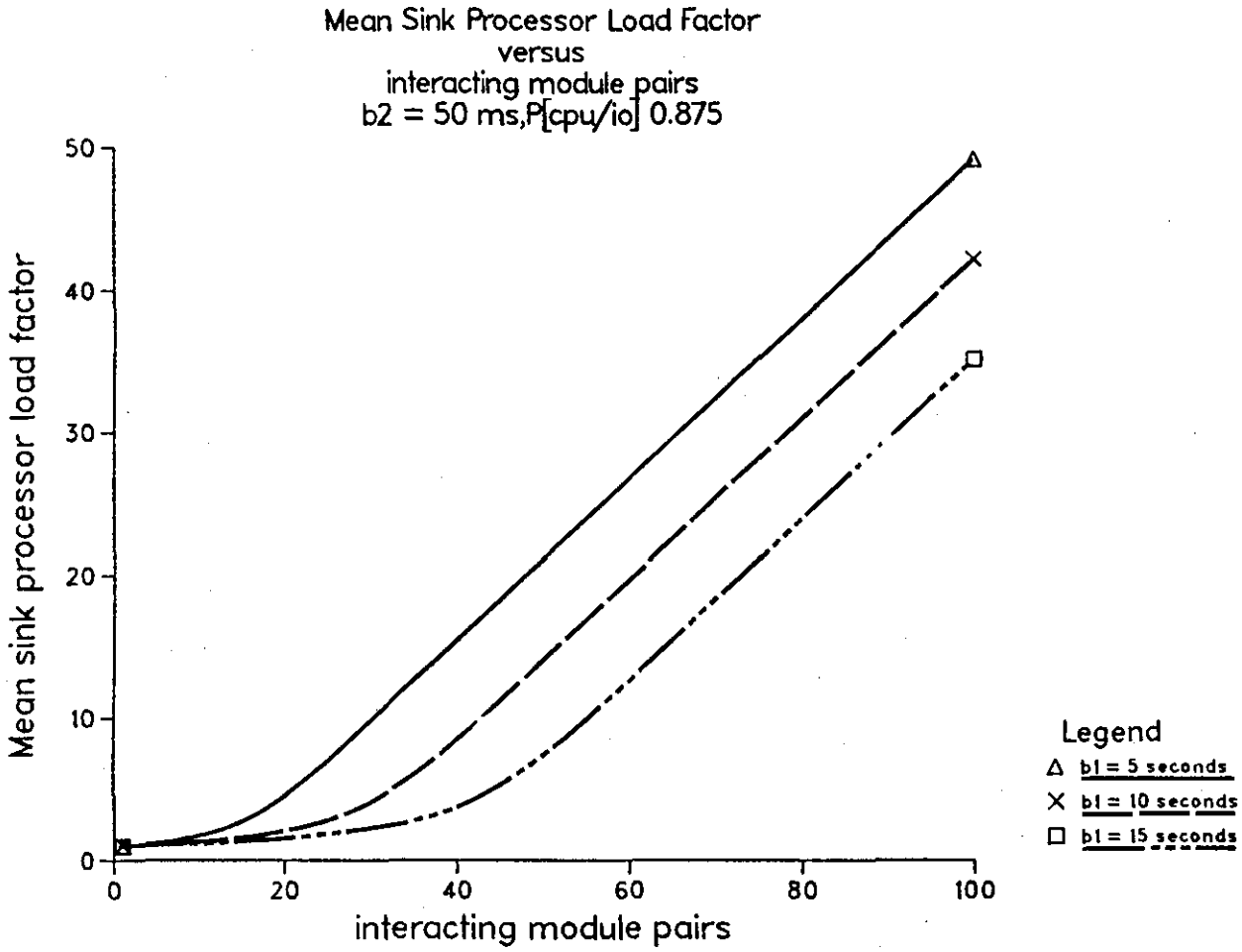


Fig. 6.9 Mean Sink Processor Load Factor

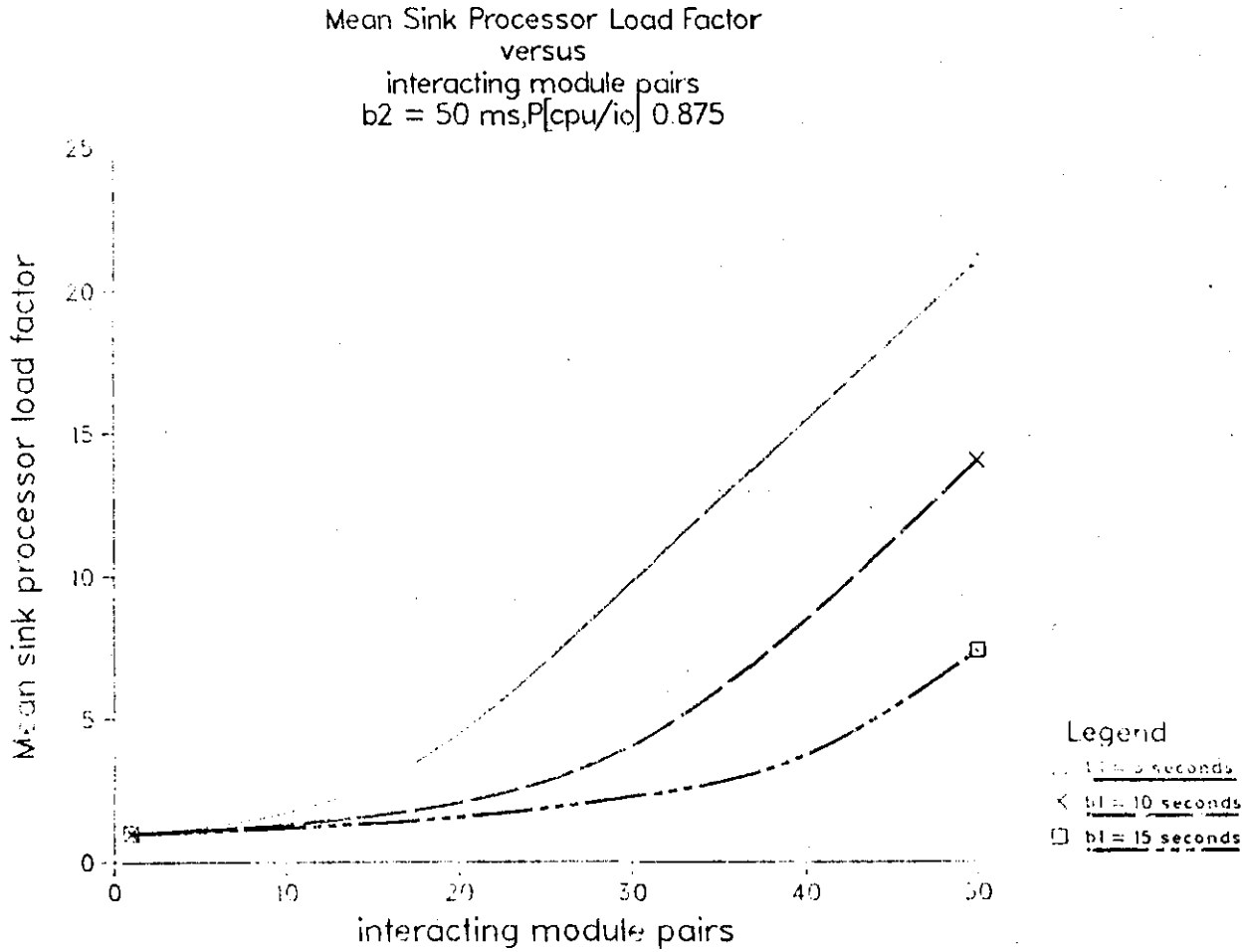


Fig. 6.10 Mean Sink Processor Load Factor

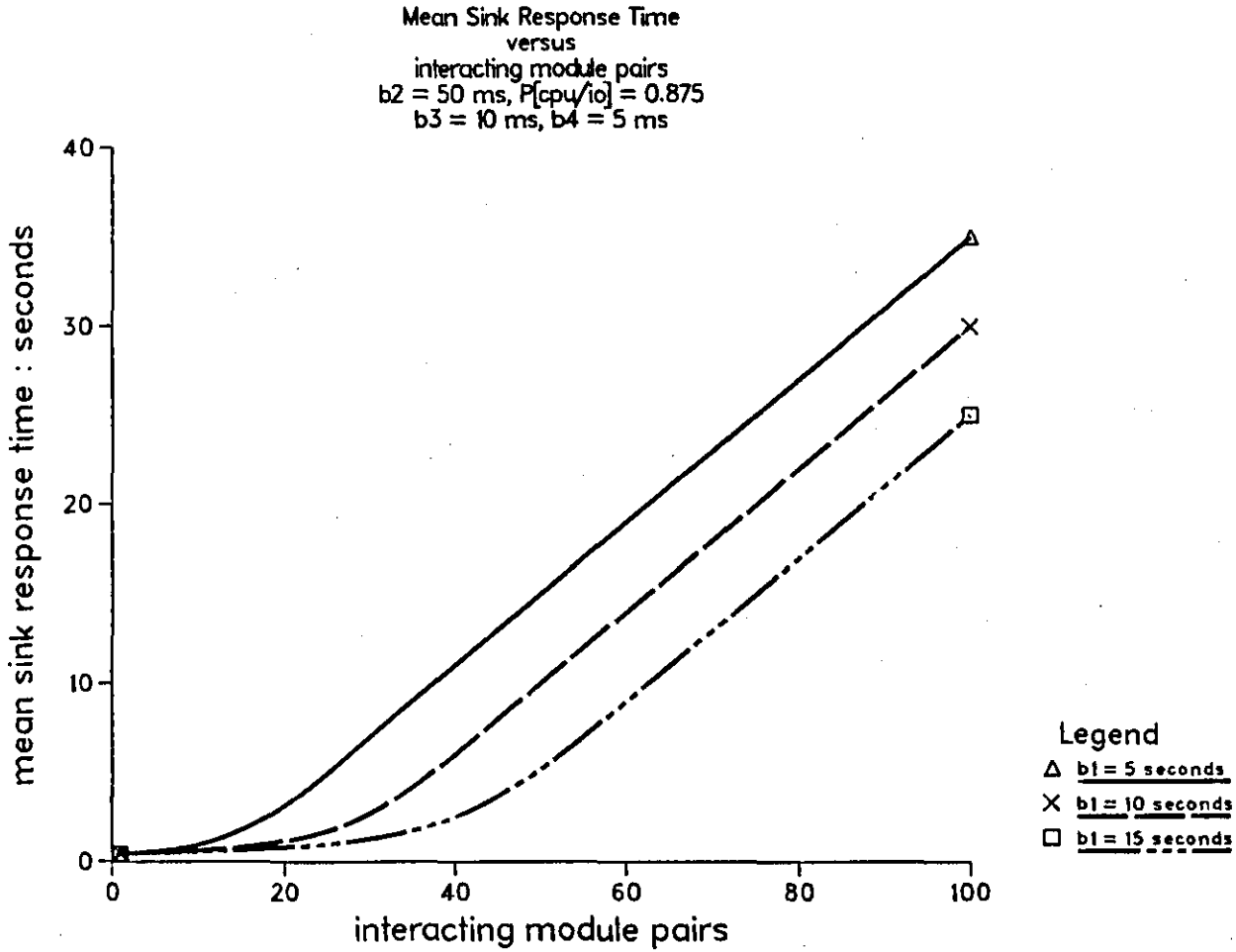


Fig. 6.11 Mean Sink Response Time

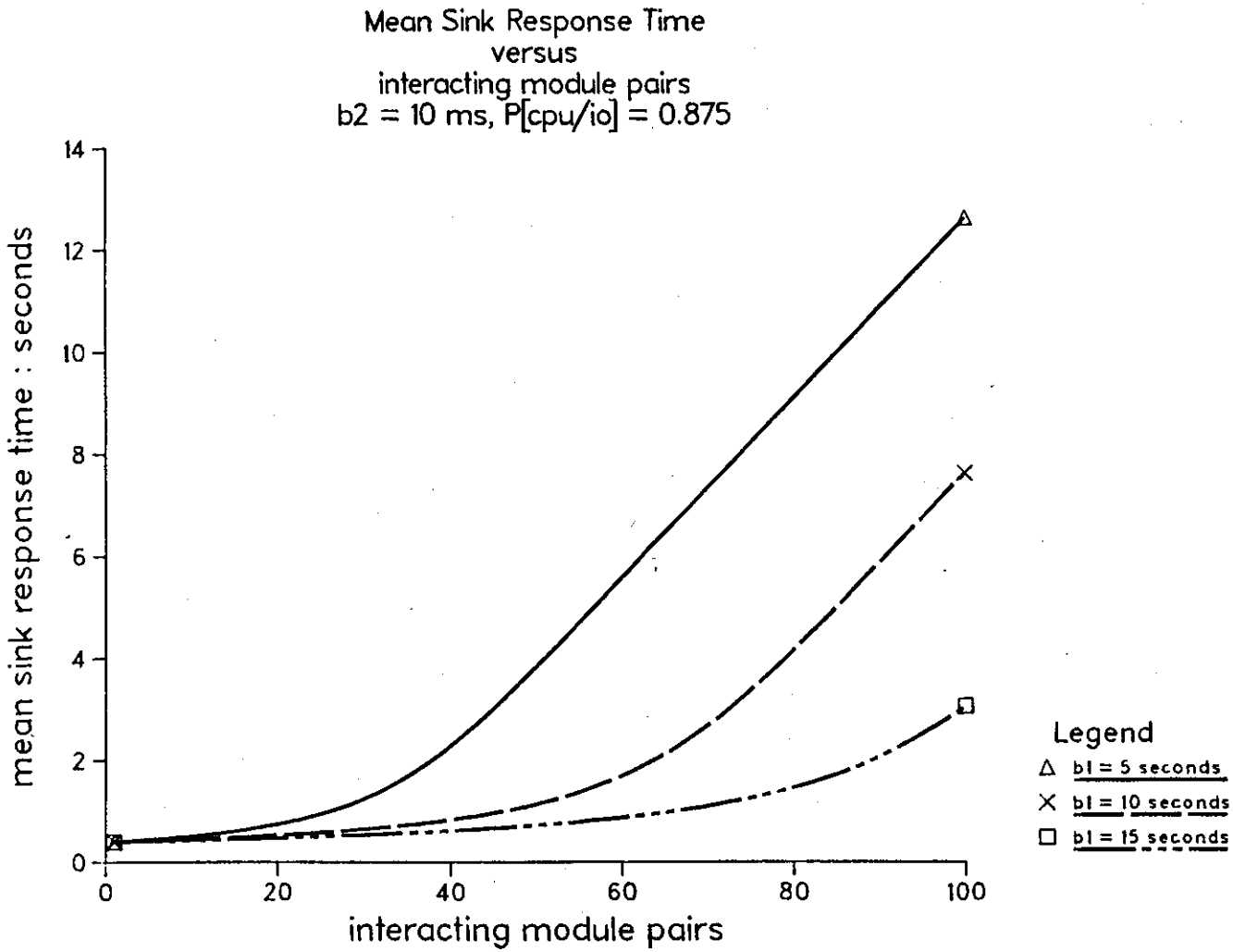


Fig. 6.12 Mean Sink Response Time

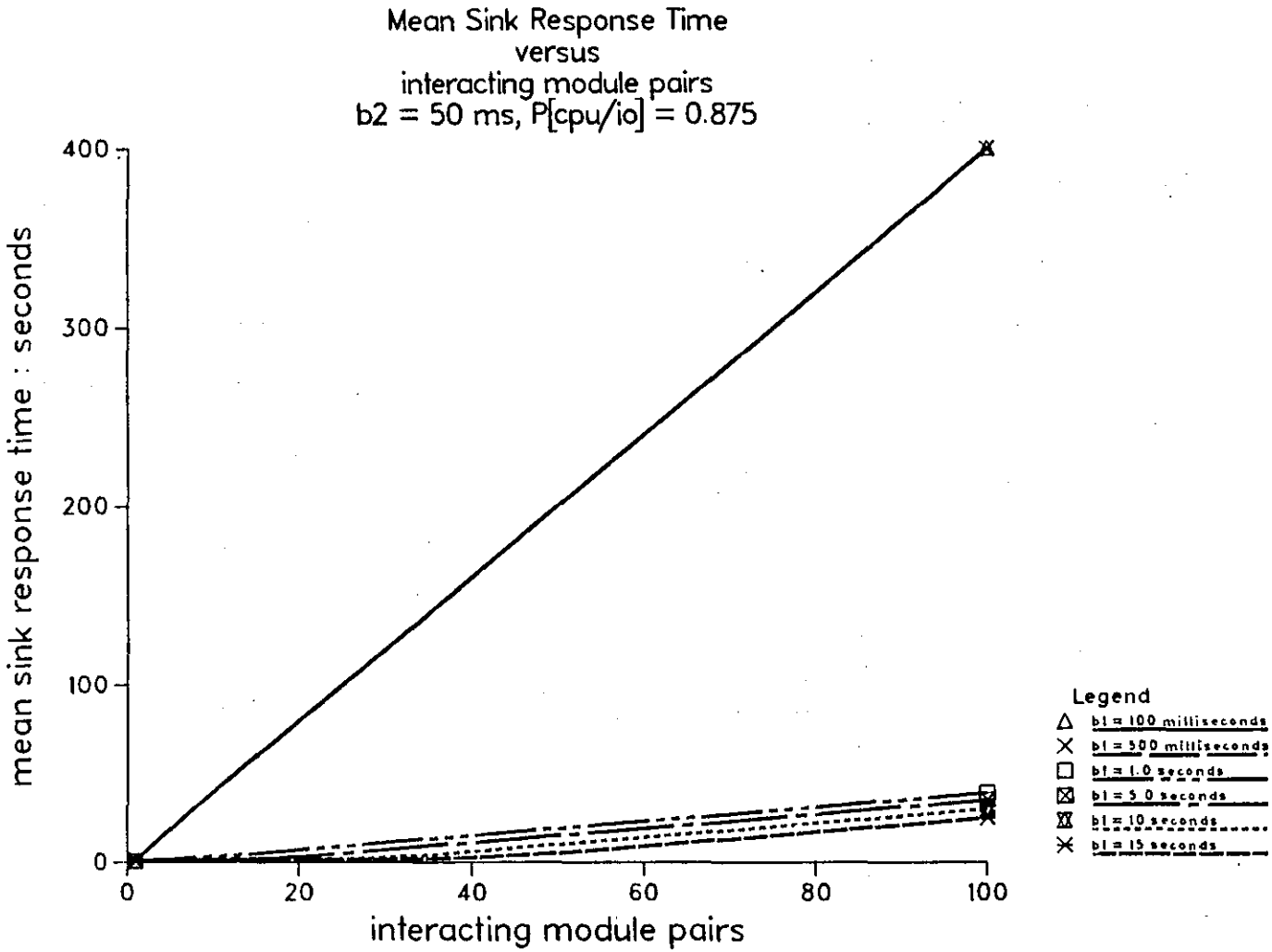


Fig. 6.13 Mean Sink Response Time

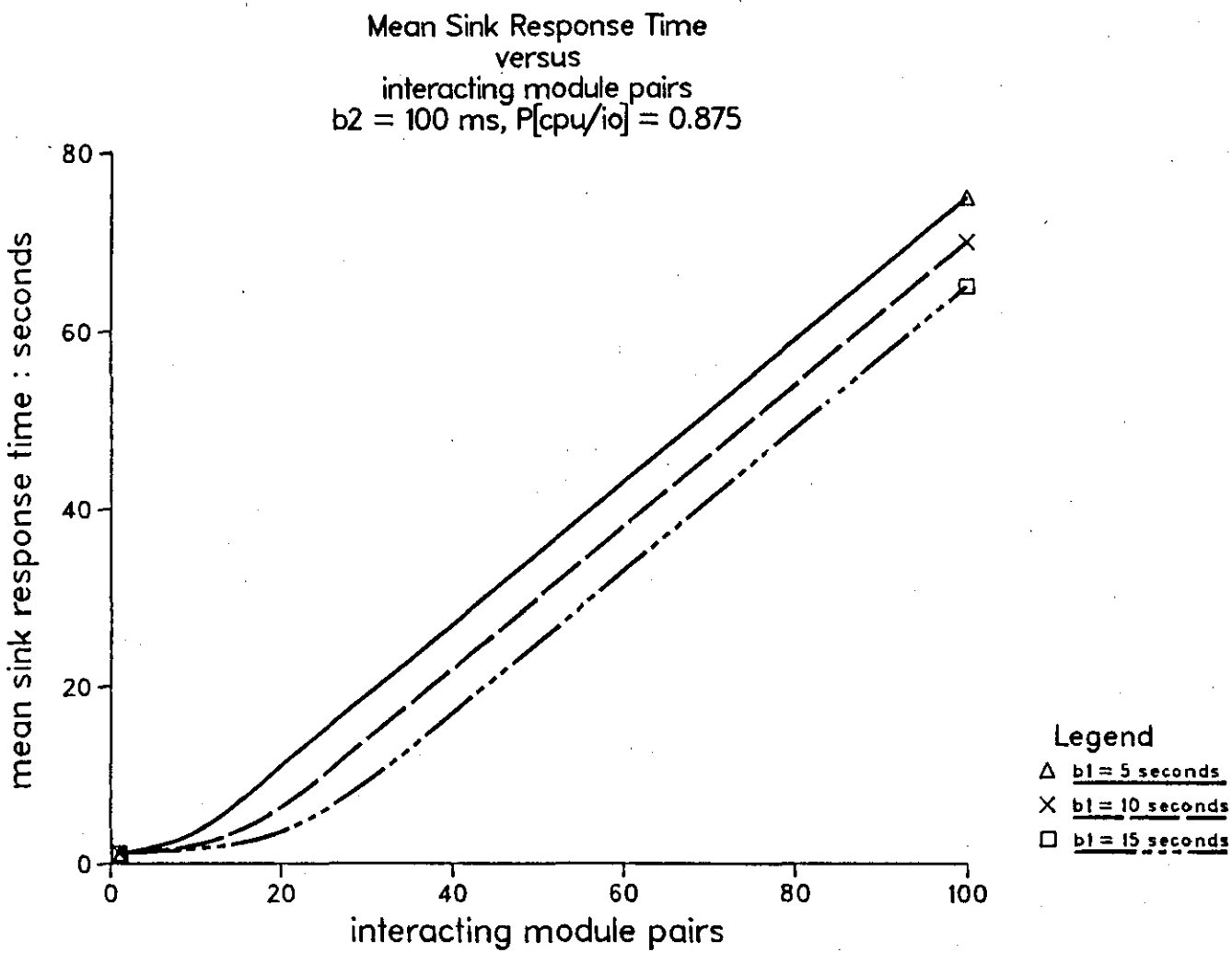


Fig. 6.14 Mean Sink Response Time

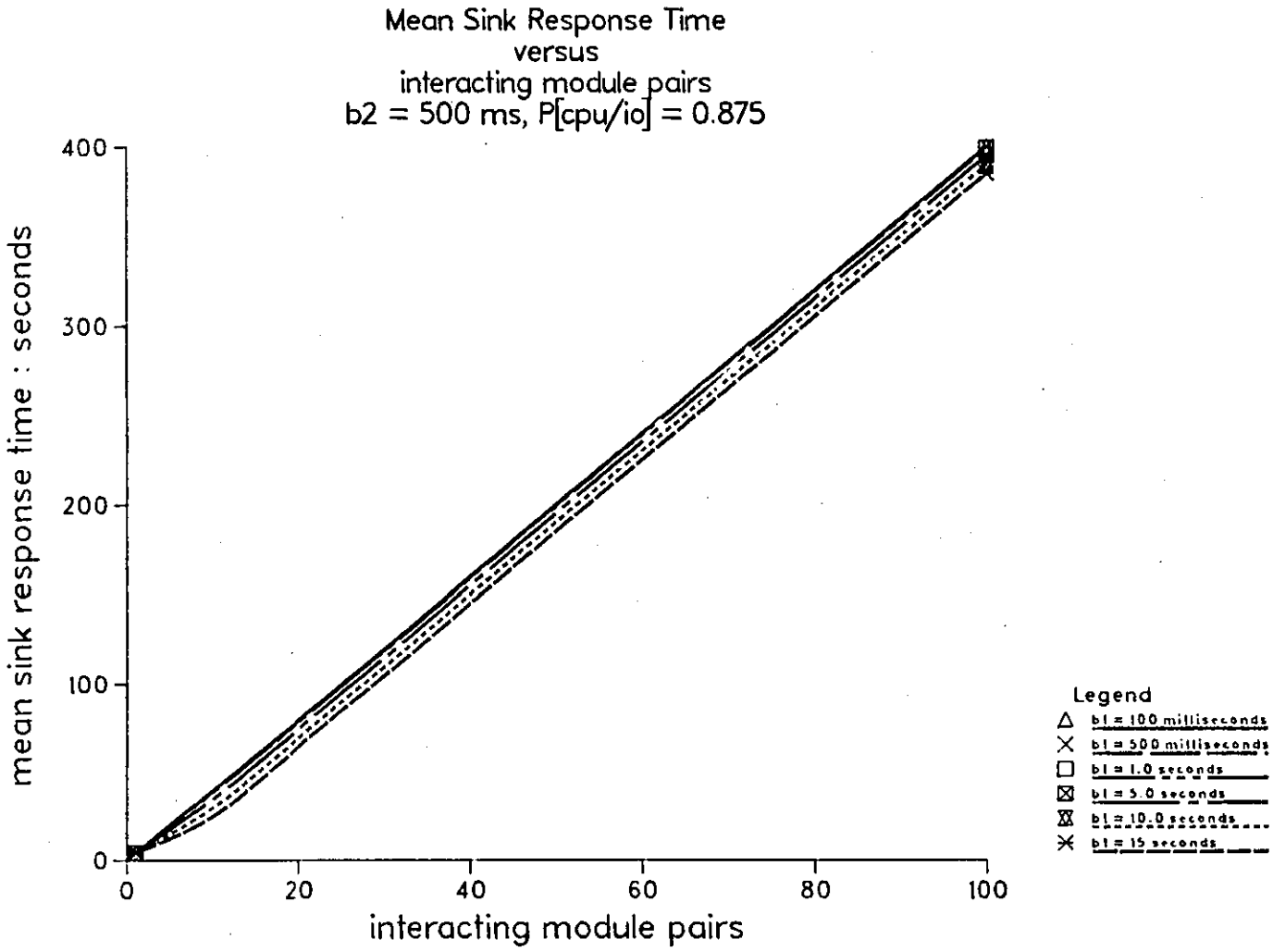


Fig. 6.15 Mean Sink Response Time

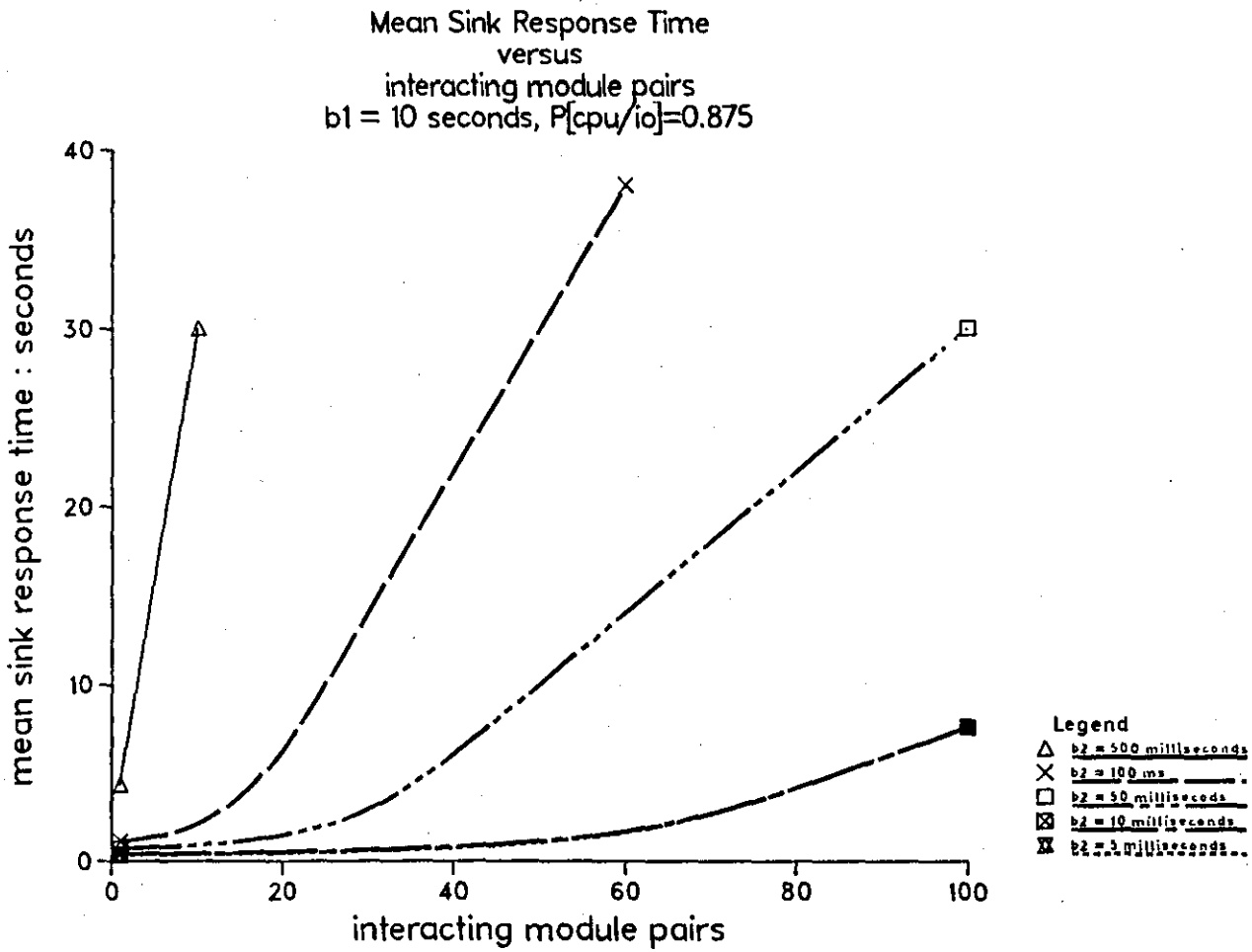


Fig. 6.16 Mean Sink Response Time

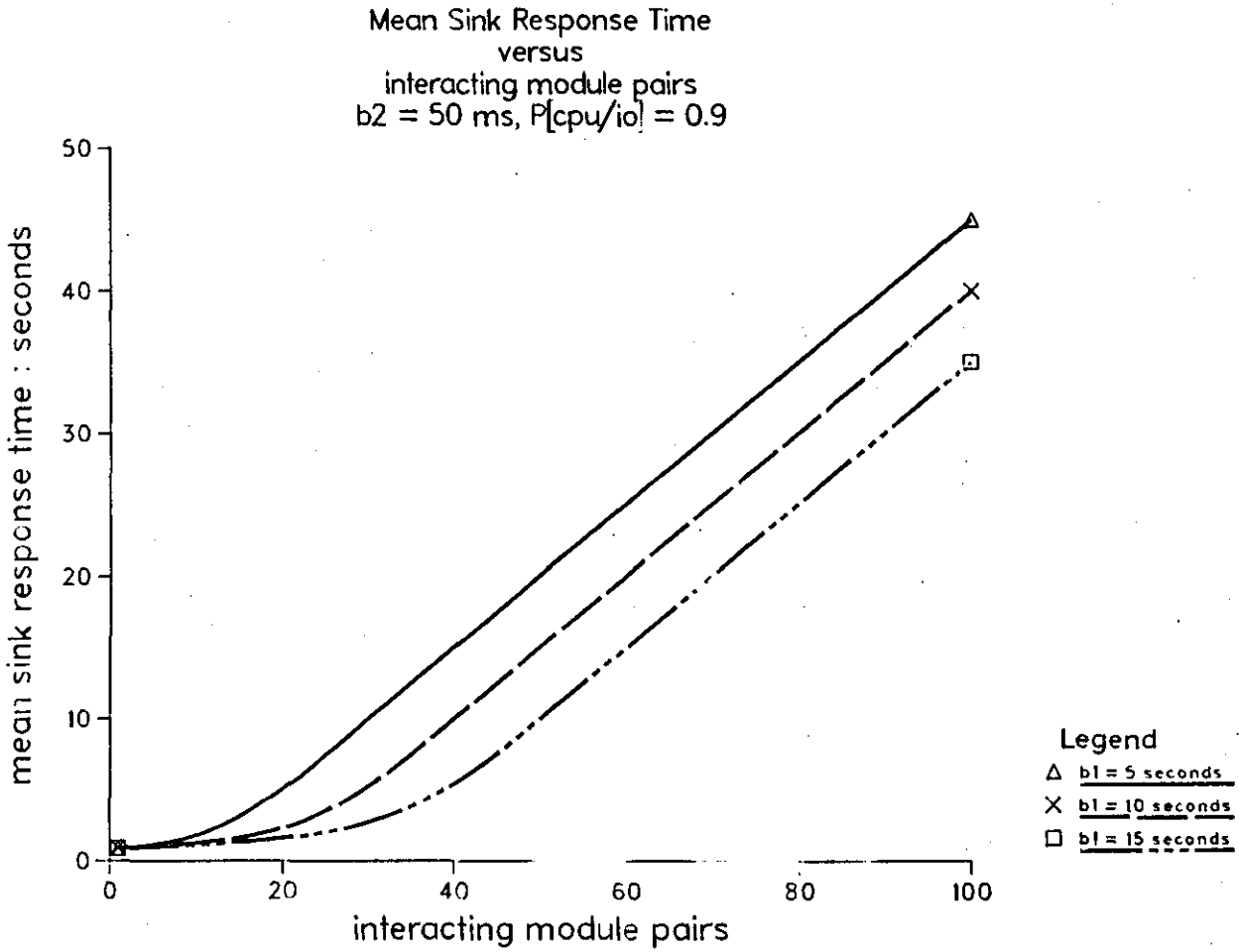


Fig. 6.17 Mean Sink Response Time

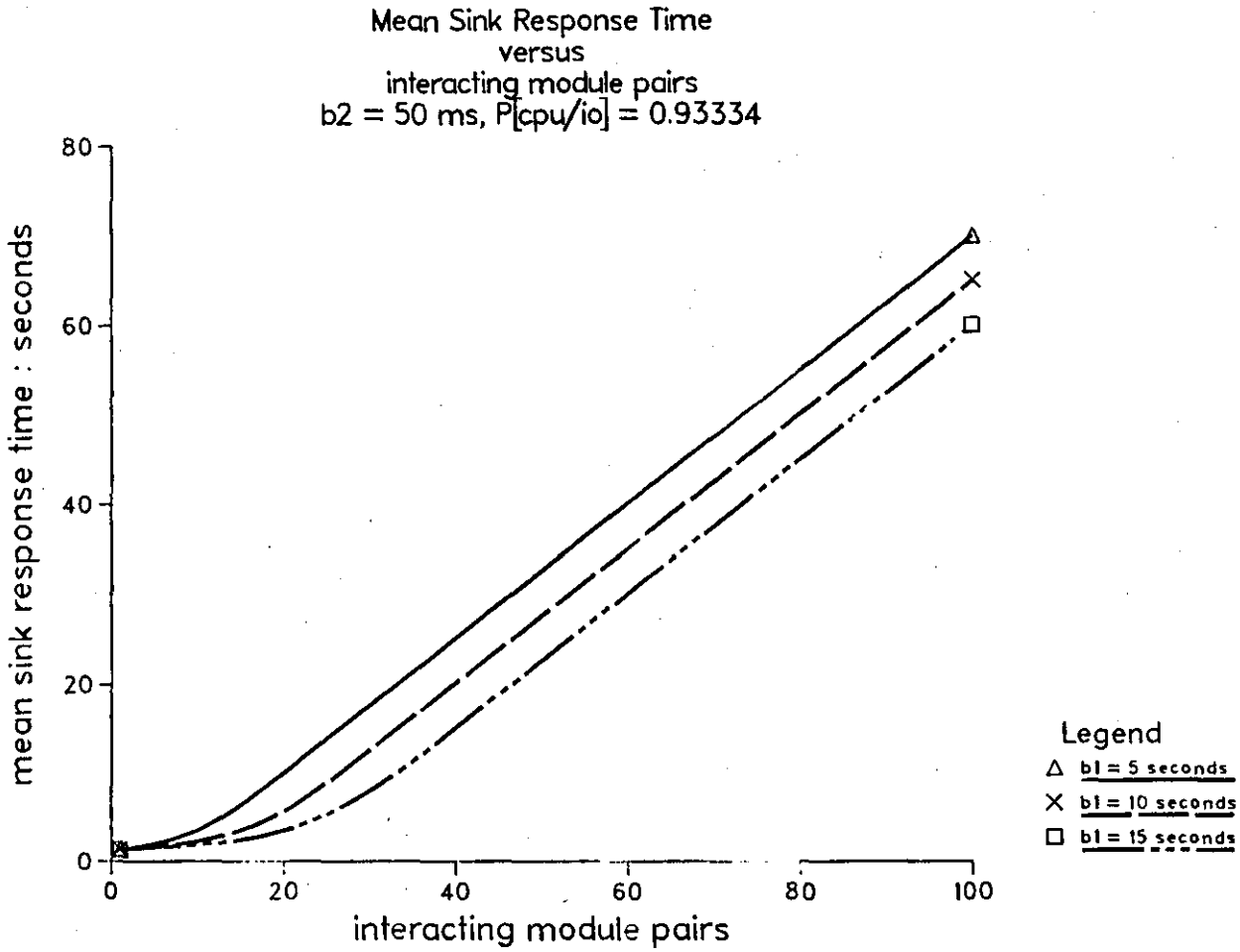


Fig. 6.18 Mean Sink Response Time

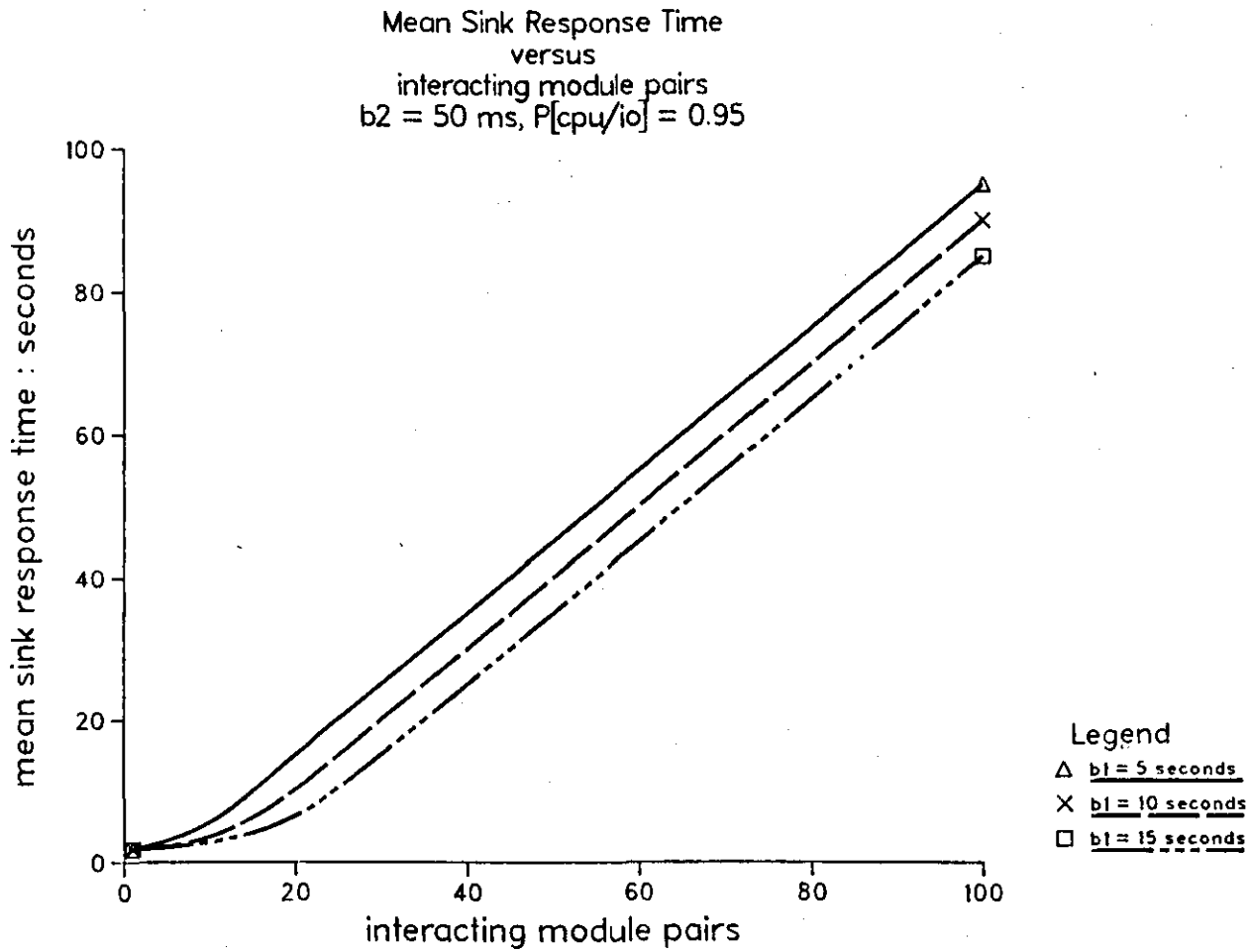


Fig. 6.19 Mean Sink Response Time

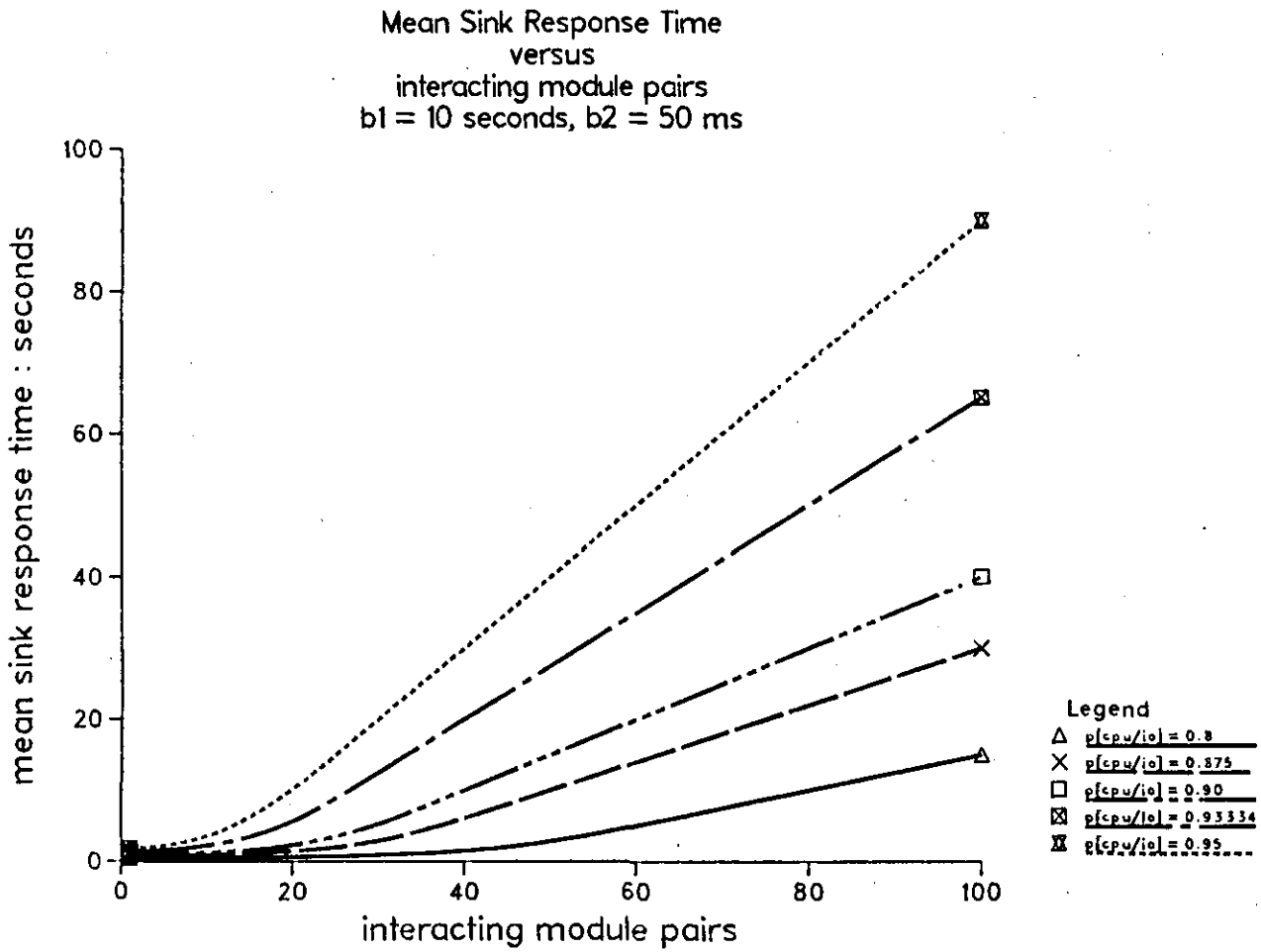


Fig. 6.20 Mean Sink Response Time

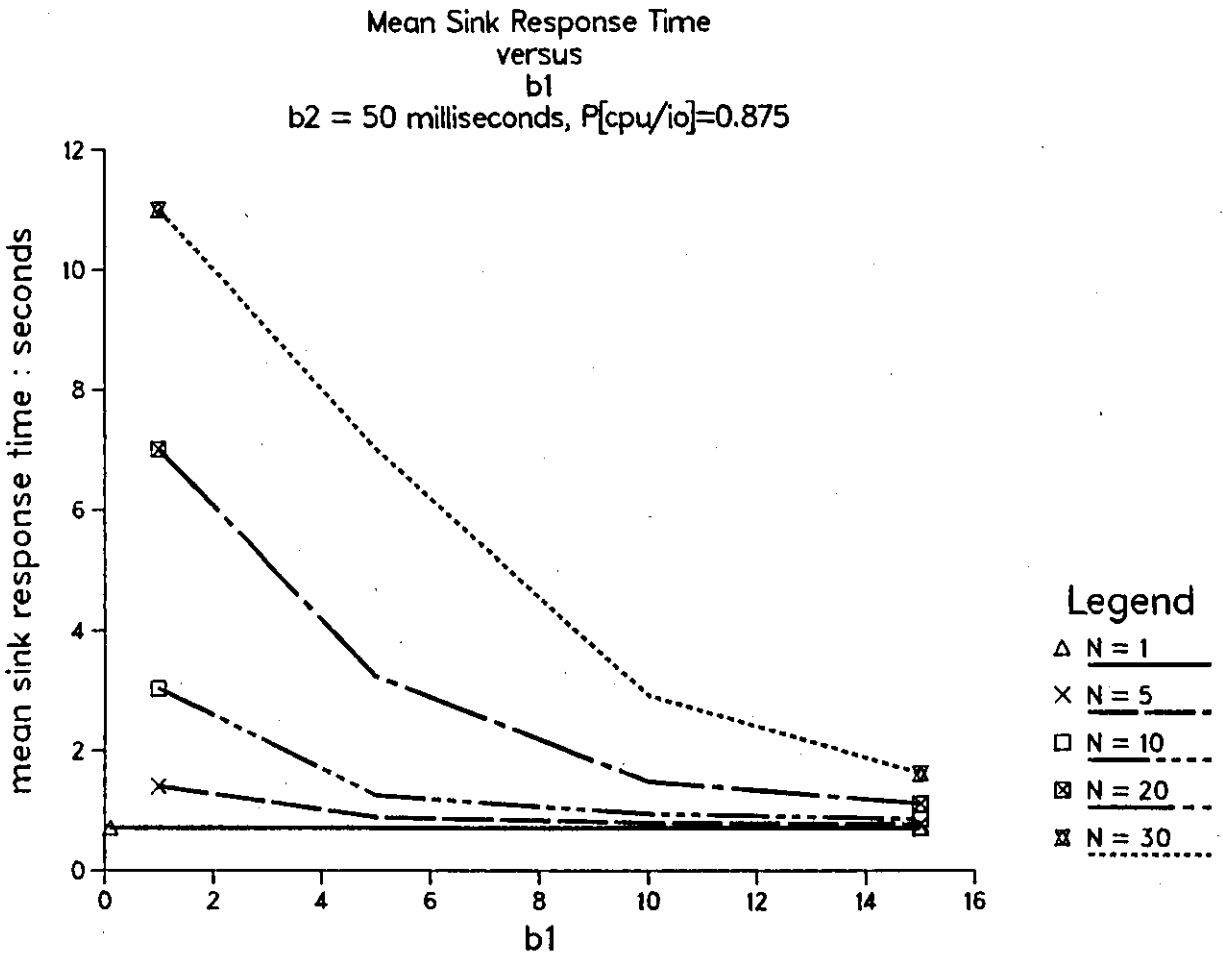


Fig. 6.21 Mean Sink Response Time

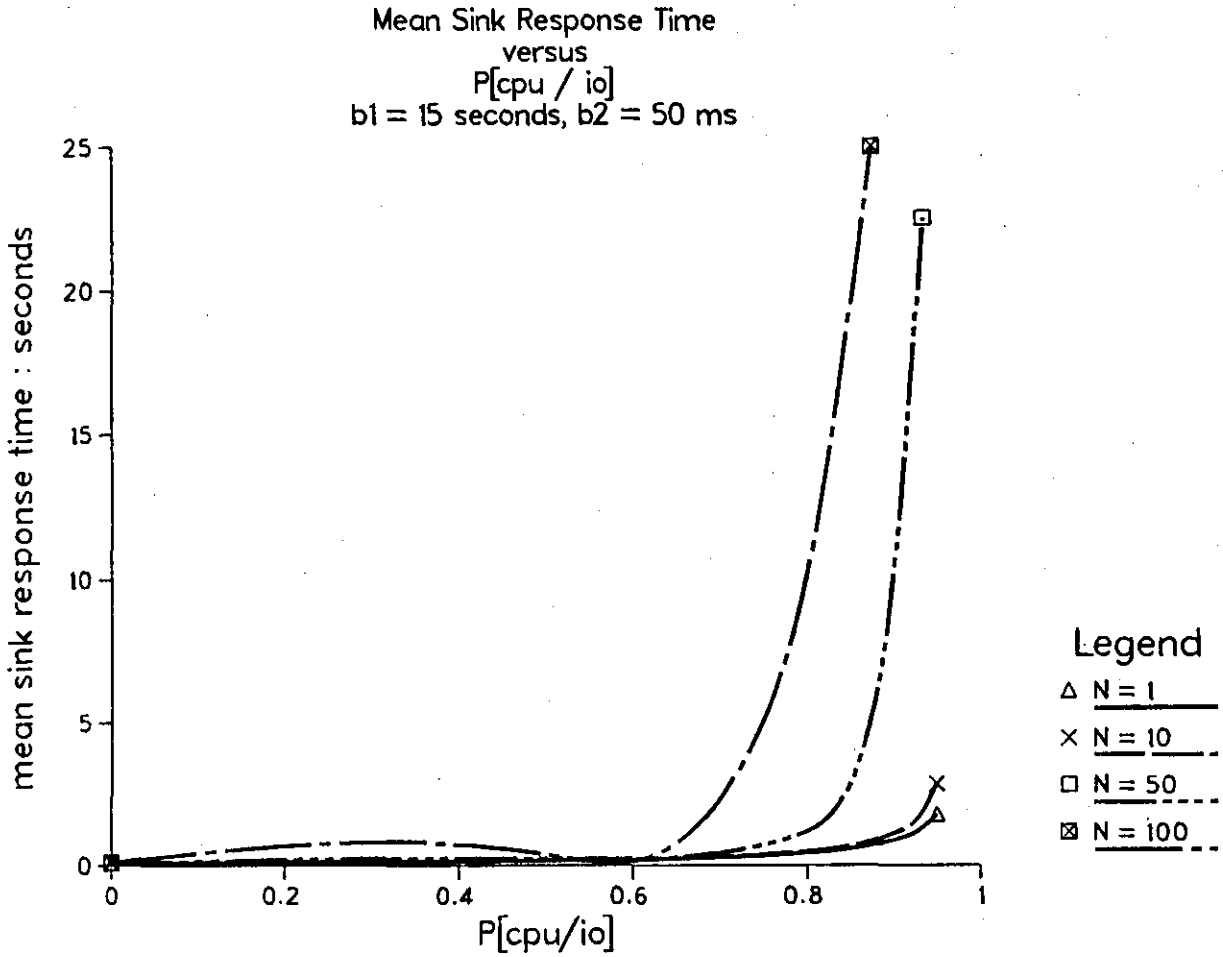


Fig. 6.22 Mean Sink Response Time

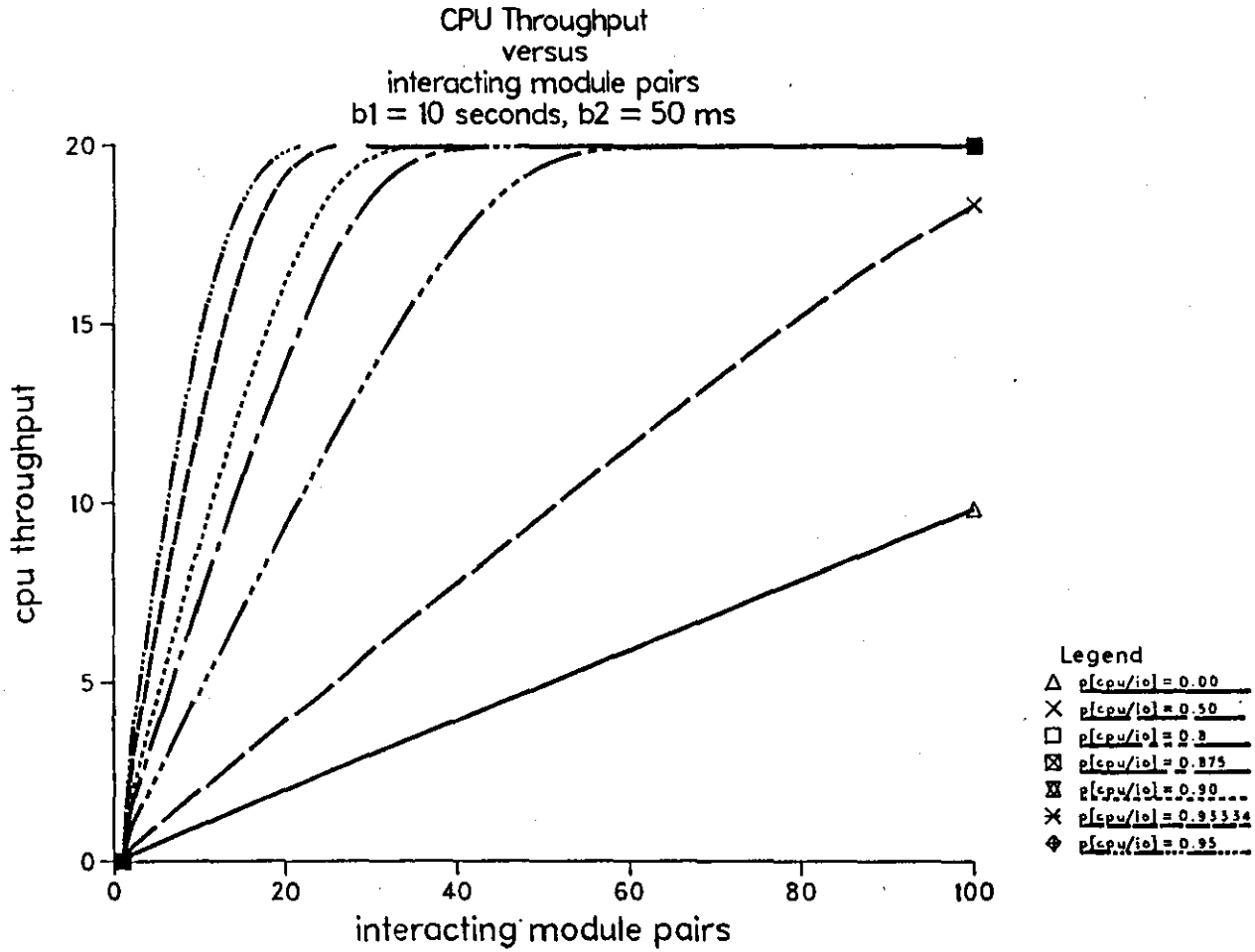


Fig. 6.23 CPU Throughput

processor system. Hence, it can be seen that the main factors that contribute to the computational workload and T(n) at the sink processor system are

a) n : the network population

b) b_m : the mean of the service time distributions

c) q : which determines the computational requirements

d) p : which determines the I/O requirements

The values of b_m are the mean values of the exponential service time distributions at the various system service facilities. Specifically, b_1 models the mean time between intermodule references and its value is very critical to the overall system performance.

Figs. 6.8 to 6.23 summarise the analytical performance result of the sink processor system as obtained by the use of the MVAA computational algorithm. They show how T(n) (or F(n)) varies with n for various values of b_m, p, and q. These values also show how the bottleneck and saturation effects, discussed earlier, govern the behaviour of T(n) and F(n). In this way, the variations of T(n) and F(n) are characterised by the horizontal and the linear asymptotes so that T(n) and F(n) rise very slowly with n, at first, but after the critical value of n, (n_s), is reached there is a sudden change of the linear asymptote slope. Hence, as long as n_s is not reached, T(n) remains quite low and is approximately equal to T(1), but the actual value is only marginally higher, as given by (30). The other system performance measures such as the service facility mean throughput, queue lengths, and queueing time and resource utilizations for each queue or resource may be obtained from the MVAA computational algorithm. Fig. 6.21 shows the variation of the sink processor system CPU throughput as a function of n for various values of b_m, p, and q.

This variation of the CPU throughput also shows the saturation and bottleneck effects when n exceeds certain critical values (n_s) and is also seen to be strongly dependent on the values of p, q, and b_m. From these results it may be seen that the behaviour of T(n) as predicted by equations (12) and (30) show great similarity, but the use of the computational algorithm may produce more information and with less effort than the use of the balance equation to the Markovian networks. Furthermore, the computation algorithm can form a simpler basis for simulation experimentation.

6.3 SIMULATION EXPERIMENTATION

Without measurement, it is difficult to have a true science. But, in the design and development of almost all systems, measurement is not possible. However, modelling becomes a necessary tool in such situations in order to estimate the system performance that may be expected from the complete system. Section 6.2 presented the analytic model of the dual processor CPU cache distributed computation system and some important system model performance measures such as the sink processor system response time and load factor were obtained. In such analytic models, many assumptions have to be made in order to obtain reasonable abstractions of the system performance to which probability theory can be used to obtain the equations that characterise system performance. The method of simulation experimentation can then be used to test the validity of these abstracted analytic models and to check whether the assumptions on which the models are based are valid or not. This section presents the simulation results of the model shown by fig. 6.5.

6.3.1 Simulation Model

Simulation experiments can be classified as either clock-driven or event-driven. Event-driven simulation models for queueing systems can be quite conveniently described, (ref. 6.35).

An event-driven simulation model for a queueing system can be considered as consisting of two basic phases, (ref. 6.35):

a) data generation

b) bookkeeping

Data generation involves the production of inter-arrival and service times where needed throughout the queueing system experiment and taking the queueing discipline at each queue into account. These queueing times are generated from the relevant probability distributions. The negative exponential distribution has been used as the service time distribution in the various queues. This is accomplished by the use of the various random number generators (ref. 6.36), and these are usually available at most university computer centres (e.g. the NAG routines). On the other hand, the bookkeeping phase of the simulation model deals with updating the system queues when new events (arrivals and departures) occur, monitoring and recording the system states as they change, and keeping track of the various quantities such as the beginning and end of busy times, idle times, queue lengths, and waiting times from which the various performance measures such as the throughput, utilization, and response time may be estimated. In this way, each event may be described by the time it is expected to occur and by the actions that must follow. For queueing network models, the simulation program maintains a list of events ordered by their time of occurrence. Hence, the program cycles through the following three basic steps;

a) Select the event with the earliest time

b) Set the simulated clock to this time

c) Perform the action

With FCFS queueing disciplines, the only events that need to be considered are the service completions. When the jobs are in the service facility, the simulation program does not need to take any action at all. However, when a job completes service the program must do all the book-keeping and reassign the server to a waiting job, if there is one, move the job to the next queue and possibly initiate service for the job there. With exponential distributions for service time, the probability of two or more simultaneous events is negligible. However, with non-exponential service time distributions, the probability of simultaneous events may occur frequently. (ref. 6.35).

6.3.2 Simulation Performance Estimates

As explained earlier, some of the most important basic performance estimates of a queueing model are the mean values of the resource utilization, resource throughput, queue length, and queueing time. From the mean queueing time, the system response time may be estimated. As in the analytic model of system behaviour, the simulation model assumes that the modelled system has attained equilibrium (ref. 6.37).

Resource utilization (U) may be defined as the fraction of time the server is busy. Hence, if the simulation experiment runs for time T, then U may be estimated by summing the individual busy times of the server and dividing this sum by T, so that

$$U = \frac{\text{sum of busy times}}{T}$$

For m identical servers

$$U = \frac{\text{sum of the busy times of the m servers}}{m.T}$$

The running sum of the busy times can be conveniently accumulated by recording the difference between when the server becomes busy and when the server becomes idle, and adding all such sub-busy periods to the running sum, fig. 6.24.

The resource throughput may be defined as the average number of jobs processed per unit of time. Hence, the throughput may be estimated simply by counting the number of jobs which get served at the particular resource and then dividing this by the length of the simulation run T, i.e.

$$R = \frac{\text{number served}}{T}$$

The mean queue length may be obtained in a similar way to the busy times: i.e. by finding the accumulated area of fig. 6.25 and dividing by the length of the simulation run T. The area may be estimated by first recording the time at which the queue length changes, subtracting the previously recorded time, and multiplying this time difference by the previous queue length, and finally adding that subarea to the running sum of the area, i.e.

$$L = \frac{\text{Accumulated area}}{T}$$

The mean queueing time may be obtained from the above values of queue length and throughput by the application of Little's result (ref. 6.29), since the mean queueing time is equal to the mean queue length divided by the throughput, i.e.

$$Q = \frac{\text{Accumulated area}}{\text{number served}}$$

The mean system response time may then be estimated by summing the mean queueing times at the various queues.

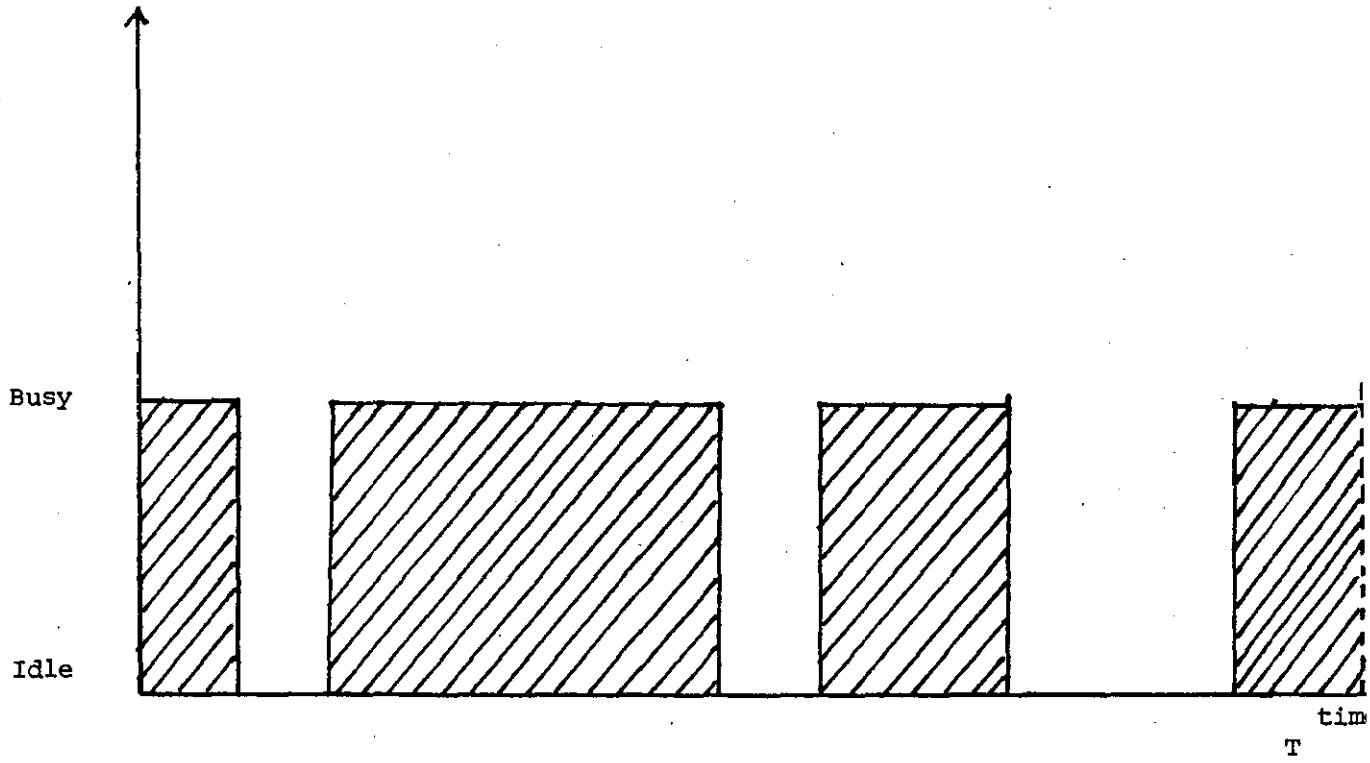


Fig. 6.24 Busy periods

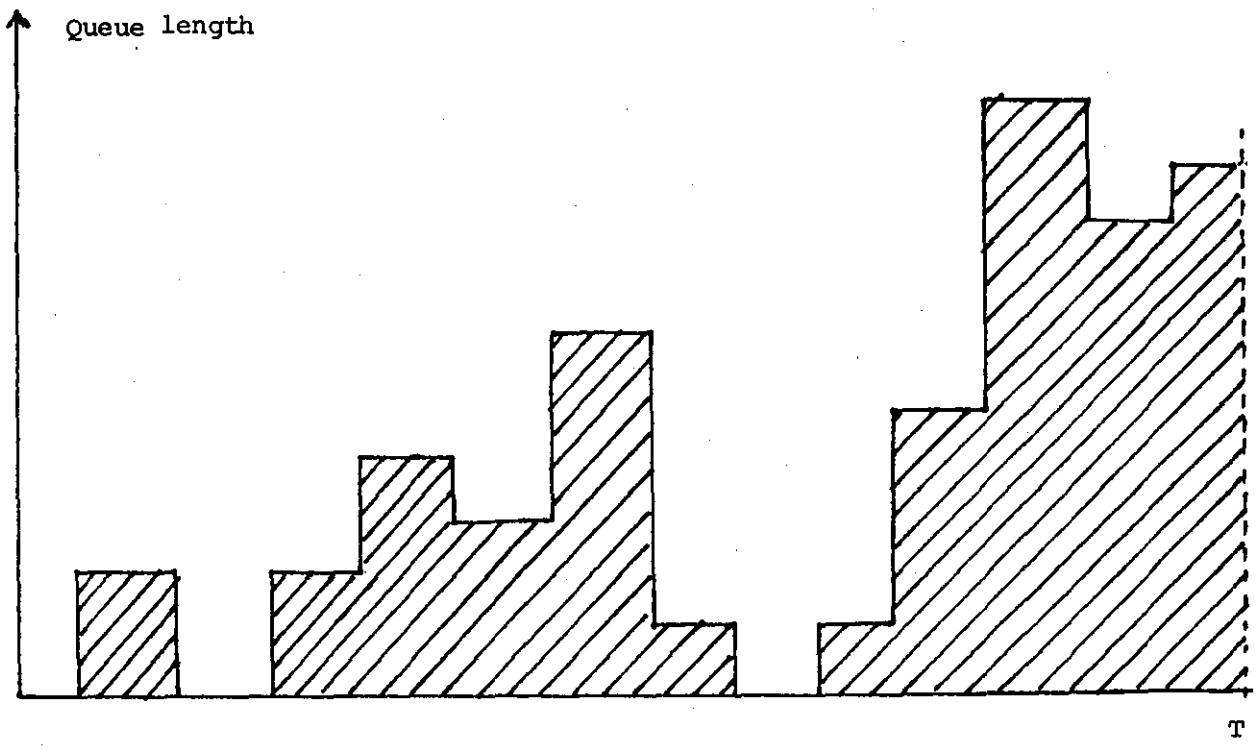


Fig. 6.25 Number in the system

In many cases, simulation experiments can be constructed with arbitrary amount of detail so that they model the system behaviour as closely as possible. In this way, they can be made as general as possible. Besides being one of its greatest advantages, this generality of the simulation experimentation provides it with a severe liability because the simulation models are liable to become unwieldy due to excess detail. If the running of a simulation is viewed as an experiment which entails statistical behaviour, then the methods of statistical analysis may be employed to deal with the statistical variability of the simulation results, (ref. 6.38). The two main methods for statistical analysis of simulation results are the methods of independent replications and the regenerative method (ref. 6.39, 6.40, 6.38, 6.41). In both these methods, the primary aim is to analyse the statistical behaviour of the results by estimating the confidence interval (ref. 6.42, 6.43, 6.44), which may be estimated by obtaining the estimates of the mean and variance of the performance measures. Some typical confidence levels used in such simulation analysis are the 90%, 95% or 99%. For example the 90% confidence level in the unit normal distribution defines the interval (-1.645, 1.645), (ref. 6.43, 6.44).

In the use of the method of independent replications in the statistical analysis of a simulation model, the aim is to repeat (replicate) the experiment many times and then use the average of these experimental values as the final estimate of the relevant performance measure. By making many such identical replications of the simulation runs, then it may be reasonably assumed that they obey the law of large numbers and that the central limit theorem (ref. 6.40), is applicable. If this is the case,

then it may be assumed that the average over the replications has a normal distribution, with a finite mean and a finite variance, so that the confidence level may be estimated (ref. 6.40, 6.43). In using these estimates for the mean and variance too, it is assumed that the simulation runs long enough to have attained equilibrium (ref. 6.37).

On the other hand, the method of regeneration in the statistical analysis of simulation models exploits the specific behaviour of a Markov process (ref. 6.39, 6.42). Since the future behaviour of a Markov process is dependent only upon the current state of the process, then each time the process enters that state the process will have the same expected future behaviour. In this way, a Markov process regenerates each time it enters a specified regeneration state and produces regeneration cycles between successive entrances to the state. A simulation model can take advantage of this regenerative phenomenon to estimate the confidence intervals for equilibrium behaviour if a regeneration state which is entered frequently enough can be identified. A more frequently entered state ensures short regeneration cycles. The main advantage of the regenerative method is that if the simulation is initialized in a regeneration state, then the simulation may be assumed to have been initialized in an equilibrium condition, so that observing the regeneration cycles is equivalent to observing periods of equilibrium behaviour (ref. 6.39, 6.42). Hence, besides recognizing the entrances to the regeneration states, the regeneration cycles which are of random length must be determined and used to estimate the confidence intervals, (ref. 6.43, 6.44). For many networks, and as long as no queue is saturated, the Markov state in which there are no jobs in the system is usually the most frequently occurring state and can be used as the regeneration state.

For a queue with exponential interarrival times, exponential service times, and having a FCFS single fired rate server, the queue length is geometrically distributed (ref. 6.30), so that

$$P[n] = (1-U).U^n \text{ , for } n = 0, 1, 2, \ldots.$$

where U is the utilization of the server, and since $U < 1$, then, $P[0] = 1-U$, is the most probable queue length. This result also holds for PS and the LCFSPR with single fixed rate server and arbitrary service time distributions (ref. 6.30).

Hence, if the number of replications, or the number of the regeneration cycles is large, and if the value of the performance measure for each of these simulation runs can be taken to be independent and identically distributed random variables, then the law of large numbers and the central limit theorem may be used to obtain the confidence interval (C.I.) The confidence interval is obtained by the use of the standard (unit) normal distribution given by (ref. 6.40)

$$F_z(z_o) = \int_{-\infty}^{z_o} \frac{1}{\sqrt{2\pi}} \; e^{-z^2/2} \; .dt$$

with density

$$P_z(z_o) = \frac{1}{\sqrt{2\pi}} \; . \quad e^{-z_o^2/2}$$

If $F_z^{-1}(a)$ = inverse of $F_z(z_o)$

$$= P[z_o \leq F_z^{-1}(a)] = a$$

so that

$$P[0 \leq z_o \leq F_z^{-1}(a)] = a - 0.5$$

and

$$P[-F_z^{-1}(a) \leq z_o \leq F^{-1}(a)] = 2a - 1$$

or, for $0 \leq C \leq 1$, then

$$P\left[-F_z^{-1}\left((1+C)/2\right) \leq z_o \leq F_z^{-1}\left((1+C)/2\right)\right] = C$$

If $C = 0.9$, then

$$F_z^{-1}(0.95) = 1.645$$

so that

$$P\left[-1.645 \leq z_o \leq 1.645\right] = 0.9$$

Hence for n independent and identically distributed random variables, each with mean m and variance σ^2, then, if the sample mean is y_n and the sample variance is $\frac{\sigma^2}{n}$, then for n large enough y_n can be assumed to have a normal distribution and $(y_n - m).\frac{\sqrt{n}}{\sigma}$ has the standard normal distribution, so that

$$P\left[-F_z^{-1}\left((1+C)/2\right) \leq (y_n - m).\frac{\sqrt{n}}{\sigma} \leq F_z^{-1}\left((1+C)/2\right)\right] = C$$

or

$$P\left[y_n - d \leq m \leq y_n + d\right] = C$$

where

$$d = F_z^{-1}\left[(1+C)/2\right]\frac{\sigma}{\sqrt{n}}$$

and

$\left[y_n - d, y_n + d\right]$ is a random interval called the confidence interval whose 90% confidence level may be defined by $C = 0.9$.

The sample variance, s^2, of n independent and identically distributed random variables x_i is defined by, (ref. 6.40)

$$s^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (x_i - y_n)^2$$

$$= \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n.y_n^2\right)$$

which, for large n, may be used to estimate the variance of the performance measures.

In the calculation of the queueing time performance measure a quotient of two averages x_n and w_n, (queue length and number served), is used, where x_n is the average of the set of the random variables V_i, and w_n is the average of the set of random variables U_i, during the i^{th} regeneration cycle. In this case the joint sample variance may be obtained from (ref. 6.41, 6.42, 6.43).

$$S^2 = S_u^2 - 2.y_n.S_{uv} + y_n^2.S_v^2$$

where

$$S_u^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} U_i^2 - n.w_n^2 \right)$$

$$S_{uv} = \frac{1}{n-1}\left(\sum_{i=1}^{n} U_i.V_i - n.w_n.x_n \right)$$

and

$$S_v^2 = \frac{1}{n-1}\left(\sum_{i-1}^{n} V_i^2 - n.x_n^2 \right)$$

from which the confidence interval estimate $[y_n - d, y_n + d]$ may be obtained, where

$$d = \frac{F_z^{-1}\,((1+C)/2).s}{x_n.\sqrt{n}}$$

Hence, the simulation program must recognise that the i^{th} regeneration cycle has ended and maintain values of U_i, V_i, $U_i.V_i$, U_i^2, for i = 1, 2, ... n.

6.3.3 Simulation Language

Many programming languages are available for simulation. Some of the main examples of such languages are Basic, Fortran, Pascal, PL/1, Coral 66, and APL. Among these languages, Fortran is the oldest and was the first high-level language to be introduced, and has continued to enjoy

wider acceptance over the years. It is simple to use, and its wide acceptance and dominance over the years has resulted in a great wealth of software and experience. In particular the GINO and NAG Library routines, which are available to computer centres of many universities, is a rich source of a variety of many efficiently coded, sophisticated and invaluable subroutines. Unfortunately, Fortran has many shortcomings such as cumbersome character handling, limited flexibility and in some cases, inefficient use of core store. More seriously, Fortran suffers from the so-called "spaghetti-code" problem in which the statements in a program are convoluted, often with no discernible beginning or end. One reason for this is the use of the control "goto" and the statement label numbers which tend to spread out an algorithm. Some of these problems, however, have been removed by a recent version of structured Fortran.

Pascal has been used for the simulation of the queueing models of fig. 6.5 (ref. 6.45, 6.46, 6.48). Besides being a simple language to use it has also quickly gained wide acceptance. One of the main advantages of Pascal is that it is a block-structured language and it is easy to implement the methods of top-down design with it as explained in an earlier chapter. As Pascal is implemented on both the Prime and Multics computers of the computer centre of the university, it is possible to use many of the existing NAG and GINO library routines such as the random number generators (ref. 6.47), and other routines used in the simulation of the queueing model.

The results of the simulation are shown in figs. 6.26 to 6.35. These figures show the results of the various performance measures of the queueing model of the dual processor system. The comparison of the

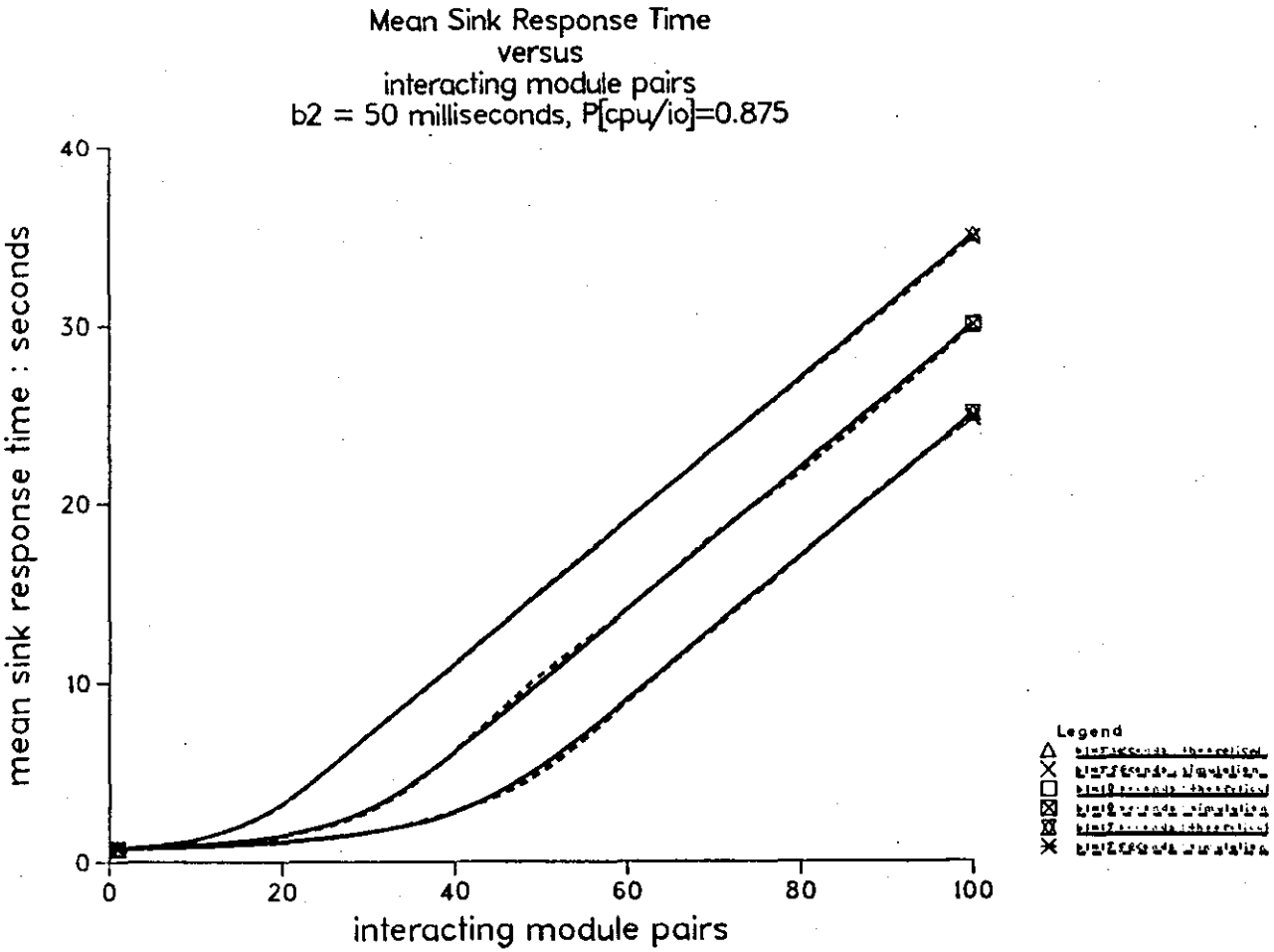


Fig. 6.26 Mean Sink Response Time

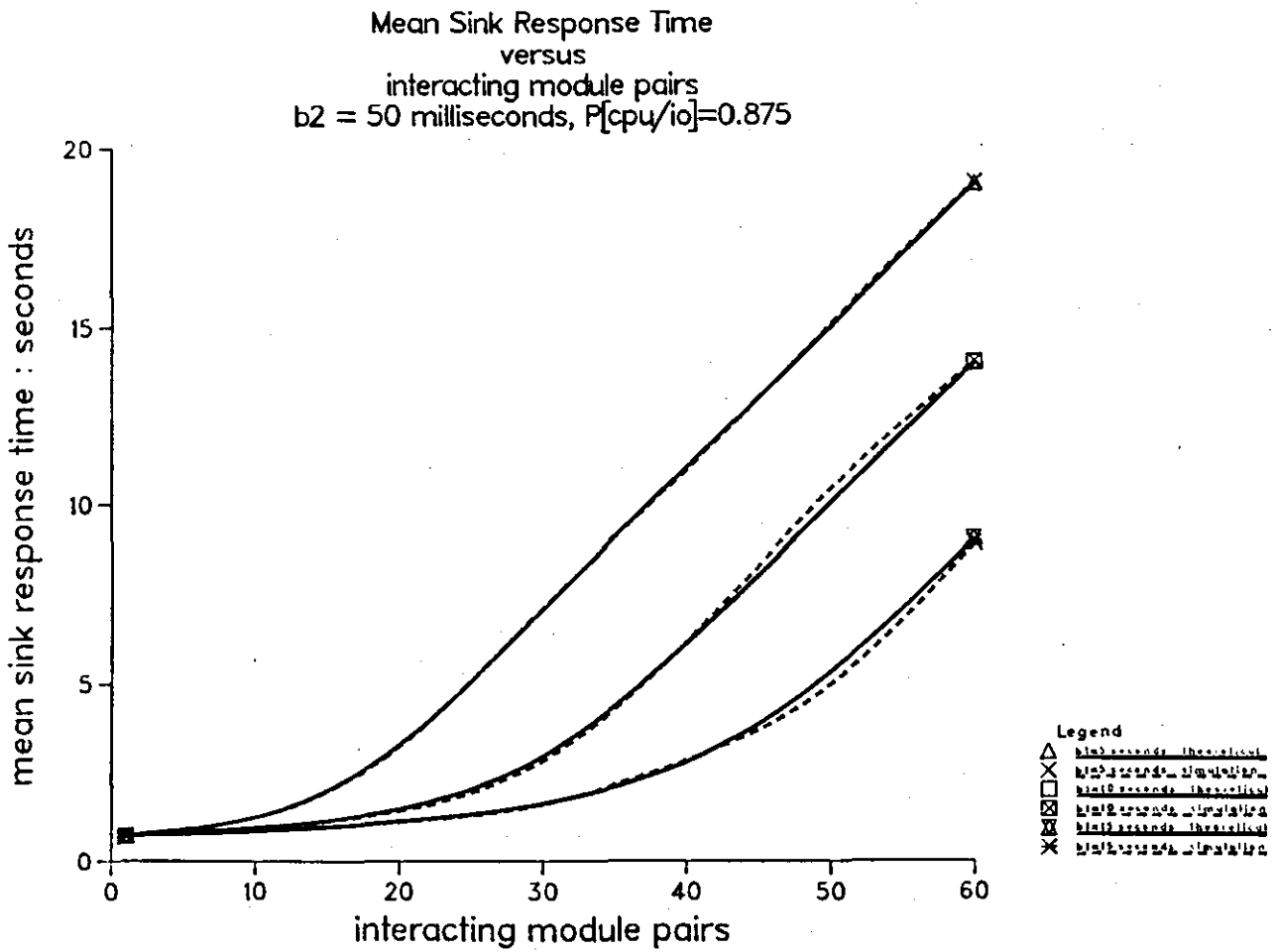


Fig. 6.27 Mean Sink Response Time

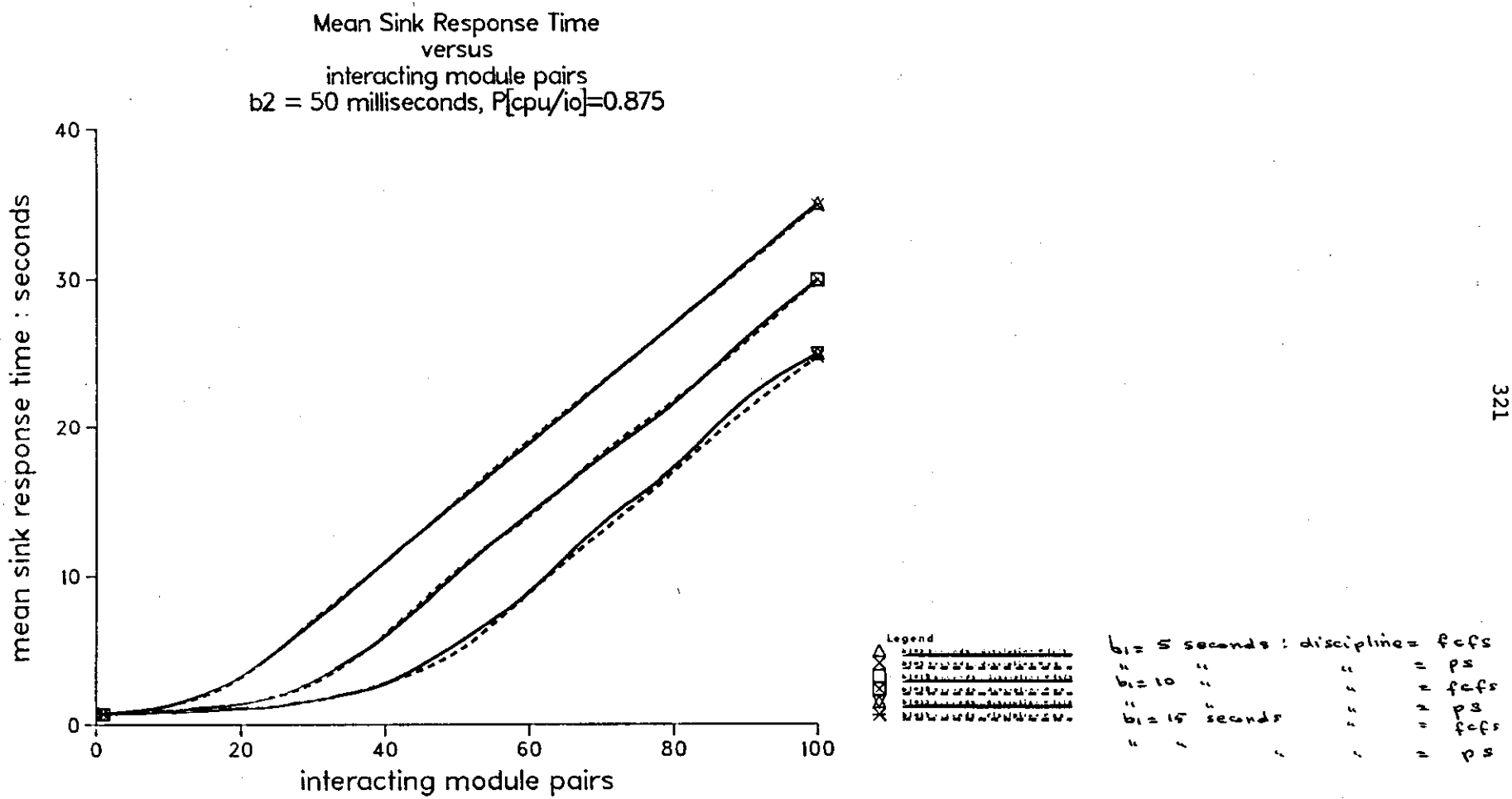


Fig. 6.28 Mean Sink Response Time

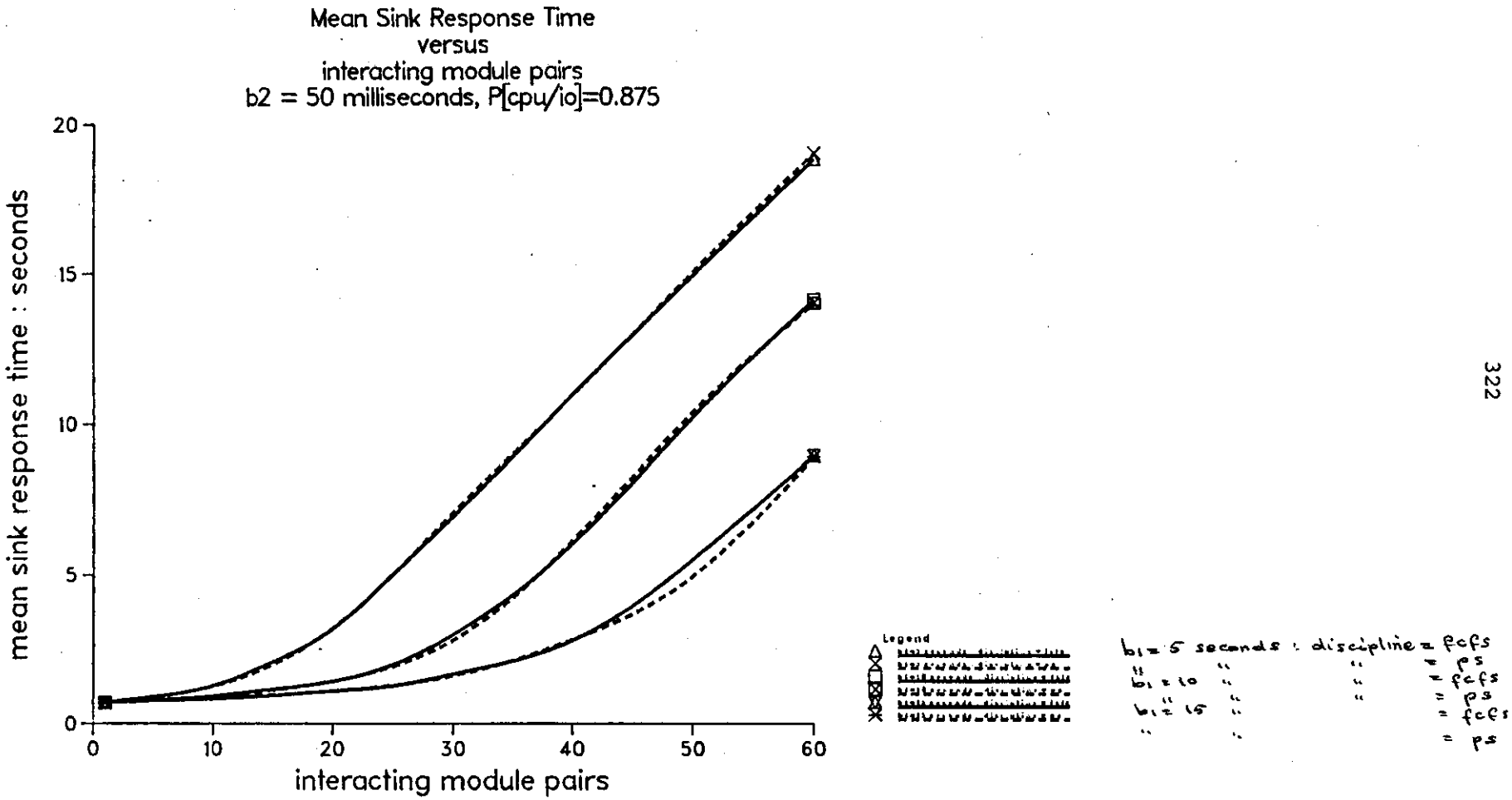


Fig. 6.29 Mean Sink Response Time

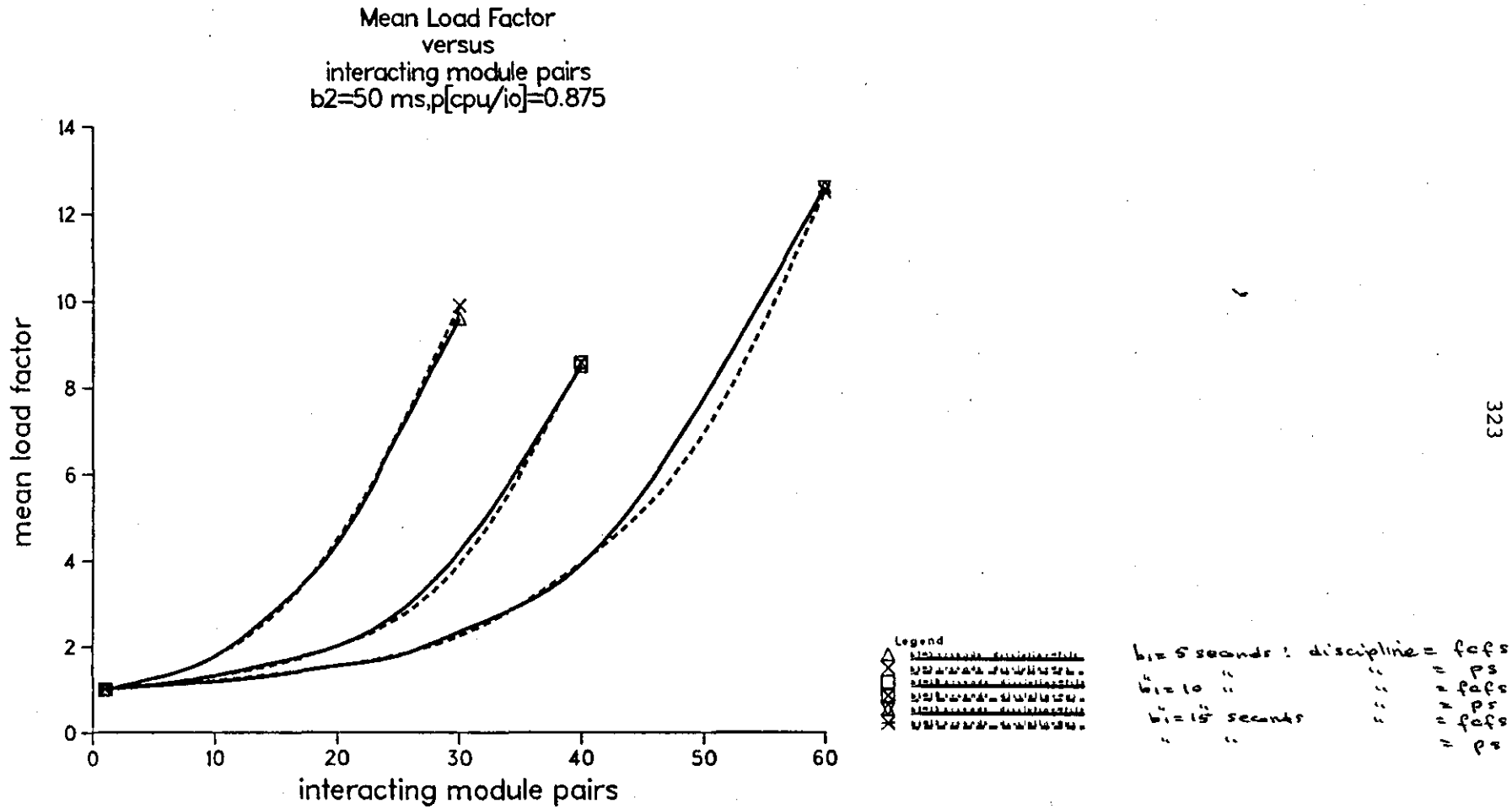


Fig. 6.30 Mean Load Factor

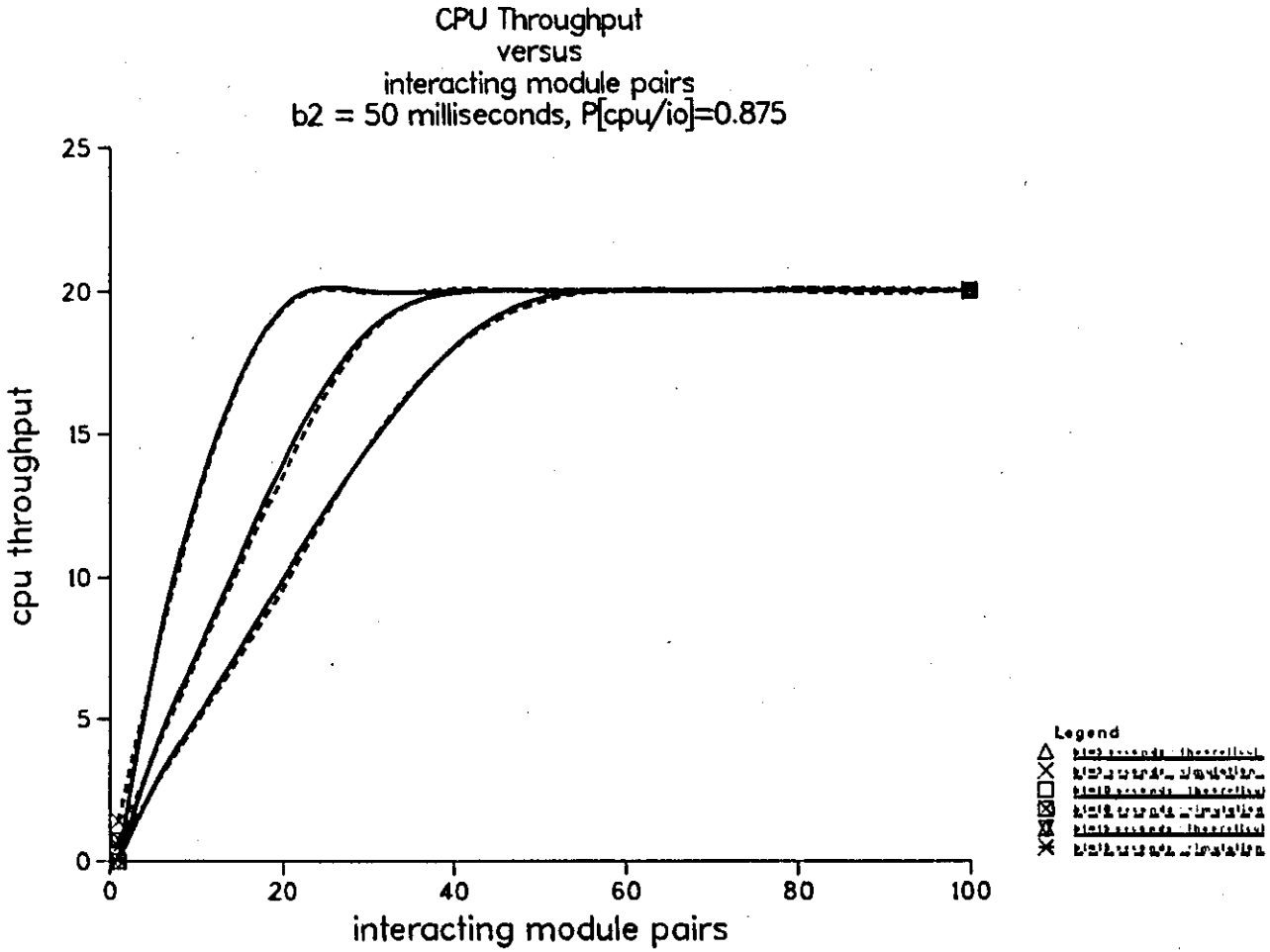


Fig. 6.31 CPU Throughput

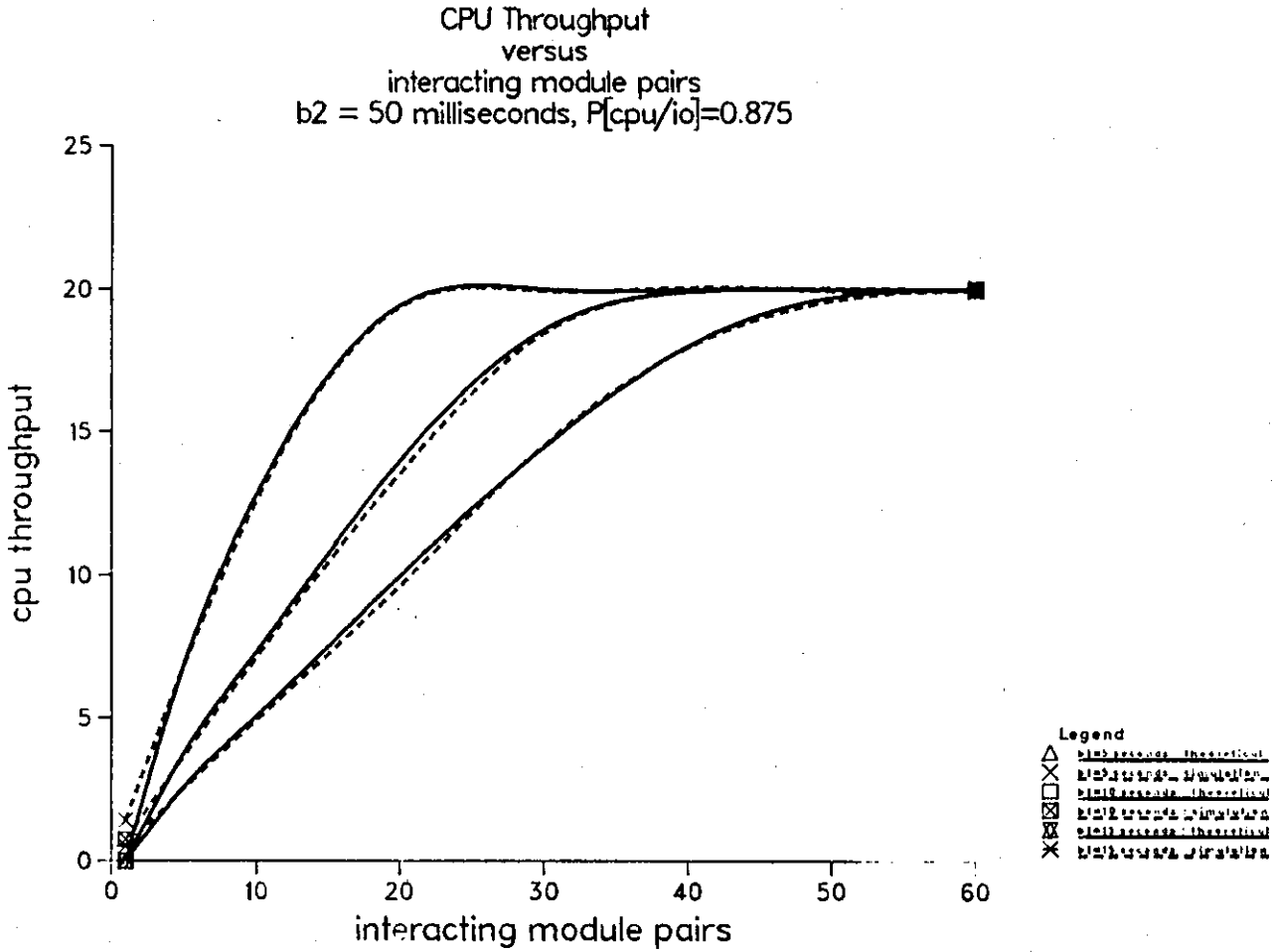


Fig. 6.32 CPU Throughput

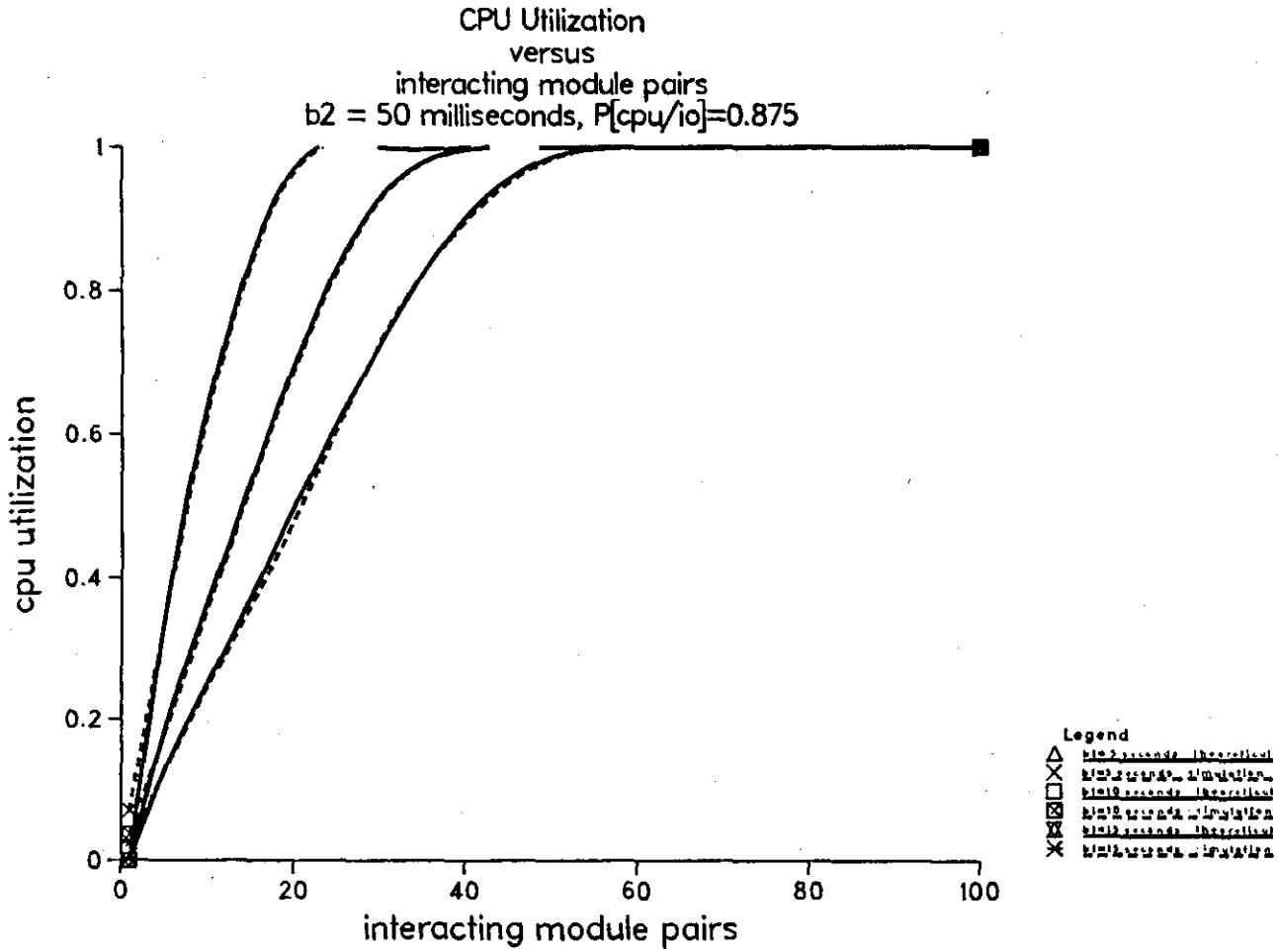


Fig. 6.33 CPU Utilization

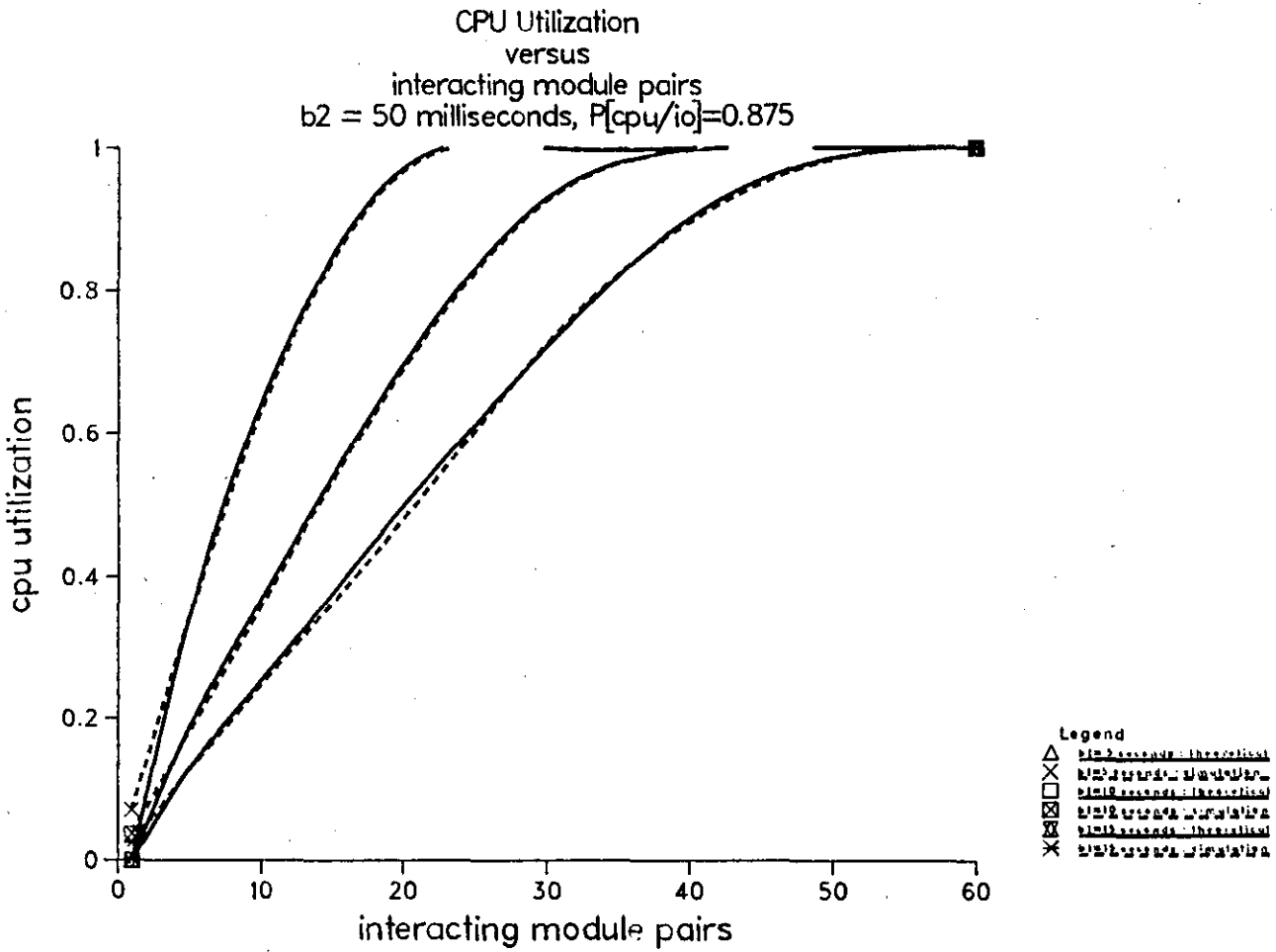


Fig. 6.34 CPU Utilization

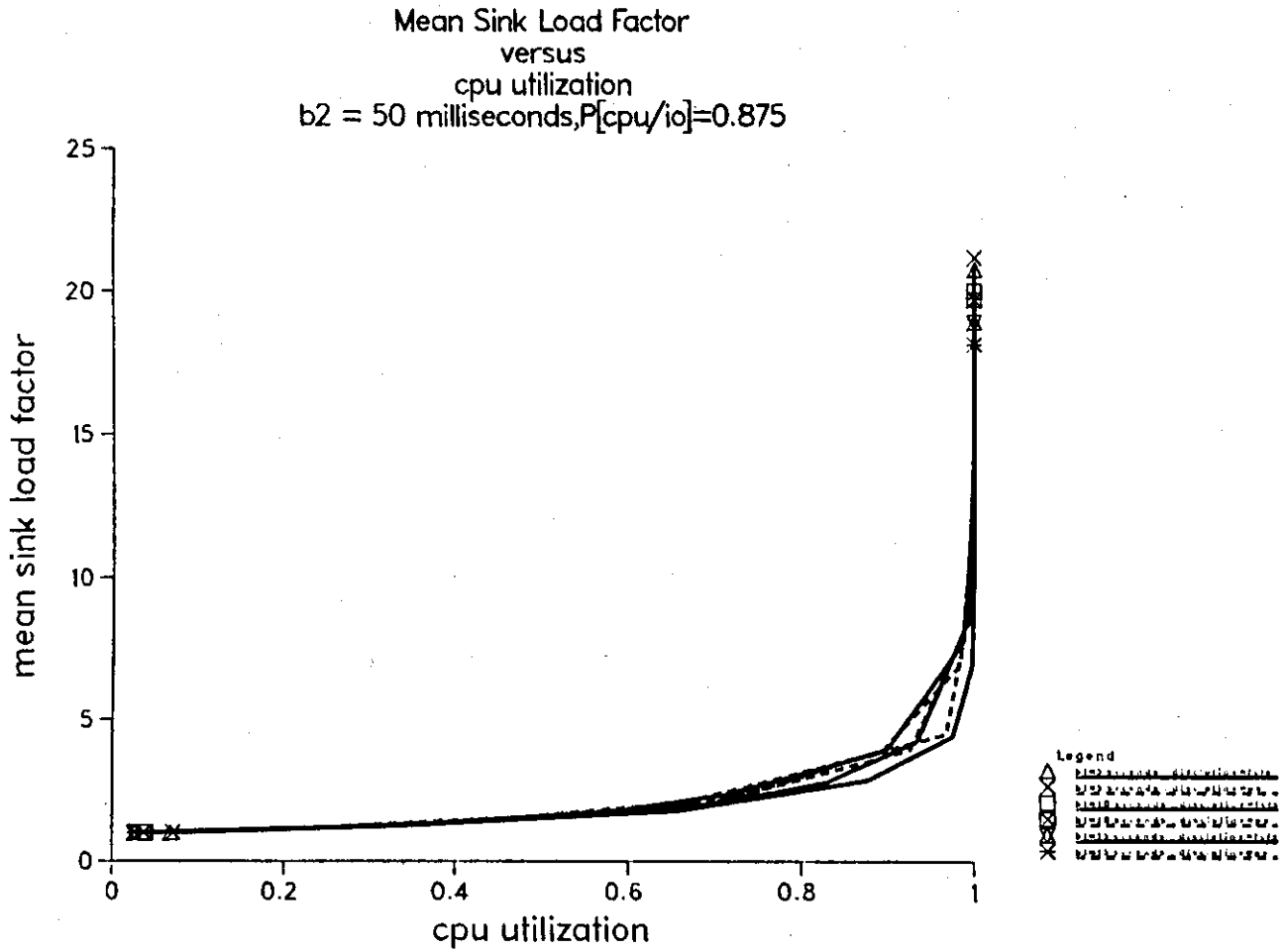


Fig. 6.35 Mean Sink Load Factor

theoretical and the simulation experiment are presented as well as the performance comparison of the simulation results for the various sink processor CPU scheduling algorithms. From these results it may be observed that the theoretical (solid lines) and the simulation (dashed lines) show close agreement for the range of values of b_m, p and q ($P\lceil CPU/IO \rceil$) used. These results were obtained using the Prime and Multics Computers of the computer centre of the university. The Multics computer system also supports the Tellagraph graphics system (ref. 6.47), which were used for plotting the results.

6.4 MODULE BEHAVIOUR

The results of section 6.3 summarise the behaviour of the sink processor system as the computational workload there increases. This behaviour has been characterised by the sink processor system response time and load factor as shown by the theoretical results of figs. 6.8 to 6.20, and the simulation results of figs. 6.26 to 6.30 and for the various queueing disciplines at the sink processor CPU of the time-shared central server model of fig. 6.5. As was explained earlier, the sink processor system response time (or load factor) is the most important performance measure in the dual processor CPU cache distributed computation system in that it is the dominant factor in determining whether the system modules may be processable in the dual processor environment. The results show how the load factor (or system response time) performance measure is bounded by the two ideal asymptotes: a horizontal asymptote in which the load factor is largely insensitive to the number of users currently getting service at the sink processor system, and a linear asymptote in which the system load factor rises linearly at a constant slope and in which each

user delays all other users by an amount of time equal to his own processing time, as previously explained. From this behaviour of the system load factor it may be suspected that the benefits of the dual processor system are maximum in this horizontal region. However, as the system load factor approaches the linear asymptote, substantial delay is introduced in the system and hence it may be expected that the source processor users planning to partition and assign their modules to the sink processor will have to "think twice"; or are discouraged from apportioning any of their computation to the sink processor. Furthermore, because of the feedback nature of the system, it may be expected that those source processors already with modules at the sink processor may wish to recall some of their modules to process at "home".

An alternative way of looking at the way the modules behave in the system is to examine the position of the minimum cut. The two extreme positions of the minimum cut, as explained in an earlier chapter, correspond to when all the modules are scheduled and assigned to either processor, i.e.

a) all modules assigned to the source processor

b) all modules assigned to the sink processor

In the case in which all or some of the modules are assigned to the sink processor, it may be expected that the position of the minimum cut will approach case (a), perhaps gradually at first and then rapidly later, as the load factor increases beyond the critical value of n (n_s). Such module behaviour may be compared with a queueing arrival process with impatience.

A dual processor CPU cache distributed computation system may be viewed as a queueing system with impatience because of the expected

reaction of the source processors and their modules in the feedback system. Impatient users may be described as either balking, reneging, or jockeying for a queue position (ref. 6.49, 6.50). If a customer (modules, user) decides not to join the queue upon arrival, then he is said to have balked. On the other hand, a customer may join the queue, but after waiting for a while lose patience and decide to leave, in which case he is said to have reneged. Both balking and reneging may be expected to exist in a dynamic dual processor CPU cache distributed computation system in which a feedback mechanism is present and used to control the system behaviour by broadcasting the level of the sink processor system load factor before, during, and after the module scheduling and assignment to processors.

In practice, it may be expected that users become discouraged when the queue is long and may not wish to risk waiting. Such a queueing system may be modelled as a birth-death process with limited waiting room (ref.6.49), in which an arriving customer does not join the queue if he sees K ahead of him. If K_q is the greatest queue length at which an arrival would not balk, then K_q is a random variable whose distribution B(n) is the same for all the users, (ref. 6.49), so that

$$B(n) = P[K_q \leq n]$$

Let

B(n-1) = the probability that the arrival refuses to join when n are in the queue.

then

B(n-1) defines the balking distribution, so that

$$P[\text{arrival joins the queue}] = P[K_q \geq n]$$

$$= 1 - P[K_q < n]$$

Hence

$$1 - P[K_q < n] = 1 - B(n-1), \text{ for } n \geq 0,$$

or

$$B(n-1) = P[K_q < n]$$

Let

$$G(n) = 1 - F(n)$$

$$\text{or } G(n-1) = 1 - F(n-1)$$

and $$\lambda_n = \lambda . G(n-1)$$

then

$$P[n] = P[o] . \prod_{i=1}^{n} \frac{\lambda_{i-1}}{\mu_i}$$

where λ_i and μ_i are state dependent birth and death rates, so that

$$P[n] = P[o] . \prod_{i=1}^{n} \frac{\lambda . G(i-1)}{\mu}$$

$$= \left(\frac{\lambda}{\mu} \right)^n . P[o] . \prod_{i=1}^{n-1} G(i-1), \text{ for } n>1$$

where

$$P[1] = \frac{\lambda}{\mu} . P[o]$$

and

$$P[o] = \frac{1}{\left[1 + \frac{\lambda}{\mu} + \sum_{n=2}^{\infty} \left(\frac{\lambda}{\mu} \right)^n . \prod_{i=1}^{n-1} G(i-1) \right]}$$

and

$$P[n+1] = \frac{\lambda}{\mu} . P[n] . G(n-1)$$

Balking functions which are dependent on the value of the sink processor system load factor may be used to model the behaviour of the modules in the dual processor CPU cache distributed computation system. Fig. 6.36

characterises the queue lengths of a system with balking for various values of G(n), given by

$$G(n) = \begin{cases} \dfrac{1}{1+b.n^{c}} & , \text{ for } o \leq n \leq k \\ 0 & n > k \end{cases}$$

where b and c are constants.

It shows how customers are discouraged from joining the queue as the value of k increases. Figs. 6.37, 6.38 and 6.39 show the simulation results in which the number of modules scheduled and assigned to the sink processor is observed as the load factor increases both uniformly, fig. 6.37 and fig. 6.38, and exponentially, fig. 6.39. As expected, it is seen that the source processors decide against having more modules process at the sink processor as the computational workload there increases. This module behaviour is due to the feedback mechanism of the system and is equivalent to the position of the minimum cut shifting further and further away from the source towards the sink, in the basic graph of the modular program, as shown in fig. 6.40.

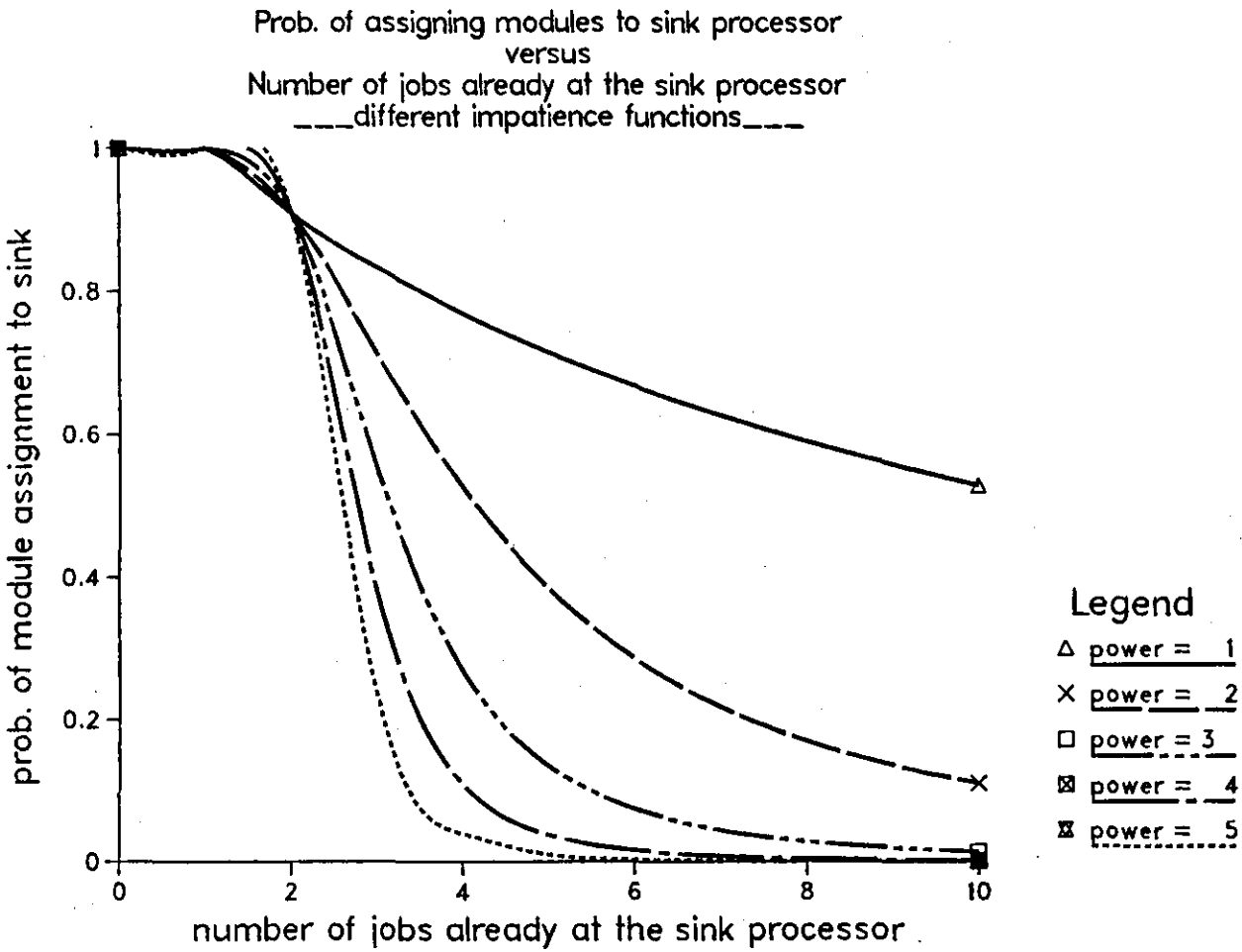


Fig. 6.36 Module Assignment

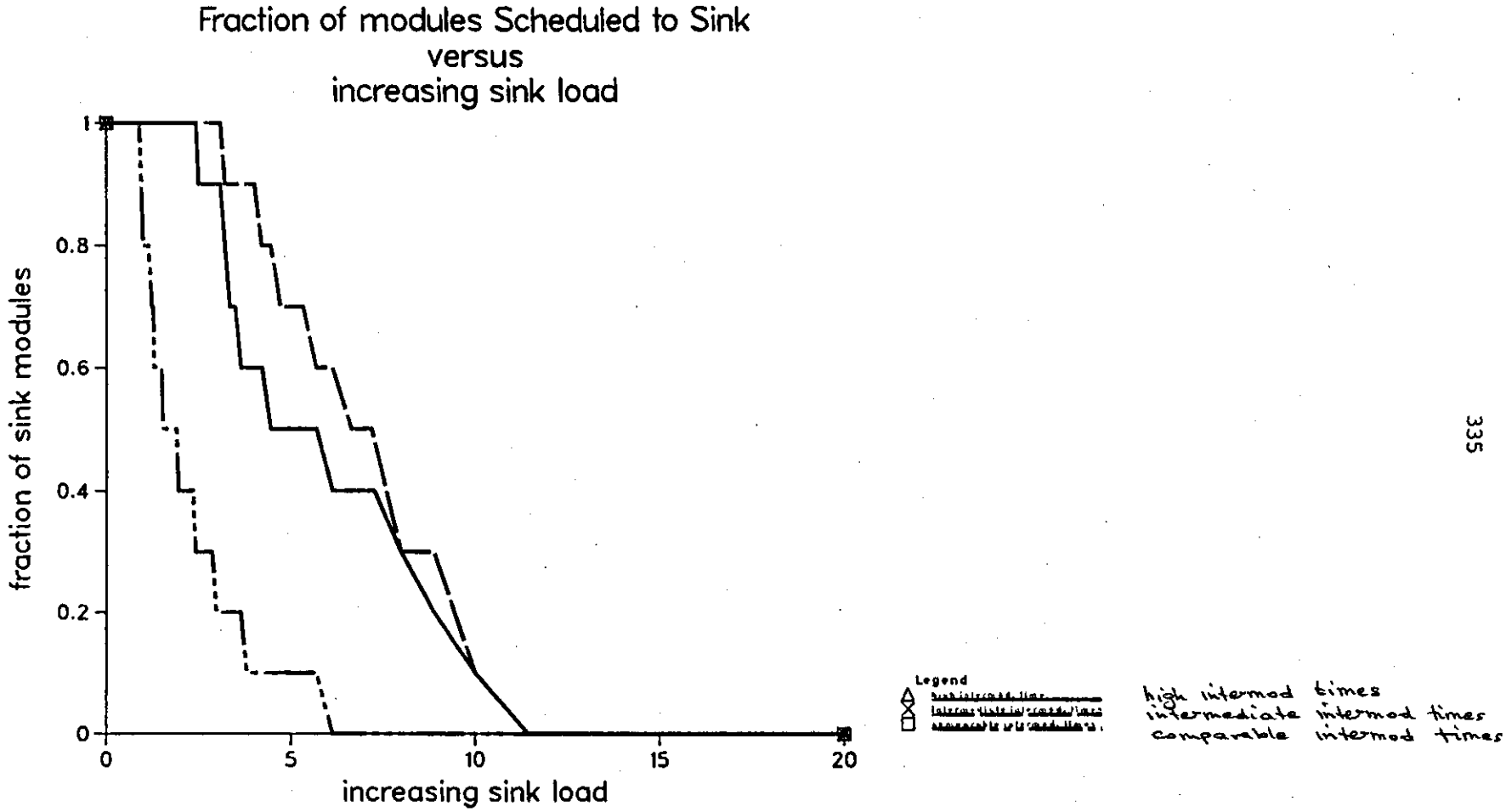


Fig. 6.37 Fraction of Scheduled Modules

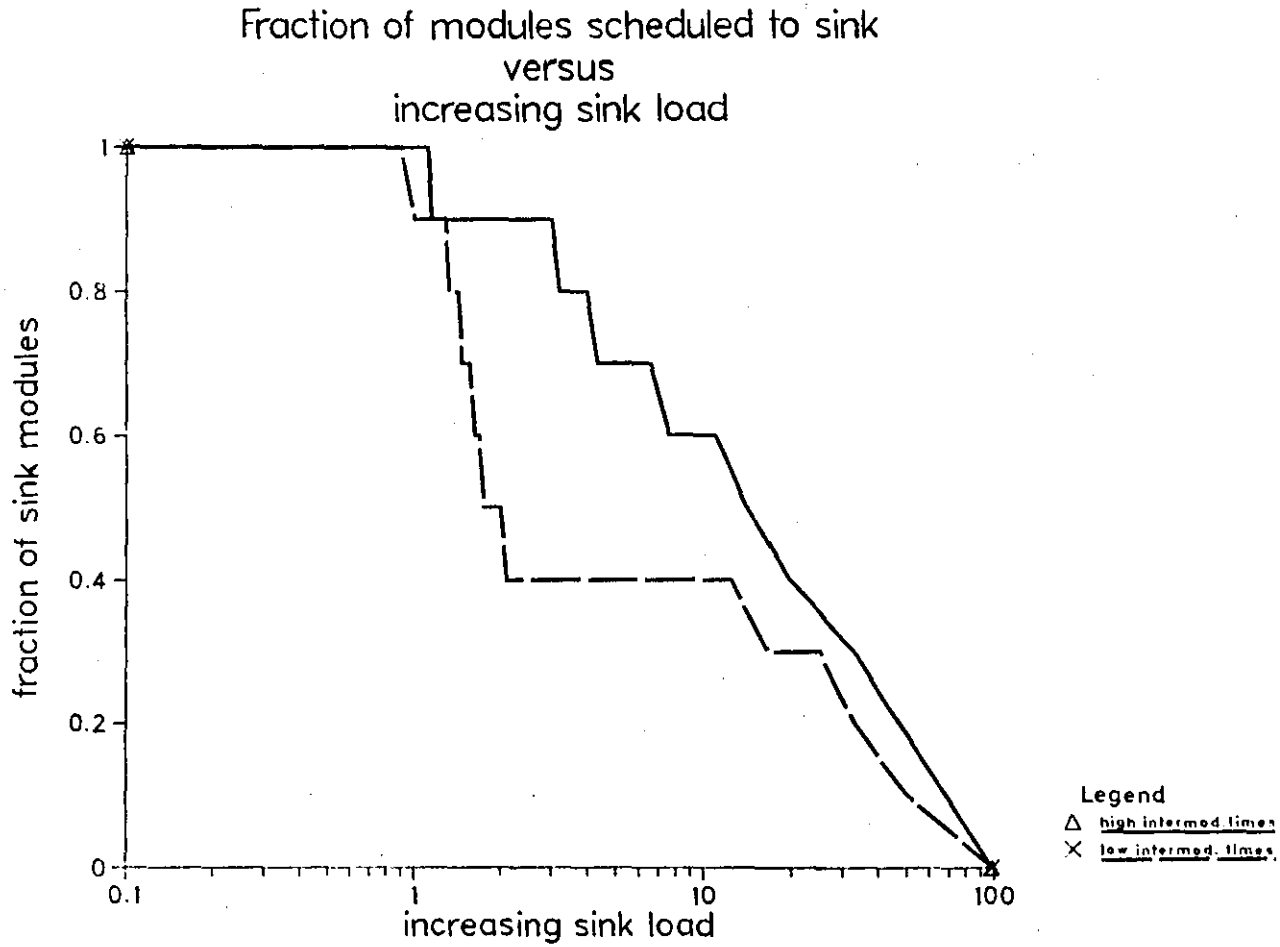


Fig. 6.38 Fraction of Scheduled Modules

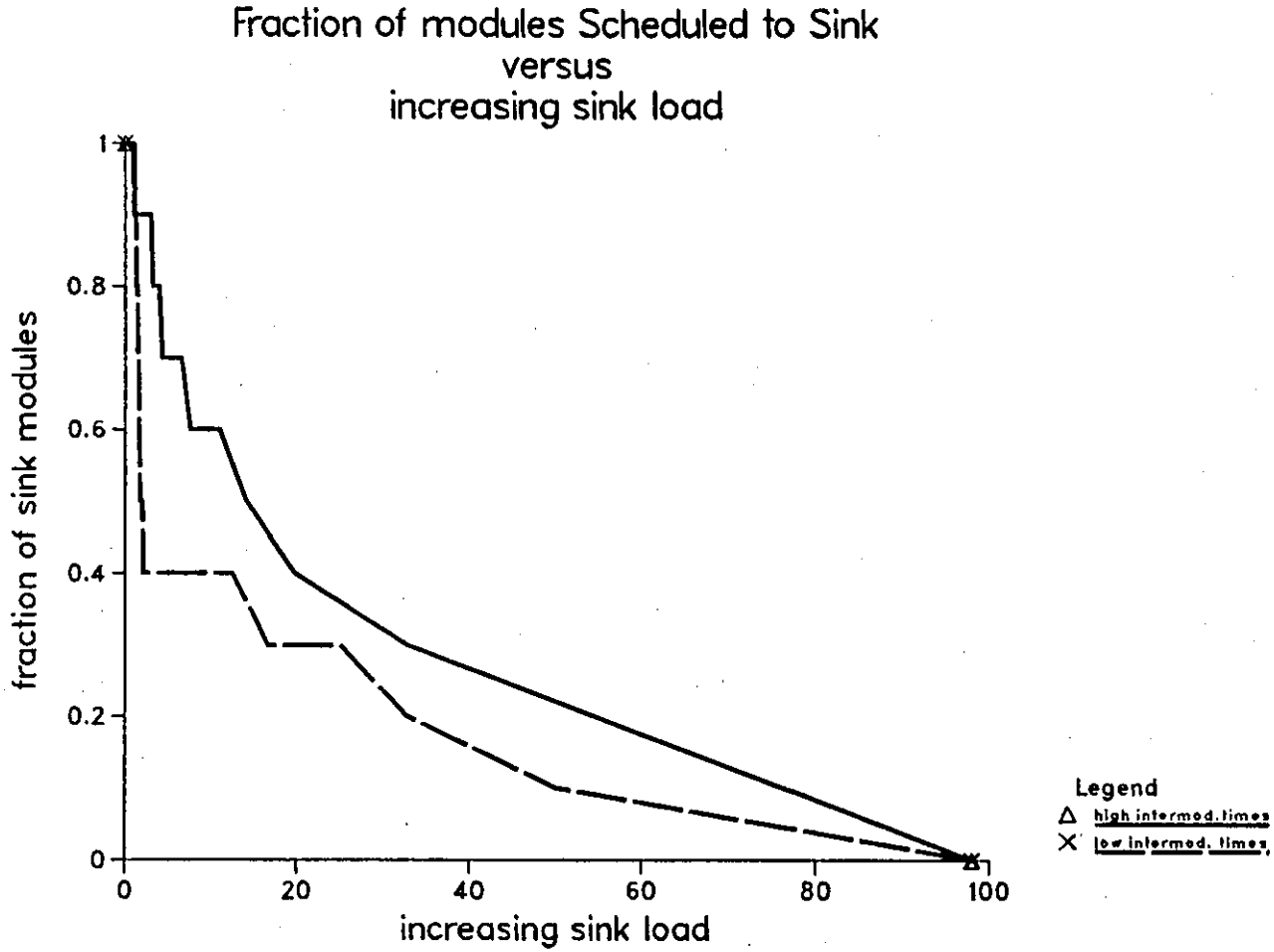


Fig. 6.39 Fraction of Scheduled Modules

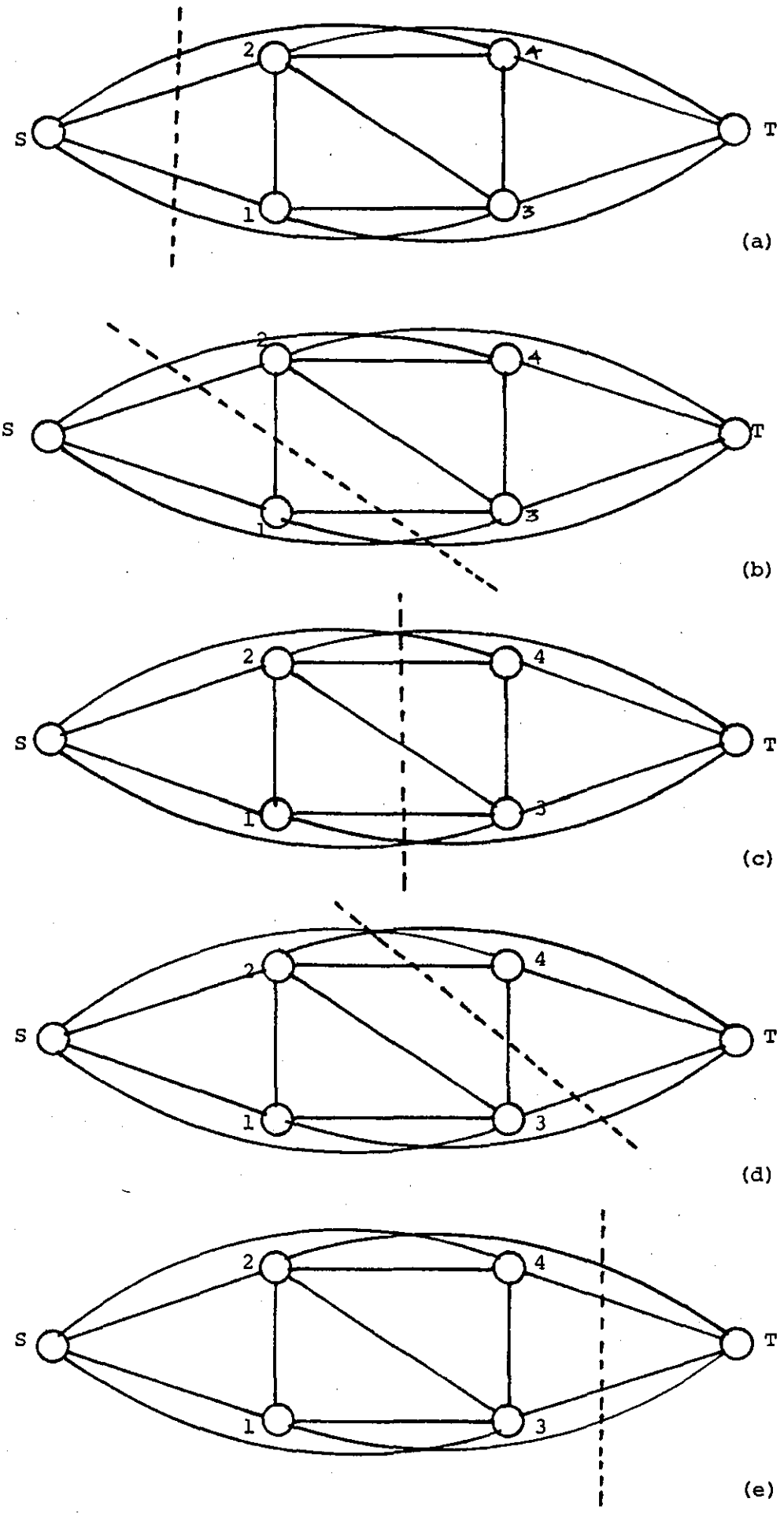


Fig. 6.40 Effect of Increasing Load Factor on the Location of the Minimum Cut

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 INTRODUCTION

The purpose of the research reported in this thesis is to investigate some aspects of performance in a distributed computation environment. In the type of distributed computation environment considered, the aim is to partition a single computation and assign the various portions of the computation to the various autonomous computers in the system. Various arrangements for distributed computation systems are available but this research was mainly concerned with the CPU cache system. A CPU cache system is primarily a dual processor distributed computation system in which the single computation is partitioned into two and assigned to the two autonomous computers in such a way that the total computation time of the problem is minimized. In order for the single computation to be partitioned and run thus, it is first organised in the form of program modules so that a module scheduler may then be used to partition and assign the program modules to the two processors using graph-theoretic concepts.

The CPU cache dual processor distributed computation system examined in this research is one in which an arbitrary number of the various small, but autonomous, and relatively less powerful computers (source processors) existing in a relatively small geographical area (LAN) decide to share their computations with a much larger and relatively more powerful computer (sink processor) also existing within the same LAN. The source processors may be microcomputers or minicomputers while the sink processor is primarily a large multi-accessed time-shared computer. All the various computers in the network (network-users) are interconnected by a wideband broadcast-

type bus topology communications subnet. Such a CPU cache dual processor distributed computation arrangement can exist in a LAN among network-users in a small factory complex or a research complex establishment.

Hence the main goal of the CPU cache dual processor system as explained above is to partition and assign some of the computations of the individual source processors to the sink processor, whenever possible. The decision of the source processors to partition and assign any computation to the sink processor is dependent on the currently existing workload at the sink processor. If the computational workload at the sink processor is low then the source processors are encouraged to assign some portions of their computation to the sink processor, and conversely. The decision of the source processors whether to assign any of their computations to the sink processor is effected by the existence of an internal feedback mechanism which is built within the broadcast-type distributed computation system in such a way that the sink processor broadcasts back to all the source processors, at regular intervals, the up-to-date value of the load factor or computational workload currently existing at the sink processor. In this way the source processors can update the values of the module and intermodule run times at the sink processor. By so doing, the source processors can calculate and decide the fraction of their total computation to schedule and assign to the sink processor during the module scheduling time.

Below we review briefly some of the main results of our investigation and we present suggestions for possible future development and further investigations.

7.2 REVIEW OF RESULTS

We showed that the concept of the CPU cache can be extended to the case in which many small computers with a CPU cache problem can be organised to share the resources of the computational power of a large computer coexisting in the same LAN. We examined and characterised the main performance measures in such a CPU cache dual processor distributed computation system. In particular we showed and quantified the three main factors that dominate the overall performance of such a system, i.e.

a) Module scheduling time

b) LAN delay performance

c) Computational workload at the sink processor

With regard to the module scheduling time we showed that the asymptotic space and time complexity of the module scheduling algorithm is a major performance consideration. The performance of two very different module schedulers were examined in detail. It was shown that as the number of program modules increases beyond a critical number of modules, the performance of the polynomial complexity maxflow-mincut module scheduling algorithm, at both the source processor (microcomputer) and sink processor (a large mainframe computer), is far superior to the corresponding exponential complexity enumerative module scheduling algorithm. But even in this case the scheduling time can run into many seconds, and even hours, of CPU time at either processor if the number of modules is large. Hence an even more efficient and faster polynomial complexity module scheduler, preferably O(n), is necessary in the CPU cache dual processor distributed computation system to keep the module scheduling time to a minimum. For a large number of modules, it was shown that it may be worth while to consider transporting all the modules to the sink

processor, whenever possible, so that the module scheduling may be done at the sink processor in order to reduce the overall scheduling time of the modules.

But the actual time taken to schedule the modules at either processor can be considered in both absolute and relative terms. A module scheduling time of one second may be both absolutely and relatively smaller than the module scheduling time of one hour, if the same module scheduler is employed to schedule ten modules. But if the value of the minimum cut has weights of ten seconds and ten hours respectively, then it may be seen that the use of a module scheduling time of one second is impractical compared to the other one. Hence, the absolute value of the module scheduling time cannot be taken in isolation and it may be found better to consider both the absolute and relative values of the scheduling time with respect to the value of the minimum cut. But the underlying assumption is that the module scheduling time is much smaller than the value of the minimum cut. A further underlying assumption is that the overall computation time of the CPU cache dual processable computation may be considered to be quite long since it may be pointless to partition and distribute a computation which lasts just a few seconds to process completely at the source processor. Hence, a problem for solution in a dual processor arrangement is bounded from below by the minimum computation time at the source processor and also by the relative and absolute values of the module scheduling time.

In a CPU cache dual processor distributed computation system it is necessary that, during the computation time, modules move freely between the two processors depending on the relative values of the computational workload at either processor. One way to accomplish this module movement

between the two processors is to keep running the module scheduler at regular intervals to determine whether the existing module assignment is acceptable. If the module assignment is found to be acceptable, then the modules continue with the same assignment, otherwise the scheduler must determine a new module assignment to the processors. In this case it is necessary that the module scheduling time is as small as possible, in absolute terms. With such a fast scheduler, and if the scheduling time is very small compared to the value of the minimum cut, dynamic module scheduling and assignment can be supported in the system. Also, with such a fast scheduler, and for a relatively few number of modules, all the cuts and module assignments may be pre-determined and stored in some form of a look-up table. In such a case, all the run-time environment routines have to do is to modify and up-date the table, at regular intervals, as the computational workload at the sink processor changes, and by so doing also determine the new module assignment. On the other hand, if the absolute module scheduling time is not small, or if the number of modules scheduled to the sink processor is large, then it is not possible to run the module scheduler very often. In this case, the scheduler may be run only once and the dynamic movement of the modules between the processors may be accomplished by examining the relative running times of the modules at the two processors so that those modules that are most affected by the increased load factor of the sink processor may now be moved to process at the respective source processor, and conversely if the load factor decreases.

With regard to the LAN delay performance, it was shown that the module and intermodule packets experience small delay when the channel traffic is low to medium. Hence the communications component of delay in the CPU cache

dual processor distributed computation system is not very significant. Furthermore, due to the internal feedback mechanism of the dual processor system, the actual number of users contending for the channel may be expected to remain small, most of the time.

Finally, the computational workload at the sink processor itself was characterised in terms of the sink processor system response time and load factor. The sink processor was considered to be a multi-accessed and time-shared and modelled as a central server time-shared resource system. The variation of the sink processor system response time and load factor was examined as the number of the source processors sending their computation to the sink processor increased. In particular, it was shown that the computational workload (or load factor) increased very slowly and gradually at first until a critical value of the number of source processors interacting with the sink processor was reached when the computational workload increased more rapidly. In the region of the load factor curve where the rate of rise of the computational workload with the number of source processors is slow and gradual the expected computational delay is very small and hence, in this region, the source processors derive maximum benefit from the computational crunching power of the sink processor system. In the deeply saturated region of the load factor, however, the expected delay for the modules scheduled and assigned to the sink processor system is substantial and it may be expected that the source processors will be discouraged from considering to assign much of their computation to the sink processor system to avoid long computational delay there. As explained earlier, the source processors are discouraged thus via the system feedback mechanism in which the level of the load factor is broadcast

back to all the source processors to let them know the volume of the computational workload there. Through such an internally built feedback mechamism, the source processors are constrained to schedule and do module assignment in the horizontal part of the load factor curve. As explained before, the load factor is a dimensionless quantity and represents a multiplication factor which the source processors must use to multiply the values of the module run times at the sink processor. As the load factor gradually increases with the computational workload, the feedback information forces the source processors to schedule and reassign their modules again so that fewer and fewer modules get assigned to the sink processor. This module behaviour due to this feedback is equivalent to the minimum-cut shifting further and further away from the source towards the sink, in the basic graph of the modular program, as explained before.

A major assumption in the derivation of the results which characterise the computational workload at the sink processor is that, on average, the multitude of the source processors may be considered to be homogeneous with respect to their computational requirements. Hence, if the average intermodule data request time was relatively long compared to the average service time for the intermodule request then the load factor curve show a relatively long horizontal portion. A long average intermodule data request time can be considered to be consistent with well designed program modules which are relatively autonomous and logically distinct. From the overall computation point of view, it is necessary that the program modules remain as autonomous and logically distinct as possible. In the way in which the dual processor system is organised in this research it is possible to have some modules which are not processable by the source processor. This is because such modules may already be existing on the

sink processor system, such as in the form of NAG library routines, as explained earlier. In such cases, it may be neither possible nor worthwhile to transport the routines from the sink processor because they may be implementation dependent and written in a different programming language.

7.3 SUGGESTIONS FOR FUTURE WORK

The following suggestions for future work along the lines of this research are now offered.

(i) As explained earlier, the efficiency and speed of a module scheduling algorithm is very important. A fast polynomial complexity module scheduler algorithm can go a long way in reducing the module scheduling and assignment delay in a dynamic dual processor system.

(ii) The minimum and maximum size of modules in terms of the computation time can give an indication as to whether a module is too small to be organised as a separate module.

(iii) A study of intermodule relationships in terms of their computational precedence relationships among the modules is also important.

(iv) The possibility of parallel execution of modules can also be investigated.

(v) The component of computational delay due to the contention for the primary memory at the sink processor system can be investigated since the sink processor system is likely to support multiprogramming.

```
1                    program sita (output) ;
2
3   $import
4         'qmill(pl1)':qmill$
5    const
6     n= 12 ;
7     unscanned= -12 ;
8     infinity=10000 ;
9
10  type
11      node=1..n ;
12      xnode=-n..n ;
13      vector=array[node] of xnode ;
14      matrix=array[node,node] of real ;
15      whichway=(push,pull) ;
16
17   var
18      s,t : node  ;
19      c,f : matrix ;
20      initial,final,cputime : real  ;
21      x : integer  ;
22
23
24  function qmill (var x : integer) : real ;external  ;
25  procedure generate ( var c : matrix) ;
26    var
27      i,j,cost,runtime : integer  ;
28    begin
29      for i:=1 to n do
30        for j:=1 to n do
31            if (i=j)  or (i=t) or (j=s)   or (i=s)  and  (j=t)
32               then c[i,j] := 0  ;
33      for  j:= (s+1)  to  (n-1)  do
34            begin
35                runtime := 10 * j mod 61  ;
36                c[j,t] := runtime  ;
37                c[s,j] := 61 - 0.8*runtime  ;
38            end  ;
39        i := s+1  ;
40           while  (i<=(n-2))  do
41                    begin
42                        j := i + 1  ;
43                        while (j<=(n-1))  do
44                           begin
45                              cost := i mod 11  ;
46                              c[i,j] := cost    ;
47                              c[j,i] := cost   ;
48                              j := j + 1
49                            end  ;
50                        i := i+ 1
51                    end  ;
52    end  ;
53
54   procedure maxflow (s,t:node ;c:matrix ; var f:matrix) ;
55    var  refnode :node ;  (*node with least excess capacity  *)
56     minpotential :real ;  (*excess capacity of the ref node *)
57     layer :vector ;  (*the layered network is defined by this array *)
58     r : real ;
59     i,j :node ;  (*indices *)
60     minimumcut : real  ;
```

```
 61
 62    function  min (x,y:real):real ;
 63     (*determines the minimum amount of flow *)
 64     begin
 65      if x<y
 66      then min :=x
 67       else min :=y  ;
 68     end ;
 69
 70    procedure walk (i:node ) ;
 71     (*traverse the layered network from t,inverting layer numbers.*)
 72     var  j:node ;  li :xnode ;
 73      begin
 74       layer[i] := -layer[i] ;
 75       li :=layer[i] ;
 76       for j:= 1 to n do
 77        if (j<>s) and (-layer[j]=li-1) and ((f[j,i]<c[j,i]) or (f[i,j]>0
\c))
 78         then walk (j)  ;
 79      end ;  (*walk *)
 80
 81    function layeringpossible : boolean ;
 82      (*is it possible to build a layered network, if so build it *)
 83     var  i,j :node ;
 84      k:0..n ;
 85      emptylayer :boolean ;
 86      begin
 87       k :=0  ;    (*k keeps   track of layer being built *)
 88       for i:= 1 to n do
 89        layer[i] :=unscanned ;  (*initialize each node *)
 90        layer[s] :=k ;  (*source node is in  layer 0 *)
 91       repeat
 92        k :=k+1 ;   (*now locate all nodes in layer k *)
 93        emptylayer := true ;  (*an empty layer stops the algorithm *)
 94        for i:= 1 to n do
 95         if -layer[i] = k-1
 96          then
 97           (*i is in layer k-1,its neighbors may be in layer k  *)
 98           for j := 1 to n  do  (*check each node adjacent to i *)
 99            if (layer[j]=unscanned) and ((f[i,j]<c[i,j]) or (f[j,i]>0))
100             then
101              begin
102               layer[j] := -k ;
103               emptylayer := false
104              end ;
105      until  (layer[t] <> unscanned ) or emptylayer ;
106       layeringpossible := not  emptylayer ;
107       walk (t) ;   (*prune off the dead ends  *)
108      end ;  (*layeringpossible *)
109
110   procedure findrefnode  (i:node)  ;
111    (*traverse the layered network from t, seeking the ref node *)
112    var  j :node ;
113     li,lj :xnode ;
114     incap, outcap : real ;
115     begin
116      li  := layer[i] ;
117      incap :=0  ;
118      outcap := 0 ;
119      for j :=1 to n do
120       (*examine each node adjacent to i *)
```

```
      begin
       lj :=layer[j] ;
       if (lj =li-1) and (j<>s) and ((f[j,i]<c[j,i]) or (f[i,j]>0))
        then  findrefnode (j)  ;
       if  lj = li-1
        then incap :=incap + (c[j,i]-f[j,i])+f[i,j]  ;
       if lj =li+1
        then outcap :=outcap + (c[i,j]-f[i,j])+f[j,i]
      end ;
      if  (i<>s) and (i<>t) and  (min (incap,outcap)<minpotential)
      then
       (*node i has smaller potential than the current ref node *)
       begin
        minpotential := min (incap, outcap)  ;
        refnode := i  ;
       end ;
    end  ;  (*findrefnode *)

  procedure  pushpull (i :node ; flowleft :real ; p : whichway ) ;
   (*augment the flow thro' i by pushing or pulling minpotential units
\c *)
   var  j, k1, k2, layersought : 0..n ;
    begin
    j := 0 ;
     while (flowleft >0) and (j<n) do
      begin
       j :=j+1 ;
       if p=push
        then
         begin
          k1:=i ;
          k2:=j ;
          layersought:=layer[i]+1
         end
        else
         begin
          k1:=j ;
         k2:=i ;
          layersought :=layer[i]-1
         end ;
        r:=min (flowleft,c[k1,k2]-f[k1,k2]+f[k2,k1] )  ;
         (*amount of flow to move *)
        if  (r>0) and (layer[j]=layersought)
         then
          begin  (*push/pull some flow to/from an adjacent  layer *)
           flowleft :=flowleft -r  ;
           f[k1,k2] :=f[k1,k2]+r-min(r,f[k2,k1])  ;
            (*augment positive flow *)
           f[k2,k1] := f[k2,k1]-min (r,f[k2,k1])  ;
            (*push reverse flow backwards *)
           if (j<>s) and (j<>t)
            then  pushpull (j,r,p)
          end
      end
    end  ;(*pushpull *)

  begin   (*maxflow *)
   for i :=1 to n do
    for j :=1 to n do
     f[i,j] :=0  ;  (*initialy no flow *)
     f[s,t] :=c[s,t]  ;  (*if an s_t link exists , saturate it *)
```

```
      minimumcut := 0 ;
     while layeringpossible do   (*assign nodes to layers  *)
      begin
       minpotential := infinity  ;
       findrefnode (t)  ;   (*find the reference node *)
       pushpull (refnode,minpotential,push) ;  (*push flow towards the
\csink*)
       pushpull (refnode,minpotential,pull) ; (*pull flow from source*)
       minimumcut := minimumcut + r ;
      end  ;
    writeln ('minimum cut =  ',minimumcut:12:3)  ;
   end  ;  (*maxflow*)

begin    (*main program *)
  writeln ('maxflow___mincut scheduling ')  ;
  writeln ('**************************')  ;
  s :=1 ;
  t := n ;
  initial := 0  ;
  final := 0  ;
  initial := qmill(x)  ;
  generate (c)  ;
  maxflow (s,t,c,f)  ;
  final := qmill(x)  ;
  cputime := final - initial  ;
  writeln (' time =  ',cputime:12:3)  ;
end.
```

```
 1  program enumerate (output)  ;
 2  $import
 3
 4        'qmill(pl1)':qmill$
 5  const
 6      n = 10  ;
 7
 8  var
 9    x : integer  ;
10     initial,final,cputime : real  ;
11
12  function qmill (var x : integer) : real ; external  ;
13
14  procedure allcuts  ;
15    var
16      i,j,ja,jb,k : integer  ;
17      z,zz,s,t,best,total : integer  ;
18      out : array [1..n] of integer  ;
19      comp : array [1..n] of integer  ;
20      bout : array [1..n] of integer  ;
21      bcomp : array [1..n] of integer  ;
22          t1 : array [1..n] of integer  ;
23          t2 : array [1..n] of integer  ;
24          c  : array [1..n,1..n] of integer  ;
25
26    function power : integer  ;
27      var
28        i,answer : integer  ;
29      begin
30          answer := 1  ;
31          for i := 1 to n do
32              answer := answer * 2  ;
33          power := answer
34      end  ;
35
36    begin
37        for s := 1 to n do
38            t1[s] := 61 - 8 * s mod 61  ;
39        for s := 1 to n do
40            t2[s] := 10 * s mod 61  ;
41        for s := 1 to n do
42          for t := 1 to n do
43            if s=t
44               then c[s,t] := 0
45               else c[s,t] := s mod 11  ;
46        best := maxint  ;
47        for i := 0 to power do
48            begin
49                j := i  ;
50                for ja := 1 to n do
51                    begin
52                        jb := j div 2  ;
53                        if j <> 2*jb
54                           then
55                              begin
56                                  out[ja] := ja  ;
57                                  comp[ja] := 0  ;
58                              end
59                           else
60                              begin
```

```
                                    out[ja] := 0  ;
                                    comp[ja] := ja  ;
                                 end  ;
                            j := jb  ;
                       end  ;
                    total := 0  ;
                    for z := 1 to n do
                        begin
                            if out[z] <> 0
                               then
                                  begin
                                      total := total + t2[z]  ;
                                      for zz := 1 to n do
                                          if comp[zz] <> 0
                                              then  total:= total + c[z,zz
\c]
                                  end
                              else total := total + t2[z]  ;
                        end  ;
                     if best>total
                        then
                           begin
                               best := total  ;
                               for z := 1 to n do
                                   begin
                                       bout[z] := out[z]  ;
                                       bcomp[z] := comp[z]  ;
                                   end  ;
                           end
              end  ;
           writeln ('minimum cut =  ',best)  ;
       end  ;

begin  (* main program *)
    writeln (' enumerated cuts as follows')  ;
    writeln ('**************************')  ;
    initial := 0  ;
    final := 0  ;
    initial := qmill(x)  ;
    allcuts  ;
    final := qmill(x)  ;
    cputime := final - initial  ;
    writeln ('time =  ',cputime:12:3)  ;
end.
```

Appendix C Channel Delay Performance

```
1  program approxchannel (output)  ;
2
3  $import
4        'g05ccf(fortran)':g05ccf  ;
5        'g05caf(fortran)':g05caf  ;
6        'g05dbf(fortran)':g05dbf$
7
8  const
9      b1 = 200  ;
10     a = 0.001  ;  (* prop. delay = 5ys, P=5Kb, C=1Mbps *)
11     nq = 1  ;
12
13 type
14    elementptr = ^element  ;
15    element = record
16                   time : real  ;
17                   param : integer  ;
18                   next : elementptr
19              end  ;
20
21 var
22   clock : real  ;
23   totallength : integer  ;
24   i : integer  ;
25   first,last,avail : elementptr  ;
26   queues : array [1..nq] of
27                record
28                     numberservers : integer  ;
29                     meanservice : real  ;
30                     length : integer  ;
31                     timelengthchanged : real  ;
32                     sumtimelength : real  ;
33                     sumbusytime : real  ;
34                     numbercompletions : integer  ;
35                     bt : real  ;
36                     tl : real  ;
37                     nc : real  ;
38                     btsq : real  ;
39                     btxcl : real  ;
40                     ncsq : real  ;
41                     ncxcl : real ;
42                     tlsq : real  ;
43                     tlxcl : real  ;
44                     tlxnc : real  ;
45                end  ;
46   run,numberevents,eventlimit,eventmax : integer  ;
47   noeventsduringcycles,numbercycles,nocycm1 : integer  ;
48   timecyclestarted,cyclelength,
49        sumcl,sumclsq,varcl,dcl : real  ;
50   util,dutil,varbt,covarbtcl,vart : real  ;
51   tput,dtput,varnc,covarnccl : real  ;
52   ql,dql,vartl,covartlcl : real  ;
53   qt,dqt,covartlnc : real  ;
54   dummymeanvalue,negexpomean : real  ;
55   negexpotime : real  ;
56   meaninterarrival : real  ;
57   v1,v2 : integer  ;
58
59 procedure g05ccf  ; external  ;
60
```

```
 61  function g05caf ( var dummymeanvalue : real) : real ; external ;
 62
 63  function g05dbf (var negexpomean : real) : real ; external  ;
 64
 65  function min (v1,v2 : integer) : integer  ;
 66     begin
 67         if v1<v2
 68            then
 69               min := v1
 70            else
 71               min := v2
 72     end  ;  (*  min  *)
 73
 74  procedure insertevent (t : real ; q : integer)  ;
 75
 76     (* insertevent adds event at time t for param q to list *)
 77
 78     var
 79       temp, n, l : elementptr  ;
 80
 81     begin
 82         if avail = nil
 83            then
 84               new(temp)
 85            else  (* previously used storage available *)
 86               begin
 87                    temp := avail  ;
 88                    avail := avail^.next
 89               end  ;
 90            temp^.time := t  ;
 91            temp^.param := q  ;
 92            if first = nil
 93               then
 94                  begin  (* list was empty *)
 95                      first := temp  ;
 96                      last := temp  ;
 97                      temp^.next := nil  ;
 98                  end
 99               else
100                  if t<first^.time
101                     then
102                        begin   (* insert at beginning of list *)
103                            temp^.next := first  ;
104                            first := temp  ;
105                        end
106                     else
107                        if t>=last^.time
108                           then
109                              begin  (* insert at end of list *)
110                                  last^.next := temp  ;
111                                  last := temp  ;
112                                  temp^.next := nil  ;
113                              end
114                           else
115                              begin  (* insert somewhere in middle of li
\cst *)
116                                  l := first  ;
117                                  while t>=l^.next^.time do
118                                      l := l^.next  ;
119                                  temp^.next := l^.next  ;
120                                  l^.next := temp  ;
```

```
121                                 end  ;
122     end  ;  (* insertevent *)
123
124  procedure removefirstevent (var t : real ; var q : integer)  ;
125     (* removefirstevent returns time t and param q of first event *)
126
127     var
128       temp : elementptr  ;
129
130     begin
131         if first = nil
132            then
133               begin
134                   writeln (' removefirstevent --- empty list') ;
135                   (*  halt *)
136               end
137            else
138               begin
139                   t := first^.time  ;
140                   q := first^.param  ;
141                   temp := first  ;
142                   first := first^.next  ;
143                   if first = nil
144                      then
145                         last := nil  ;
146                   temp^.next := avail  ;
147                   avail := temp  ;
148               end
149     end  ; (*  removefirstevent *)
150
151  procedure complete (q : integer)  ;
152     (*  handles completion of a job at queue q  *)
153   begin
154     with queues[q] do
155       begin   (* statistics  *)
156         numbercompletions := numbercompletions + 1  ;
157         sumtimelength := sumtimelength+(clock-timelengthchanged)*length
\c ;
158         sumbusytime := sumbusytime + (clock-timelengthchanged)*
159                                        min(length,numberservers)  ;
160         timelengthchanged := clock  ;
161             (*  mechanics  *)
162         length := length - 1  ;
163         if length >= numberservers
164            then
165               begin
166                   g05ccf  ;
167                   negexpotime := g05dbf(meanservice)  ;
168                   insertevent (clock + a + negexpotime,q)  ;
169               end
170         end
171     end  ;  (*  complete  *)
172
173  procedure arrive (q : integer)  ;
174     (*  handles arrival of a job at queue q  *)
175     begin
176         with queues[q] do
177            begin
178                (*  statistics  *)
179                sumtimelength := sumtimelength+(clock-timelengthchanged)
\c*
180                                                                  length  ;
```

```
181                sumbusytime := sumbusytime + (clock-timelengthchanged)*
182                                             min(length,numberservers) ;
183                timelengthchanged := clock  ;
184                 (*  mechanics  *)
185                length := length + 1  ;
186                if length <= numberservers
187                   then
188                      begin
189                          g05ccf  ;
190                          negexpotime := g05dbf(meanservice)  ;
191                          insertevent (clock+a+negexpotime,q)  ;
192                      end
193             end
194    end  ;   (*  arrive  *)
195
196  function endcycle : boolean  ;
197     (* determines whether at end of regeneration cycle. If so, *)
198     (* endcycle updates accumulators. *)
199     var
200       q : integer  ;
201     begin
202         if (totallength = 0) and (numberevents>0)
203            then
204               begin
205                   endcycle := true  ;
206                   noeventsduringcycles := numberevents  ;
207                   numbercycles := numbercycles + 1  ;
208                   cyclelength := clock - timecyclestarted  ;
209                   timecyclestarted := clock  ;
210                   sumcl := sumcl + cyclelength  ;
211                   sumclsq := sumclsq + sqr(cyclelength)  ;
212                   for q := 1 to nq do
213                     with queues[q] do
214                        begin
215                            sumtimelength := sumtimelength +
216                                   (clock-timelengthchanged)*length  ;
217                            sumbusytime := (sumbusytime +
218                                   (clock-timelengthchanged)*
219                                   min(length,numberservers))/numberserv
\cers ;
220                            timelengthchanged := clock  ;
221                            bt := bt + sumbusytime  ;
222                            tl := tl + sumtimelength  ;
223                            nc := nc + numbercompletions  ;
224                            btsq := btsq + sqr(sumbusytime)  ;
225                            btxcl := btxcl + sumbusytime*cyclelength  ;
226                            sumbusytime := 0.0  ;
227                            ncsq := ncsq + sqr(numbercompletions)  ;
228                            ncxcl := ncxcl+numbercompletions*cyclelength
\c  ;
229                            tlsq := tlsq + sqr(sumtimelength)  ;
230                            tlxcl := tlxcl + sumtimelength*cyclelength
\c;
231                            tlxnc := tlxnc+sumtimelength*numbercompletio
\cns ;
232                            numbercompletions := 0  ;
233                            sumtimelength := 0.0  ;
234                        end
235                end
236             else
237                endcycle := false
238    end  ; (*  endcycle  *)
239
240  begin     (*  main program  *)
```

```
241     (*  initialization  *)
242     meaninterarrival := 0.00625 ;
243     avail := nil  ;
244     eventlimit := 100  ;
245     for run := 1 to 3 do
246       begin
247           first := nil  ;
248           last := nil  ;
249           clock := 0.0  ;
250           numberevents := 0  ;
251           numbercycles := 0  ;
252           timecyclestarted := 0.0  ;
253           sumcl := 0.0  ;
254           sumclsq := 0.0  ;
255           eventlimit := 10*eventlimit  ;
256           eventmax := 2*eventlimit  ;
257           for i := 1 to nq do
258             with queues[i] do
259                begin
260                    length := 0  ;
261                    timelengthchanged := 0.0  ;
262                    sumtimelength := 0.0  ;
263                    sumbusytime := 0.0  ;
264                    numbercompletions := 0  ;
265                    bt := 0.0  ;
266                    tl := 0.0  ;
267                    nc := 0.0  ;
268                    btsq := 0.0  ;
269                    btxcl := 0.0  ;
270                    ncsq := 0.0  ;
271                    ncxcl := 0.0  ;
272                    tlsq := 0.0  ;
273                    tlxcl := 0.0  ;
274                    tlxnc := 0.0  ;
275                end  ;
276           queues[1].numberservers := 1  ;
277           queues[1].meanservice := 1.0/b1  ;
278           totallength := 0  ;
279           g05ccf  ;
280           negexpotime := g05dbf(meaninterarrival)  ;
281           insertevent (negexpotime,0)  ;
282            (*  run  *)
283           while (first<>nil) and (numberevents<eventmax)
284                                  and
285                 ((numberevents<eventlimit) or not endcycle)
286                                  do
287               begin
288                   numberevents := numberevents + 1  ;
289                   removefirstevent (clock,i)  ;
290                   if i = 0
291                      then
292                         begin
293                             totallength := totallength + 1  ;
294                             arrive(1)  ;
295                             g05ccf  ;
296                             negexpotime := g05dbf(meaninterarrival)  ;
297                             insertevent (clock + negexpotime,0)  ;
298                         end
299                      else
300                         begin
```

```
301                          complete(i)  ;
302                          if i<>nq
303                             then
304                                arrive (i + 1)
305                             else
306                                totallength := totallength - 1  ;
307                       end
308             end  ;
309
310        (*  print statistics  *)
311      writeln  ;
312      writeln ('no. of events :',numberevents:8,
313               '  simulated time :',clock:10:3)  ;
314     writeln  ;
315     writeln ('queue  util  tput queuelength  queueingtime') ;
316     if numbercycles>1
317        then   (*  produce confi. interval estimates  *)
318           begin
319               cyclelength := sumcl/numbercycles  ;
320               nocycm1 := numbercycles - 1  ;
321               varcl := (sumclsq-sqr(sumcl)/numbercycles)/nocycm1  ;
322               for i := 1 to nq do
323                 with queues[i] do
324                    if nc>0
325                       then
326                          begin
327                              util := bt/sumcl  ;
328                              varbt := (btsq-sqr(bt)/numbercycles)/
329                                                      nocycm1  ;
330                              covarbtcl:=(btxcl-bt*sumcl/numbercycles)/
331                                                        nocycm1  ;
332                              dutil:=1.645*sqrt((varbt-2*util*covarbtcl
333                                    +sqr(util)*varcl)/numbercycles)/
334                                                         cyclelength  ;
335                              tput := nc/sumcl  ;
336                              varnc := (ncsq-sqr(nc)/numbercycles)/
337                                                      nocycm1  ;
338                              covarnccl:=(ncxcl-nc*sumcl/numbercycles)/
339                                                        nocycm1  ;
340                              dtput:=1.645*sqrt((varnc-2*tput*covarnccl
341                                     +sqr(tput)*varcl)/numbercycles)/
342                                                          cyclelength  ;
343                              ql := tl/sumcl  ;
344                              vartl := (tlsq-sqr(tl)/numbercycles)/
345                                                      nocycm1  ;
346                              covartlcl:=(tlxcl-tl*sumcl/numbercycles)/
347                                                        nocycm1  ;
348                              dql:=1.645*sqrt((vartl-2*ql*covartlcl
349                                  +sqr(ql)*varcl)/numbercycles)/
350                                                       cyclelength  ;
351                              qt := tl/nc  ;
352                              covartlnc:=(tlxnc-tl*nc/numbercycles)/
353                                                        nocycm1  ;
354                              dqt:=1.645*sqrt((vartl-2*qt*covartlnc
355                                  +sqr(qt)*varnc)/numbercycles)/
356                                                       (nc/cyclelength)  ;
357                              writeln ('UPPER',
358                                       util+dutil:12:3,tput+dtput:11:3
\c,
359                                       ql+dql:13:3,qt+dqt:14:3)  ;
360                              writeln (i:5,util:12:3,tput:11:3,
```

```
                                        ql:13:3,qt:14:3)  ;
                                writeln ('LOWER',
                                        util-dutil:12:3,tput-dtput:11
\c:3,
                                        ql-dql:13:3,qt-dqt:14:3)  ;
                            end  ;
                        writeln  ;
                        writeln('no. of cycles :',numbercycles:8)  ;
                        if noeventsduringcycles<>numberevents
                           then
                              writeln ('no. of discarded events :',
                                      numberevents-noeventsduringcycles:8)
\c ;
                        writeln ('ave. no of events :',
                                 noeventsduringcycles/numbercycles:10:3)
\c;
                        dcl:=1.645*sqrt(varcl/numbercycles)  ;
                        writeln ('ave. length :',cyclelength:10:3,
                                 '  C.I. : (',cyclelength-dcl:10:3,',',
                                 cyclelength+dcl:10:3,')')
                end
             else     (*   produce point estimates only  *)
                for i := 1 to nq do
                  with queues[i] do
                     if numbercompletions+trunc(nc)>0
                        then
                           begin
                              sumtimelength := sumtimelength + tl  ;
                              sumbusytime := sumbusytime+bt*numberservers
\c ;
                              numbercompletions:=numbercompletions+trunc(
\cnc) ;
                              sumtimelength:=sumtimelength +
                                 (clock-timelengthchanged)*length  ;
                              sumbusytime := sumbusytime +
                                 min(length,numberservers)*
                                 (clock-timelengthchanged)  ;
                              writeln (i:5,
                                       sumbusytime/(numberservers*clock):1
\c2:3,
                                       numbercompletions/clock:11:3,
                                       sumtimelength/clock:13:3,
                                       sumtimelength/numbercompletions:14:
\c3)  ;
                           end  ;
          (*  put leftover events on avail list  *)
              if first<>nil
                 then
                    begin
                       last^.next := avail  ;
                       avail := first  ;
                    end
       end
    end.
```

```
 1                    PROGRAM sabini (output) ;
 2   CONST
 3    n=12 ;
 4    loopmax = 50 ;
 5    unscanned=-12 ;
 6    infinity=10000 ;
 7
 8  TYPE
 9      node=1..n ;
10      xnode=-n..n ;
11      vector=array[node] of xnode ;
12      matrix=array[node,node] of real ;
13      WhichWay=(push,pull) ;
14      A5=ARRAY [1..30] OF INTEGER ;
15      cutset = array  [1..50]  of real  ;
16
17   VAR
18      i,s,t : node ;
19      J : NODE ;
20      c,f : matrix ;
21      x,y,FlowLeft :real ;
22      p: WhichWay ;
23      B:A5 ;
24      INT:INTEGER ;
25      countone,counttwo,countzero : integer  ;
26      loop : integer  ;
27      count,avem,avelinks : integer  ;
28      a1,p1,p2 : real  ;
29      MinCut,avesource,avesink ,dodsn  : real  ;
30      parameter : integer  ;
31      MinimumCut : real  ;
32      ratio : real  ;
33      xx,yy : cutset  ;
34      nn,mode : integer  ;
35      mean,meanone,meantwo,meanthree : REAL  ;
36
37    PROCEDURE TIMDAT (VAR M:A5  SHORT ; VAR N:INTEGER  SHORT) ; EXTERN ;
38       (* returns cpu and i/o time parameters *)
39
40  PROCEDURE C1051n  ;  EXTERN  ;
41       (* the graph plotting device *)
42
43  PROCEDURE DEVEND  ;  EXTERN  ;
44       (* closes graph plotting routines *)
45
46  PROCEDURE GRAF (VAR xx,yy:cutset short; VAR nn,mode:integer short);EXT
\cERN;
47        (* graph plotting  GINO routine *)
48
49  PROCEDURE G05CCF ; EXTERN ;
50     (* sets the generator G05CAF to non_repeatable initial state *)
51
52  PROCEDURE G05CBF (VAR parameter :integer short)  ; EXTERN  ;
53     (* sets the generator G05CAF to a repeatable initial state *)
54
55  FUNCTION RAND1 (var lower,upper :integer) :integer ;EXTERN  ;
56    (* modified G05DYF returns an integer from a uniform distribution *)
57
58  FUNCTION G05DBF (VAR mean : REAL SHORT) : REAL  ; EXTERN  ;
59    (* returns a real no. from a neg. expo. distribution *)
60
```

```
FUNCTION ChDe (p1,a1 : real) : real ;
(* The channel delay factor = normalized mean delay *)
    var
      X,Y,packet : real  ;
    begin
        packet := 0.1  ;
        X := 1+7.44*a1+p1*(1 +12.87*a1+53.37*a1*a1)/(2-2*p1-12.88*a1*p
\c1) ;
        Y := (5.44*a1-33.87*a1*a1*p1)/(1-p1*(1+6.44*a1)+p1*p1*(0.5+a1)
\c)  ;
        ChDe := (X-Y)*packet
    end  ;

FUNCTION Load (p2 : real) : real  ;
(* The effect of loading due to higher usage *)
   var
      wait : real  ;
   begin
      wait := 1.00  ;
      Load := wait/(1-p2)  ;  (* straight M/M/1,FCFS queue *)
   end  ;

PROCEDURE GENERATE ( var c : matrix) ;
  var
    lower,upper : integer  ;
    low,upp : integer  ;
    cost : integer  ;
    sinktime , sourcetime : integer  ;
    eff : real  ;
  begin
    for i:=1 to n do
      for j:=1 to n do
          if (i=j)  or (i=t) or (j=s)   or (i=s)  and  (j=t)
             then c[i,j] := 0  ;
          for  j:= (s+1)  to  (n-1)  do
                begin
                  G05CCF  ;
                  sourcetime := ROUND (G05DBF(meanone) + 1 )  ;
                  c[j,t] := sourcetime  ;
                  G05CCF  ;
                  mean := meantwo*Load(p2)  ;
                  sinktime := ROUND (G05DBF(mean)+1);(*The P1 strea
\cm*)
                  c[s,j] := sinktime  ;
                end  ;
      i := s+1  ;
            while  (i<=(n-2))  do
                begin
                    j := i + 1  ;
                    while (j<=(n-1))  do
                       begin
                          cost := ROUND (G05DBF(meanthree) + 1)  ;
                          c[i,j] := cost  ;
                          c[j,i] := cost  ;
                          j := j + 1
                       end  ;
                    i := i+ 1
                end  ;
   end  ;

 PROCEDURE MaxFlow (s,t:node ;c:matrix ; var f:matrix) ;
  var  RefNode :node ;  (*node with least excess capacity  *)
   MinPotential :real ;  (*excess capacity of the ref node *)
```

```
121     layer :vector ;  (*the layered network is defined by this array *)
122     r : real ;
123     Cut : cutset  ;
124     value : 1..maxint  ;
125     i,j :node ;  (*indices *)
126
127    FUNCTION  Min (x,y:real):real ;
128     (*determines the minimum amount of flow *)
129     begin
130      if x<y
131      then Min :=x
132       else Min :=y  ;
133     end ;
134
135    PROCEDURE Walk (i:node ) ;
136     (*traverse the layered network from t,inverting layer numbers.*)
137     var  j:node ;  li :xnode ;
138      begin
139       layer[i] := -layer[i] ;
140       li :=layer[i] ;
141       for j:= 1 to n do
142        if (j<>s) and (-layer[j]=li-1) and ((f[j,i]<c[j,i]) or (f[i,j]>0
\c))
143         then Walk (j)  ;
144      end ;  (*Walk *)
145
146    FUNCTION LayeringPossible : boolean ;
147      (*Is it possible to build a layered network, If so build it *)
148     var  i,j :node ;
149      k:0..n ;
150      EmptyLayer :boolean ;
151      begin
152       k :=0  ;     (*k keeps   track of layer being built *)
153       for i:= 1 to n do
154        layer[i] :=unscanned ;  (*initialize each node *)
155        layer[s] :=k ;  (*source node is in  layer 0 *)
156       repeat
157        k :=k+1 ;   (*now locate all nodes in layer k *)
158        EmptyLayer := true ;  (*an empty layer stops the algorithm *)
159        for i:= 1 to n do
160         if -layer[i] = k-1
161          then
162           (*i is in layer k-1,its neighbors may be in layer k  *)
163           for j := 1 to n  do  (*check each node adjacent to i *)
164            if (layer[j]=unscanned) and ((f[i,j]<c[i,j]) or (f[j,i]>0))
165             then
166              begin
167               layer[j] := -k ;
168               EmptyLayer := false
169              end ;
170       until  (layer[t] <> unscanned ) or EmptyLayer ;
171       LayeringPossible := not  EmptyLayer ;
172       Walk (t) ;   (*prune off the dead ends  *)
173      end ;  (*LayeringPossible *)
174
175    PROCEDURE FindRefNode  (i:node)  ;
176     (*traverse the layered network from t, seeking the ref node *)
177     var  j :node ;
178      li,lj :xnode ;
179      InCap, OutCap : real ;
180     begin
```

```
181       li  := layer[i] ;
182       InCap :=0  ;
183       OutCap := 0 ;
184       for j :=1 to n do
185        (*examine each node adjacent to i *)
186        begin
187         lj :=layer[j] ;
188         if (lj =li-1) and (j<>s) and ((f[j,i]<c[j,i]) or (f[i,j]>0))
189          then  FindRefNode (j)  ;
190         if  lj = li-1
191          then InCap :=InCap + (c[j,i]-f[j,i])+f[i,j]  ;
192         if lj =li+1
193          then OutCap :=OutCap + (c[i,j]-f[i,j])+f[j,i]
194        end ;
195       if  (i<>s) and (i<>t) and  (Min (InCap,OutCap)<MinPotential)
196       then
197        (*node i has smaller potential than the current ref node *)
198        begin
199         MinPotential := Min (InCap, OutCap)  ;
200         RefNode := i  ;
201        end ;
202     end  ;  (*FindRefNode *)
203
204    PROCEDURE  PushPull (i :node ; FlowLeft :real ; p : WhichWay ) ;
205     (*Augment the flow thro' i by pushing or pulling MinPotential units
\c *)
206     var  j, k1, k2, LayerSought : 0..n ;
207      begin
208      j := 0 ;
209       while (FlowLeft >0) and (j<n) do
210        begin
211         j :=j+1 ;
212         if p=push
213          then
214           begin
215            k1:=i ;
216            k2:=j ;
217            LayerSought:=layer[i]+1
218           end
219          else
220           begin
221            k1:=j ;
222           k2:=i ;
223            LayerSought :=layer[i]-1
224           end ;
225          r:=Min (FlowLeft,c[k1,k2]-f[k1,k2]+f[k2,k1] )  ;
226           (*amount of flow to move *)
227          if  (r>0) and (layer[j]=LayerSought)
228           then
229            begin  (*push/pull some flow to/from an adjacent  layer *)
230             FlowLeft :=FlowLeft -r  ;
231             f[k1,k2] :=f[k1,k2]+r-Min(r,f[k2,k1])  ;
232              (*Augment positive flow *)
233             f[k2,k1] := f[k2,k1]-Min (r,f[k2,k1])  ;
234              (*push reverse flow backwards *)
235             if (j<>s) and (j<>t)
236              then  PushPull (j,r,p)
237            end
238        end
239      end  ;(*PushPull *)
240
```

```
241    begin   (*maxflow *)
242     for i :=1 to n do
243      for j :=1 to n do
244       f[i,j] :=0  ;  (*initialy no flow *)
245       f[s,t] :=c[s,t]  ;  (*if an s_t link exists , saturate it *)
246       MinimumCut := 0 ;
247      while LayeringPossible do   (*assign nodes to layers  *)
248       begin
249        MinPotential := infinity  ;
250        FindRefNode (t)  ;   (*find the reference node *)
251        PushPull (RefNode,MinPotential,push) ;  (*push flow towards the
\csink*)
252        PushPull (RefNode,MinPotential,pull) ; (*pull flow from source*)
253        MinimumCut := MinimumCut + r ;
254       end  ;
255    end  ;  (*MaxFlow*)
256
257   begin   (*main program *)
258     writeln ('****************************************')  ;
259     writeln ('****MAXFLOW___MINCUT SCHEDULING ****')  ;
260     WRITELN ('****************************************')  ;
261     WRITELN  ;
262     writeln  ('a=0.005,T=20ys,b1=2ms,P=4000 bits,C=2Mbps')  ;
263     s :=1 ;
264     t := n ;
265     p1 := 0.00  ;
266     a1 := 0.005  ;
267     meanone := 150  ; (* ave source time *)
268     meantwo := 30  ; (* initial ave . sink time *)
269     meanthree := 10  ; (* ave intermod. time *)
270     (*for countzero := 1 to 23 do  *)
271         (*begin  *)
272           p2 := 0.00  ;
273           count := 1  ;
274           writeln  ;
275           writeln  ;
276           writeln ('---------------------------------')  ;
277           writeln ('  ethernet channel delay =',ChDe(p1,a1):7:3)  ;
278           writeln ('---------------------------------')  ;
279           writeln  ;
280           writeln  ;                                  degree of distributn
281           writeln ('count,avem,MinCut,avesource,avesink,dodsn,Load,link
\cs,ratio');
282           writeln ('----- ----- ----- --------- ------- ----- ----- ---
\c-- ----');
283           writeln  ;
284           writeln  ;
285           for countone := 1 to 50 do  (* 50 graph points *)
286               begin
287                 avem := 0  ;
288                 MinCut := 0.00  ;
289                 avesource := 0.00  ;
290                 avesink := 0.00  ;          = 50
291                 for counttwo := 1 to loopmax do
292                     begin  (* average out and increase loading at sink
\c*)
293                         GENERATE (c)  ;
294                         MaxFlow (s,t,c,f)  ;
295                         MinCut := MinCut + MinimumCut  ;
296                         for j := (s+1)  to  (n-1)  do
297                             begin
298                                 avesource := avesource + c[j,t]  ;
299                                 avesink := avesink + c[s,j]  ;
300                             end  ;
```

```
                        for  j := (s+1)  to  (n-1)  do
                             if (j<>t)  and  (f[s,j] = c[s,j])
                                  then   avem := avem + 1  ;
                    end  ; (countwo)
                avem := avem  div  loopmax ;(* no. scheduled to sink *)
                MinCut :=  MinCut/loopmax  ;(* minimum cut *)
                avesource := avesource/loopmax  ;(* ave. source time *)
                avesink := avesink/loopmax  ;(* ave. sink time *)
                avelinks := avem * (n-2-avem)  ;(* intermod links *)
                dodsn := avem/(n-2)  ;(* degree of distribution *)
                ratio := avesink/avesource  ;
                xx[count] := ratio  ;
                yy[count] := avem  ;
                write (count:4,avem:4,MinCut:4:2,avesource:9:2)  ;
                writeln (avesink:9:2,dodsn:4:2,Load(p2):7:3,avelinks:4,
\cratio:4:2)  ;
                count := count + 1  ;
                p2 := p2 + 0.0202  ;
            end  ; (countone)
            (*  p1 := p1 + 0.025  ;  *)
        (*  end  ;  *)
    nn := 50  ;
    mode := 0  ;
    for countone := 1 to 3 do
        begin
            C1051n  ;
            GRAF (xx,yy,nn,mode)  ;
            DEVEND  ;
        end  ;
    INT:=28 ;
    TIMDAT (B,INT) ;
      BEGIN
        WRITELN ;
        WRITELN ;
        WRITELN ('CPU SECONDS USED',B[7]) ;
        WRITELN ('CPU TICKS USED   ',B[8]) ;
        WRITELN ('DISK SECONDS USED',B[9]) ;
        WRITELN ('DISK TICKS USED   ',B[10]) ;
        WRITELN ('TICKS PER SECOND ',B[11]) ;
      END
end.
```

Appendix E System Performance

```
 1  program simulation (output)  ;
 2  $import
 3       'g05ccf(fortran)':g05ccf;
 4       'g05dbf(fortran)':g05dbf;
 5       'g05caf(fortran)':g05caf$
 6
 7  const
 8      nq = 4  ;
 9      nj = 100 ;
10      nn = 4  ;
11      nio = 2  ;
12      b1 = 5.0  ;
13      b2 = 0.05  ;
14      b3 = 0.22  ;
15      b4 = 0.019  ;
16
17  type
18     eventtype = (completion,nodedeparture)  ;
19     jobptr = ^jobelement  ;
20     eventptr = ^eventelement  ;
21     eventelement = record
22                          kindofevent : eventtype  ;
23                          time : real  ;
24                          job : jobptr  ;
25                          next : eventptr  ;
26                          previous : eventptr
27                    end  ;
28     jobelement = record
29                        currentnode : 1..nn  ;
30                        request : real  ;
31                        requestgranted : boolean  ;
32                        subserver : integer  ;
33                        nextjob : jobptr  ;
34                        tokenholder : jobptr  ;
35                        parent : jobptr  ;
36                        child : jobptr  ;
37                        event : eventptr
38                  end  ;
39     routingpointer = ^routingelement  ;
40     routingelement = record
41                           destination : 1..nn  ;
42                           probability : real  ;
43                           nextrouting : routingpointer
44                        end  ;
45     regenpointer = ^regenelement  ;
```

```
46     regenelement = record
47                          noderegen : 1..nn  ;
48                          lengthregen : integer  ;
49                          nextregen : regenpointer  ;
50                    end  ;
51
52   var
53     i : integer  ;
54     firstevent,lastevent,availevent : eventptr  ;
55     clock : real  ;
56     queues : array [1..nq] of
57                               record
58                                    discipline : (fcfs,lcfspr,ps)  ;
59                                    numberunits : integer  ;
60                                    numbersubservers : integer  ;
61                                    meansubservice : real  ;
62                                    firstinqueue : jobptr  ;
63                                    lastinqueue : jobptr  ;
64                                    length : integer  ;
65                                    timelengthchanged : real  ;
66                                    sumtimelength : real  ;
67                                    sumbusytime : real  ;
68                                    numbercompletions : integer  ;
69                                    bt : real  ;
70                                    tl : real  ;
71                                    nc : real  ;
72                                    btsq : real  ;
73                                    btxcl : real  ;
74                                    ncsq : real  ;
75                                    ncxcl : real  ;
76                                    tlsq : real  ;
77                                    tlxcl : real  ;
78                                    tlxnc : real
79                               end  ;
80
81    responsetime : real  ;
82     run,numberevents,eventlimit,eventmax : integer  ;
83     noeventsduringcycles,numbercycles,nocycm1 : integer  ;
84     timecyclestarted,cyclelength,sumcl,sumclsq,varcl,dcl : real  ;
85     util,dutil,varbt,covarbtcl : real  ;
86     tput,dtput,varnc,covarnccl : real  ;
87     ql,qt,dql,vartl,covartlcl,dqt,covartlnc : real  ;
88     availjob,tempjob : jobptr  ;
89     tempkind : eventtype  ;
90     firstregen,availregen : regenpointer  ;
```

```
      availrouting : routingpointer  ;
      nodes : array [1..nn] of
                              record
                                   kindofnode : (class,allocate,release,
                                                                  fission,fus
\cion)  ;
                                   queue : integer  ;
                                   lengthnode : integer  ;
                                   fusionptr : jobptr  ;
                                   routingptr : routingpointer  ;
                                   childrouting : routingpointer  ;
                              end  ;
      v1,v2 : integer  ;
       dummymeanvalue,negexpomean : real  ;

  procedure g05ccf ; external  ;

  function g05caf (var dummymeanvalue : real) : real ; external  ;

  function g05dbf (var negexpomean : real) : real ; external  ;


  function min (v1,v2 : integer) : integer  ;
     begin
         if v1<v2
            then min := v1
            else min := v2
     end  ;

  procedure insertevent (k : eventtype ; t : real ; j : jobptr)  ;
    (* insertevent adds event of kind k at time t for job j to list *)
     var
       temp, l : eventptr  ;
     begin
         if availevent = nil
            then new(temp)
            else
               begin  (* previously used storage available *)
                   temp := availevent  ;
                   availevent := availevent^.next
               end  ;
         temp^.kindofevent := k  ;
         temp^.time := t  ;
         temp^.job := j  ;
         j^.event := temp  ;
         if firstevent = nil
```

```
136             then
137                begin  (* list was empty *)
138                    firstevent := temp  ;
139                    lastevent := temp  ;
140                    temp^.next := nil  ;
141                    temp^.previous := nil  ;
142                end
143             else
144                if t<firstevent^.time
145                   then
146                      begin  (* insert at beginning of list *)
147                          temp^.next := firstevent  ;
148                          temp^.previous := nil  ;
149                          firstevent^.previous := temp  ;
150                          firstevent := temp  ;
151                      end
152                   else
153                      if t >= lastevent^.time
154                         then
155                           begin  (* insert at end of list *)
156                               lastevent^.next := temp  ;
157                               temp^.previous := lastevent  ;
158                               lastevent := temp  ;
159                               temp^.next := nil
160                           end
161                         else
162                           begin  (* insert  somewhere in middle of list
\c *)
163                               l := firstevent  ;
164                               while t >= l^.next^.time do
165                                     l := l^.next  ;
166                               temp^.next := l^.next  ;
167                               l^.next := temp  ;
168                               temp^.previous := l  ;
169                              .temp^.next^.previous := temp
170                           end  ;
171     end  ;  (* insertevent *)
172
173  procedure updatepsqueue (q : integer ; t : real ; j : jobptr)  ;
174     (* subtracts t from request for jobs currently in queue q. *)
175     (* then inserts j in the queue according to j^.request *)
176     var
177       temp : jobptr  ;
178     begin
179         with queues [q] do
180            begin
```

```
181                 temp := firstinqueue  ;
182                 while temp <> nil do
183                     begin
184                         temp^.request := temp^.request - t  ;
185                         temp := temp^.nextjob
186                     end  ;
187                 if j^.request < firstinqueue^.request
188                    then
189                       begin
190                          j^.nextjob := firstinqueue  ;
191                          firstinqueue := j  ;
192                       end
193                    else
194                       if j^.request >= lastinqueue^.request
195                          then
196                             begin
197                                 lastinqueue^.nextjob := j  ;
198                                 j^.nextjob := nil  ;
199                                 lastinqueue := j  ;
200                             end
201                          else
202                             begin
203                                 temp := firstinqueue  ;
204                                 while j^.request >= temp^.nextjob^.reque
\cst do
205                                     temp := temp^.nextjob  ;
206                                 j^.nextjob := temp^.nextjob  ;
207                                 temp^.nextjob := j  ;
208                             end
209             end
210     end  ;  (* updatepsqueue *)
211
212  procedure complete (var j : jobptr)  ;
213     (* handles completion of subserver for job j. *)
214     (* If service is complete, j remains unchanged *)
215     (* otherwise j becomes nil *)
216     var
217       leng : integer  ;
218       l : jobptr  ;
219       t : real  ;
220     begin
221         with queues [nodes[j^.currentnode].queue] do
222            begin
223                if j^.subserver < numbersubservers
224                then
225                   begin
```

```
     226                          j^.subserver := j^.subserver + 1  ;
     227                          if (discipline in [fcfs,lcfspr]) or (length =
\c1)
     228                             then
     229                                begin
     230                                    g05ccf  ;
     231                                    j^.request:=g05dbf(meansubservice)
\c;
     232                                    insertevent (completion,clock+j^.req
\cuest,j);
     233                                    j := nil  ;
     234                                end
     235                             else
     236                                begin  (* discipline = ps *)
     237                                    t := j^.request  ;
     238                                    g05ccf  ;
     239                                    j^.request:=g05dbf(meansubservice)
\c;
     240                                    firstinqueue := firstinqueue^.nextjo
\cb  ;
     241                                    updatepsqueue(nodes[j^.currentnode].
\cqueue,t,j);
     242                                    insertevent (completion,clock +
     243                                                 firstinqueue^.request *
     244                                                 length/min(length,numbe
\crunits)
     245                                                 , firstinqueue)  ;
     246                                    j := nil  ;
     247                                end
     248                       end
     249                    else
     250                       begin    (* statistics  *)
     251                           numbercompletions := numbercompletions + 1  ;
     252                           sumtimelength := sumtimelength +
\clength  ;
     253                                            (clock - timelengthchanged) *
     254
     255                           sumbusytime := sumbusytime +
     256                                          (clock - timelengthchanged)*
     257                                          min(length,numberunits)  ;
     258                           timelengthchanged := clock  ;
     259                           (*  mechanics *)
     260                           nodes[j^.currentnode].lengthnode :=
\c ;
     261                                    nodes[j^.currentnode].lengthnode - 1
     262                           length := length - 1  ;
     263                           if (discipline in [fcfs,lcfspr]) or (length =
\c0)
     264                              then
     265                                 begin
     266                                      if j = firstinqueue
     267                                         then
\ceue^.nextjob;
     268                                            begin
     269                                                firstinqueue := firstinqu
     270
                                                    if firstinqueue = nil
                                                       then lastinqueue := ni
\cl
                                                       else
```

```
     271                                                  begin
     272                                                      leng := 1  ;
     273                                                      l := firstinque
\cue  ;
     274                                                  end
     275                                              end
     276                                          else
     277                                              begin
     278                                                  l := firstinqueue  ;
     279                                                  leng := 2  ;
     280                                                  while j <> l^.nextjob do
     281                                                      begin
     282                                                          leng := leng + 1
\c ;
     283                                                          l := l^.nextjob
\c;
     284                                                      end  ;
     285                                                  if j^.nextjob = nil
     286                                                     then lastinqueue := l
\c ;
     287                                                  l^.nextjob := j^.nextjob
\c ;
     288                                                  l := l^.nextjob
     289                                              end  ;
     290                                      if length >= numberunits
     291                                         then
     292                                            begin
     293                                                while leng < numberunits do
     294                                                    begin
     295                                                        l := l^.nextjob  ;
     296                                                        leng := leng + 1  ;
     297                                                    end  ;
     298                                                if not l^.requestgranted
     299                                                   then
     300                                                      begin
     301                                                          g05ccf  ;
     302                                                          l^.request:=
     303                                                            g05dbf (meansubserv
\cice) ;
     304                                                          l^.requestgranted :=
\c true ;
     305                                                      end  ;
     306                                                insertevent(completion,clock+l^.re
\cquest,1);
     307                                            end
     308                                        end
     309                                    else
     310                                       begin  (*  discipline = ps *)
     311                                           t := j^.request  ;
     312                                           firstinqueue := firstinqueue^.nextjob  ;
     313                                           l := firstinqueue  ;
     314                                           while l <> nil do
     315                                               begin
```

```
316                                        l^.request := l^.request - t  ;
317                                        l := l^.nextjob  ;
318                                     end  ;
319                                  insertevent (completion,
320                                               clock + firstinqueue^.requ
\cest *
321                                               length/min(length,numberun
\cits),
322                                               firstinqueue)  ;
323                              end
324                        end
325               end
326        end  ;   (* complete *)
327
328     procedure removeevent (e : eventptr ; var k : eventtype ;
329                            var t : real ; var j : jobptr)  ;
330        (* removeevent returns kind k , time t and job j of event e *)
331        var
332          temp : eventptr  ;
333        begin
334            if firstevent = nil
335               then
336                  begin
337                      writeln ('Removevent --- empty list')  ;
338                      (* halt *)
339                  end
340               else
341                  if e = firstevent
342                     then
343                        begin
344                            k := firstevent^.kindofevent  ;
345                            t := firstevent^.time  ;
346                            j := firstevent^.job  ;
347                            temp := firstevent  ;
348                            firstevent := firstevent^.next  ;
349                            if firstevent = nil
350                               then  lastevent := nil
351                               else  firstevent^.previous := nil  ;
352                            temp^.next := availevent  ;
353                            availevent := temp  ;
354                        end
355                     else
356                        if e = lastevent
357                           then
358                              begin
359                                  k := lastevent^.kindofevent  ;
360                                  t := lastevent^.time  ;
```

```
     361                              j := lastevent^.job  ;
     362                              temp := lastevent  ;
     363                              lastevent := lastevent^.previous  ;
     364                              lastevent^.next := nil  ;
     365                              temp^.next := availevent  ;
     366                              availevent := temp  ;
     367                           end
     368                        else
     369                           begin
     370                              temp := firstevent  ;
     371                              while (temp <> e) and (temp <> nil) do
     372                                    temp := temp^.next  ;
     373                              if temp <> e
     374                                 then
     375                                    begin
     376                                      writeln('removeevent---event not
\cfound');
     377                                       (* halt  ; *)
     378                                    end
     379                                 else (* e is between first and lasteve
\cnt *)
     380                                    begin
     381                                       k := temp^.kindofevent  ;
     382                                       t := temp^.time  ;
     383                                       j := temp^.job  ;
     384                                       temp^.next^.previous := temp^.p
\crevious;
     385                                       temp^.previous^.next := temp^.n
\cext  ;
     386                                       temp^.next := availevent  ;
     387                                       availevent := temp  ;
     388                                    end
     389                           end
     390     end  ;  (* removeevent *)
     391
     392  procedure arrive (var j : jobptr ; c : integer)  ;
     393     (* handles arrival of a job j at class c. j becomes nil *)
     394     var
     395       dummykind : eventtype  ;
     396       t : real  ;
     397       dummyjob,temp : jobptr  ;
     398       leng : integer  ;
     399     begin
     400        j^.currentnode := c  ;
     401        j^.subserver := 1  ;
     402        j^.requestgranted := false  ;
     403        with queues[nodes[c].queue] do
     404           begin
     405              (* statistics *)
```

```
     406                sumtimelength := sumtimelength +
     407                                 (clock - timelengthchanged) * lengt
\ch  ;
     408                sumbusytime := sumbusytime +
     409                              (clock-timelengthchanged)*min(length,numbe
\crunits);
     410                timelengthchanged := clock  ;
     411                (* mechanics *)
     412                if (discipline = fcfs) or (firstinqueue = nil)
     413                   then
     414                      begin
     415                          j^.nextjob := nil  ;
     416                          if firstinqueue = nil
     417                             then firstinqueue := j
     418                             else lastinqueue^.nextjob := j  ;
     419                          lastinqueue := j  ;
     420                          nodes[c].lengthnode := nodes[c].lengthnode + 1
\c  ;
     421                          length := length + 1  ;
     422                          if length <= numberunits
     423                             then
     424                                begin
     425                                    g05ccf  ;
     426                                    j^.request := g05dbf (meansubservice
\c) ;
     427                                    j^.requestgranted := true  ;
     428                                    insertevent(completion,clock+j^.requ
\cest, j) ;
     429                                end
     430                      end
     431                   else
     432                      if discipline = lcfspr
     433                         then
     434                            begin
     435                                if length = numberunits
     436                                   then
     437                                      begin  (* preempt last in queue *)
     438                                          removeevent (lastinqueue^.even
\ct,
     439                                                       dummykind,t,dummy
\cjob)  ;
     440                                          lastinqueue^.request := t - cl
\cock  ;
     441                                      end
     442                                  else
     443                                      if length > numberunits
     444                                         then
     445                                           begin (* preempt last job in
\c service*)
     446                                                leng := 1  ;
     447                                                temp := firstinqueue  ;
     448                                                while leng < numberunits
\c do
     449                                                    begin
     450                                                        leng := leng + 1
\c ;
```

```
     451                                                    temp := temp^.ne
\cxtjob ;
     452                                                 end  ;
     453                                              removeevent (temp^.event
\c,
     454                                                      dummykind,t,d
\cummyjob);
     455                                              temp^.request := t - clo
\cck  ;
     456                                           end  ;
     457                                        j^.nextjob := firstinqueue  ;
     458                                        firstinqueue := j  ;
     459                                        nodes[c].lengthnode:=nodes[c].len
\cgthnode+1;
     460                                        length := length + 1  ;
     461                                        g05ccf  ;
     462                                        j^.request := g05dbf(meansubservi
\cce) ;
     463                                        j^.requestgranted := true  ;
     464                                        insertevent(completion,clock+j^.r
\cequest,j)
     465                          end
     466                       else   (* discipline = ps *)
     467                          begin
     468                              removeevent (firstinqueue^.event,dummykind,t
\c,
     469                                                                    dum
\cmyjob) ;
     470                              t := firstinqueue^.request - (t - clock) *
     471                                   min (length,numberunits)/length  ;
     472                              g05ccf  ;
     473                              j^.request:=g05dbf (meansubservice) ;
     474                              j^.requestgranted := true  ;
     475                              updatepsqueue (nodes[c].queue,t,j)  ;
     476                              nodes[c].lengthnode := nodes[c].lengthnode +
\c 1 ;
     477                              length := length + 1  ;
     478                              insertevent (completion,
     479                                           clock+firstinqueue^.request * l
\cength
     480                                           /min (length,numberunits), firs
\ctinqueue)
     481                          end
     482               end  ;
     483            j := nil
     484       end  ;  (* arrive *)
     485
     486  procedure addregen (n,l : integer)  ;
     487       (* initialises l jobs at node l. sets regeneration state descriptio
\cn to *)
     488       (* have l jobs at node l *)
     489       var
     490         temp : regenpointer  ;
     491         j : jobptr  ;
     492         i : integer  ;
     493       begin
     494           for i := 1 to l do
     495             begin
```

```
496              if availjob = nil
497                 then  new(j)
498                 else
499                    begin
500                        j := availjob  ;
501                        availjob := availjob^.nextjob  ;
502                    end  ;
503              j^.tokenholder := nil  ;
504              j^.parent := nil  ;
505              j^.child := nil  ;
506              arrive (j,n)  ;
507           end  ;
508        if availregen = nil
509           then  new(temp)
510           else
511              begin
512                  temp := availregen  ;
513                  availregen := availregen^.nextregen
514              end  ;
515        temp^.noderegen := n  ;
516        temp^.lengthregen := l  ;
517        temp^.nextregen := firstregen  ;
518        firstregen := temp
519     end  ;  (* addregen *)
520
521  function endcycle : boolean  ;
522    (* determines whether at at end of regeneration cycle. If so,endcycl
\ce
523       updates accumulators *)
524     var
525       result : boolean  ;
526       temp : jobptr  ;
527       l,q : integer  ;
528       rtemp : regenpointer  ;
529     begin
530        if firstevent = nil
531           then
532              begin
533                  writeln ('endcycle ---- event list empty')  ;
534                  endcycle := false  ;
535              end
536           else
537              begin
538                  if firstevent^.kindofevent = completion
539                     then  result := true
540                     else  result := false  ;
```

```
     541                    rtemp := firstregen  ;
     542                    while result and (rtemp <> nil) do
     543                        begin
     544                            if nodes[rtemp^.noderegen].lengthnode <>
     545                                        rtemp^.lengthregen
     546                               then result := false  ;
     547                            rtemp := rtemp^.nextregen
     548                        end  ;
     549                    if result
     550                       then
     551                          begin
     552                              q := 1  ;
     553                              while result and (q<=nq) do
     554                                  begin
     555                                      with queues[q] do
     556                                      if length > 0
     557                                         then
     558                                              if numbersubservers > 1
     559                                                 then
     560                                                    begin
     561                                                        if discipline=fcf
\cs
     562                                                           then
     563                                                             begin
     564                                                               temp:=firs
\ctinqueue;
     565                                                                 l := 1
\c;
     566                                                                 while re
\csult and
     567                                                                   (l<=mi
\cn(length
     568                                                                   ,numbe
\crunits))
     569                                                                do
     570                                                               begin
     571                                                                if temp^.s
\cubserver
     572                                                                     <> 1
     573                                                                     then
     574                                                                    resul
\ct:=false;
     575                                                                     l :=
\c l + 1 ;
     576                                                                     temp
\c :=
     577                                                                     temp
\c^.nextjob
     578                                                               end
     579                                                        end
     580                                                 else
     581                                                 begin
     582                                                 temp:=firstinque
\cue ;
     583                                                 l := 1 ;
     584                                                 while result and
     585                                                       (l<=length
\c)
```

```
     586                                                          do
     587                                                          begin
     588                                                          if temp^.sub
\cserver
     589                                                             <> 1
     590                                                              then
     591                                                             result :=
\c false;
     592                                                          l := l + 1
\c;
     593                                                          temp:=temp^.
\cnextjob
     594                                                          end
     595                                                    end
     596                                       end  ;
     597                                  q := q + 1
     598                              end
     599                        end  ;
     600                  if numberevents = 0
     601                     then
     602                        if not result and
     603                           (firstevent^.kindofevent=completion)
     604                           then
     605                              begin
     606                                 writeln('endcycle --- not initially
\cin
     607                                         regeneration state')  ;
     608                                 (* halt  ; *)
     609                              end
     610                           else
     611                              endcycle := false
     612                     else
     613                        if result
     614                           then
     615                              begin
     616                                 endcycle := true  ;
     617                                 noeventsduringcycles := numberevents
\c  ;
     618                                 numbercycles := numbercycles + 1  ;
     619                                 cyclelength := clock - timecyclestar
\cted ;
     620                                 timecyclestarted := clock  ;
     621                                 sumcl := sumcl + cyclelength  ;
     622                                 sumclsq := sumclsq + sqr (cyclelengt
\ch)  ;
     623                                 for q := 1 to nq do
     624                                    with queues[q] do
     625                                      begin
     626                                           sumtimelength :=sumtimelen
\cgth +
     627                                                  (clock-timelengthcha
\cnged)*
     628                                                  length  ;
     629                                          sumbusytime:=(sumbusytime
\c+
     630                                        (clock-timelengthchanged)*
```

```
     631                                                  min(length,numbersubservers))
\c/
     632                                                        numbersubservers  ;
     633                                                  timelengthchanged := clock
\c  ;
     634                                                  bt := bt + sumbusytime  ;
     635                                                  tl := tl + sumtimelength
\c;
     636                                                  nc := nc + numbercompletio
\cns ;
     637                                                  btsq := btsq + sqr (sumbus
\cytime)  ;
     638                                                  btxcl := btxcl+sumbusytime
\c*
     639                                                              cyclelength
\c  ;
     640                                                  sumbusytime := 0.0  ;
     641                                                  ncsq:=ncsq+sqr(numbercompl
\cetions) ;
     642                                                  ncxcl := ncxcl + numbercom
\cpletions*
     643                                                                cycleleng
\cth  ;
     644                                                  tlsq:=tlsq + sqr (sumtimel
\cength);
     645                                                  tlxcl:=tlxcl+sumtimelength
\c*
     646                                                                  cyclele
\cngth  ;
     647                                                  tlxnc := tlxnc + sumtimele
\cngth*
     648                                                              numbercompl
\cetions  ;
     649                                                  numbercompletions := 0  ;
     650                                                  sumtimelength := 0.0
     651                                              end
     652                                        end
     653                                   else
     654                                      endcycle := false  ;
     655                        end
     656      end  ;  (* endcycle *)
     657
     658
     659  function nextnode (j : jobptr) : integer  ;
     660     (* finds the next node for job j to go to *)
     661     var
     662       prob : real  ;
     663       temp : routingpointer  ;
     664     begin
     665        if (nodes[j^.currentnode].kindofnode = fission)
     666                              and
     667                        (j^.parent <> nil)
     668           then
     669              temp := nodes[j^.currentnode].childrouting
     670           else
     671              temp := nodes[j^.currentnode].routingptr  ;
     672        if temp = nil
     673           then
     674              begin
     675                  writeln ('nextnode---undefined routing from node',
```

```
676                             j^.currentnode)  ;
677                     (* halt  ; *)
678                 end  ;
679             if temp^.probability < 1.0
680                then
681                   begin
682                       g05ccf  ;
683                       prob := g05caf (dummymeanvalue)  ;
684                       while (prob > temp^.probability)
685                                       and
686                             (temp^.nextrouting <> nil ) do
687                             begin
688                                 prob := prob - temp^.probability  ;
689                                 temp := temp^.nextrouting  ;
690                             end
691                 end  ;
692             nextnode := temp^.destination
693         end  ;      (* nextnode *)
694
695     procedure adddestination (i,j : integer ; p : real ; c : boolean)  ;
696         (* adds destination node j to routing list for node i with probabil
    \city p*)
697         (* if c then routing is for child, otherwise parent *)
698         var
699           temp : routingpointer  ;
700         begin
701             if availrouting = nil
702                then
703                   new (temp)
704                else
705                   begin
706                       temp := availrouting  ;
707                       availrouting := availrouting^.nextrouting  ;
708                   end  ;
709             temp^.probability := p  ;
710             temp^.destination := j  ;
711             if c
712                then
713                   begin
714                       temp^.nextrouting := nodes[i].childrouting  ;
715                       nodes[i].childrouting := temp  ;
716                   end
717                else
718                   begin
719                       temp^.nextrouting := nodes[i].routingptr  ;
720                       nodes[i].routingptr := temp  ;
```

```
721                 end
722     end  ;      (* adddestination *)
723
724  begin        (* main program *)
725      (* initialization *)
726      availevent := nil  ;
727      availjob := nil  ;
728      availrouting := nil  ;
729      availregen := nil  ;
730      eventlimit:= 100  ;
731      for run := 1 to 3 do
732        begin
733            firstevent := nil  ;
734            lastevent := nil  ;
735            clock := 0.0  ;
736            numberevents := 0  ;
737            firstregen := nil  ;
738            numbercycles := 0  ;
739            timecyclestarted := 0.0  ;
740            sumcl := 0.0  ;
741            sumclsq := 0.0  ;
742            eventlimit := eventlimit * 10  ;
743           eventmax := 2*eventlimit  ;
744            for i := 1 to nq do
745                with queues[i] do
746                   begin
747                       discipline := fcfs ;
748                       numbersubservers := 1  ;
749                       numberunits := 1  ;
750                       firstinqueue := nil ;
751                       lastinqueue := nil  ;
752                       length := 0  ;
753                       timelengthchanged := 0.0  ;
754                       sumtimelength := 0.0  ;
755                       sumbusytime := 0.0  ;
756                       numbercompletions := 0  ;
757                       bt := 0.0  ;
758                       tl := 0.0  ;
759                       nc := 0.0  ;
760                       btsq := 0.0  ;
761                       btxcl := 0.0  ;
762                       ncsq := 0.0  ;
763                       ncxcl := 0.0  ;
764                       tlsq := 0.0  ;
765                       tlxcl := 0.0  ;
```

```
766                     tlxnc := 0.0  ;
767                     end  ;
768               for i := 1 to nn do
769                 with nodes[i] do
770                    begin
771                       kindofnode := class  ;
772                       queue := i  ;
773                       lengthnode := 0  ;
774                       routingptr := nil  ;
775                       fusionptr := nil  ;
776                       childrouting := nil  ;
777                    end  ;
778  (* parameters specific to this model *)
779               adddestination (1,2,1.0,false)  ;
780               queues[1].discipline := ps  ;
781               queues[1].numberunits := nj  ;
782               queues[1].meansubservice := b1  ;
783               adddestination (2,3,0.1,false)  ;
784               adddestination (2,4,0.9,false)  ;
785               queues[2].discipline := fcfs  ;
786               queues[2].meansubservice := b2  ;
787               adddestination (3,1,0.125,false)  ;
788               adddestination (3,2,0.875,false)  ;
789               queues[3].meansubservice := b3  ;
790               adddestination (4,1,0.125,false)  ;
791               adddestination (4,2,0.875,false)  ;
792               queues[4].meansubservice := b4  ;
793               addregen (1,nj)  ;
794                  (* run *)
795               while (firstevent <> nil) and (numberevents<eventmax)
796                                          and
797                     ((numberevents < eventlimit) or (not endcycle)) do
798                     begin
799                        removeevent (firstevent,tempkind,clock,tempjob
\c);
800                        if tempkind = completion
801                           then
802                              begin
803                                 numberevents := numberevents + 1  ;
804                                 complete (tempjob)  ;
805                              end  ;
806                        while tempjob <> nil  do
807                           begin
808                              i := nextnode (tempjob)  ;
809                                 arrive (tempjob,i)  ;
810                           end
```

```
     811                    end  ;
     812                    (*   print statistics  *)
     813                writeln  ;
     814                writeln ('number of events:',numberevents:8,
     815                         '     simulated time :',clock:10:3)  ;
     816                writeln  ;
     817                writeln ('queue util thruput queuelength queuetime')  ;
     818                if numbercycles > 1
     819                   then (* produce confidence interval estimates *)
     820                      begin
     821                          cyclelength := sumcl/numbercycles  ;
     822                          nocycm1 := numbercycles - 1  ;
     823                          varcl:=(sumclsq-sqr(sumcl)/numbercycles)/nocy
\ccm1;
     824                          responsetime := 0.0  ;
     825                          for i := 1 to nq do
     826                            with queues[i] do
     827                               if nc > 0
     828                                  then
     829                                     begin
     830                                         util := bt/sumcl  ;
     831                                         varbt:=(btsq-sqr(bt)/numbercyc
\cles)/
     832                                                                nocycm1
\c;
     833                                         covarbtcl:=(btxcl-bt*sumcl/
     834                                                    numbercycles)/nocy
\ccm1  ;
     835  dutil:=1.645*sqrt((varbt-2*util*
     836                                                 covarbtcl+sqr(util)*v
\carcl)/
     837                                                 numbercycles)/cyclele
\cngth  ;
     838                                         tput := nc/sumcl  ;
     839                                         varnc := (ncsq-sqr(nc)/numberc
\cycles)/
     840                                                              nocycm1
\c  ;
     841                                         covarnccl := (ncxcl-nc*sumcl/
     842                                                     numbercycles)/noc
\cycm1 ;
     843                                         dtput := 1.645*sqrt((varnc-2*t
\cput*
     844                                                 covarnccl+sqr(tput)*v
\carcl)/
     845                                                 numbercycles)/cyclele
\cngth ;
     846                                        ql := tl/sumcl  ;
     847                                        vartl := (tlsq-sqr(tl)/numbercy
\ccles)/
     848                                                              nocycm1
\c ;
     849                                        covartlcl:=(tlxcl-tl*sumcl/
     850                                                   numbercycles)/nocycm
\c1  ;
     851                                        dql := 1.645*sqrt((vartl-2*ql*
     852                                               covartlcl+sqr(ql)*varcl)
\c/
     853                                               numbercycles)/cyclelengt
\ch ;
     854                                        qt := tl/nc  ;
     855                                        covartlnc := (tlxnc-tl*nc/numbe
\crcycles)/
```

```
     856                                                          nocyc
\cm1  ;
     857                                    dqt := 1.645*sqrt((vartl-2*qt*c
\covartlnc
     858                                           +sqr(qt)*varnc)/numbercy
\ccles)/
     859                                           (nc/numbercycles)  ;
     860                                    writeln ('UPPER',util+dutil:12:
\c3,
     861                                             tput+dtput:11:3,
     862                                             ql+dql:13:3,
     863                                             qt+dqt:14:3)  ;
     864                                    writeln (i:5,util:12:3,tput:11:
\c3,
     865                                            ql:13:3,qt:14:3)  ;
     866                                    writeln ('LOWER',util - dutil:1
\c2:3,
     867                                             tput - dtput:11:3,
     868                                             ql - dql:13:3,
     869                                             qt - dqt:14:3)  ;
     870                                    if i=2
     871                                       then
     872                                          responsetime :=
     873                                                 responsetime+8*qt
     874                                           else
     875                                              if i=3
     876                                               then
     877                                               responsetime :=
     878                                       responsetime + 0.8*qt
     879                                               else
     880                                                  if i = 4
     881                                                   then
     882                                 responsetime := responsetime + 7.2
\c*qt ;
     883                                       end  ;
     884                           writeln  ;
     885                           writeln ('RESPONSE TIME =',responsetime:13:3) ;
     886                           writeln  ;
     887                           writeln ('NUMBER OF CYCLES:',numbercycles:8)  ;
     888                           if noeventsduringcycles <> numberevents
     889                              then
     890                                 writeln ('NUMBER OF DISCARDED EVENTS:',
     891                                          numberevents-noeventsduringcycles:
\c8) ;
     892                           writeln ('AVERAGE NUMBER OF EVENTS :',
     893                                    noeventsduringcycles/numbercycles:10:3)
\c ;
     894                           dcl := 1.645*sqrt(varcl/numbercycles)  ;
     895                           writeln ('AVERAGE LENGTH :',cyclelength:10:3,
     896                                    '   C.I. :(',cyclelength-dcl:10:3,',',
     897                                             cyclelength+dcl:10:3,')') ;
     898                     end
     899                  else    (* produce point estimates only *)
     900                     responsetime := 0.0  ;
```

```
     901                 for i := 1 to nq do
     902                   with queues[i] do
     903                      if numbercompletions + trunc (nc) > 0
     904                         then
     905                            begin
     906                                sumtimelength := sumtimelength + tl  ;
     907                                sumbusytime := sumbusytime + bt*numbersu
\cbservers;
     908                                numbercompletions:=numbercompletions+tru
\cnc(nc);
     909                                sumtimelength := sumtimelength +
     910                                                 (clock - timelengthchan
\cged)*
     911                                                                  length  ;
     912                                sumbusytime := sumbusytime +
     913                                            min(length,numbersubservers)
\c*
     914                                                  (clock-timelengthchang
\ced)  ;
     915                                writeln (i:5,
     916                                      sumbusytime/(numbersubservers*cloc
\ck):12:3,
     917                                         numbercompletions/clock:13:3,
     918                                         sumtimelength/clock:13:3,
     919                                         sumtimelength/numbercompletions
\c:14:3) ;
     920                                if i=2
     921                                   then
     922                                      responsetime :=responsetime+8*
     923                                       sumtimelength/numbercompletions
     924                                   else
     925                                      if i=3
     926                                       then
     927                                          responsetime := responsetime +
     928                                          0.8*sumtimelength/numbercomple
\ctions
     929                                       else
     930                                          if i=4
     931                                             then
     932                                                 responsetime := respons
\cetime+
     933                                               7.2*sumtimelength/numbercomp
\cletions
     934                            end  ;
     935                 writeln  ;
     936                 writeln ('responsetime = ',responsetime:13:3)  ;
     937                       (*   put leftovers on avail lists  *)
     938                 if firstevent <> nil
     939                    then
     940                       begin
     941                           lastevent^.next := availevent  ;
     942                           availevent := firstevent  ;
     943                       end  ;
     944           end
     945  end.
```

REFERENCES

1.1 S. Rosen : "Electronic Computers : A Historical Survey", Comp. Surv., Vol.1, No.1, March 1969, pp 7-36

1.2 J. Backus : "Can Programming Be L-berated From the Von Neumann Style : A Functional Snyle and Its Algebra of Programs", Comm. ACM 21, 1978, pp 613-641.

1.3 L. Uhr : "Algorithm-Structured Computer Arrays and Networks : Architectures and Processes for Images, Percepts, Models, Information", Academic Press, 1984

1.4 J.B. Dennis : "The Varieties of Data Flow Computers", First Int. Conf. Data Flow Comput., IEEE, 1979

1.5 C.G. Bell, J.C. Mudge and J.E. M-Namara : "Computer Engineering : A DEC View of Hardware Systems Design", Digital Press, Bedford, Mass., 1978

1.6 L.G. Valiant : "Universality Considerations in VLSI Circuits", IEEE Trans. Comput. 30, 1981, pp 135-140

1.7 D.P. Bhandarkar, J.E. Juliusen : "Semiconductor Technology : Trends and Implications", ACM Comp. Architecture News, Vol.7, No.1, 1978, pp 4-14

1.8 M. Shima : "Two Versions of 16-bit Chip Span Microprocessor, Mini-computer Needs", Electronics 21, pp 81-88, 1978

1.9 I.E. Sutherland, C.A. Mead, T.E. Everhart : "Basic Limitations in Microcircuit Fabrication Technology", Rep. R-1956-ARPA, RAND Corp.,1976

1.10 T. Uehara, and W.M. Van Cleemput : "Optimal Layout of CMOS Functional Arrays", IEEE Trans. Comput. 30, 1981, pp 305-312

1.11 H.Frank, I.T. Frisch and W. Chou : "Topological Considerations in the Design of the ARPA Computer Network", AFIPS Conf. Proc. 36, June 1970, pp 581-587

1.12 H. Frank and W. Chou : "Network Properties of the ARPA Computer Network", Networks, 4, John Wiley, 1974, pp 213-239

1.13 L. Kleinrock, W.E. Naylor and H. Opderbeck : "A Study of Line Overhead in the ARPA Network", Commun. ACM 19, No.1, Jan. 1976, pp 3-13

1.14 A.C. Yao : "The Entropic Limitations on VLSI Computation", Proc. 13th Ann. Symp. Theory Comput., 1981, pp 308-311

1.15 J.B. Dennis, G.A. Boughton, C.K. Leung : "Building Blocks for Data Flow Prototypes", Proc. 7th Ann. Symp. Comput. Archit., 1980, pp 1-8

1.16 J.B. Dennis, K.K. Weng : "Applications of Data Flow Computations to the Weather Problem", in "High Speed Computer and Algorithm Organisation", by D.J. Kuck, D.H. Lawrie and A.H.Sameh, 1977, pp 145-157, Academic Press, N.Y.

1.17 Eli T. Fathi and Moshe Krieger : "Multiple Microprocessor Systems : What, Why and When", Computer, 1983 March, pp 23-31

1.18 S.H. Fuller et al : "Multi-Microprocessors : An Overview and Working Examples", Proc. IEEE, Vol.6, No.2, Feb.1978, pp 216-218

1.19 P.H. Enslow, Jr., : "Multiprocessor Organisation - A Survey", Computing Surveys, Vol. 9, No.1, March 1977, pp 103-129

1.20 H.J. Siegel : "A Model of SIMD Machines and a Comparison of Various Interconnection Networks", IEEE Trans. Comput. 28, 1979, pp 907-917

1.21 M.J. Flynn : "Some Computer Organisations and Their Effectiveness", IEEE Trans. Comput. 21, 1972, pp 948-960

1.22 L. Uhr : "Parallel-Serial Production Systems with Many Working Memories", Proc. 5th Int. Joint Conf. Artificial Intelligence 1979c.

1.23 L.D. Wittie : "Efficient Message Routing in Mega-Micro-Computer Networks", Proc. 3rd, Ann.Symp. Comput. Archit. 1976,

1.24 L.D. Wittie : "MICRONET : A Reconfigurable Microcomputer Network for Distributed Systems Research", Simulation 31, pp 145-153, 1978

1.25 M. Hahn and P. Belanger : "Network Minimises Overhead of Small Computers", Electronics, August 1981

1.26 D.D. Clark and L. Svobodova : "Design of Distributed Systems Supporting Local Autonomy", Compcon, Spring 1980, pp 438-444

1.27 L.D. Wittie : "A Distributed Operating System for a Reconfigurable Network Computer", Proc. First Int. Conf. on Distrib. Comput. Syst., IEEE, 1979, pp 669-677

1.28 S.M. Abraham and Y.K. Dalal : "Techniques for Decentralized Management of Distributed Systems", Compcon., Spring 1980, pp 430-437

1.29 R. Smith : "The Contract Net Protocol : High Level Communication and Control in a Distributed Problem Solver", Proc. First Int. Conf. Distrib. Comput. Syst. IEEE, 1979, pp 185-192

1.30 H.S. Stone and S.H. Bokhari : "Control of Distributed Processes", Computer, Vol. II, July 1978, pp 97-106

1.31 G.M. Amdahl : "The Shructure of System/360, Part III : Processing Unit Design Considerations", IBM Syst. J., Vol.3, No.2, 1964, pp 144-164

1.32 G.M. Amdahl : "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", Proc. AFIPS, Spring, Joint Conf. 30, 1967, pp 40-54

1.33 H.S. Stone : "Multiprocessor Scheduling With the Aid of Network Flow Algorithms", IEEE Trans. Software Engineering, Vol. SE-3, No.2, Jan. 1977, pp 85-93

1.34 H.S. Stone : "Critical Load Factors in Two-Processor Distributed Systems", IEEE Trans. Software Engineering, Vol. SE-4, May 1978, pp254-25

2.1 M. Schwartz : "Computer-Communication Network Design and Analysis", Prentice-Hall, 1977.

2.2 D.W. Davies and D.L.A. Barber : "Communication Networks for Computers" John Wiley & Sons, 1973.

2.3 H. Frank, I.T. Frisch, and W. Chou : "Topological Considerations in the Design of the ARPA Computer Network", AFIPS Conf. Proc. 36, June 1970, pp. 581-587.

2.4 G.I. Chretien, W.M. Konig and J.H. Rech : "The SITA Network, Summary Description", Computer-Communication Networks Conference, University of Sussex, Sept. 1973.

2.5 D.W. Davies and D.L.A. Barber : "Communication Networks for Computers" John Wiley & Sons, 1973.

2.6 J.R. Harcharik : "TYMNET, Present and Future", IEEE Esscon Meeting, Washington, D.C., Sept.30, 1975.

2.7 D. Walden and A.A. McKenzie : "The Evolution of Host-to-Host Protocol Technology", Computer, Vol.12, pp.29-38, Sept. 1979.

2.8 C.A. Sunshine : "Interconnection of Computer Networks", Computer Networks, Vol.1, Jan.1977, pp.175-195.

2.9 A.K. Mok and S.A. Ward : "Distributed Broadcast Channel Access", Computer Networks, Vol.3, pp.327-335, Nov. 1979.

2.10 J.M. McQuillan and D.C. Walden : "The ARPA Network Design Decisions", Computer Networks, Vol.1, August 1977, pp.243-289.

2.11 E.D. Jensen, K.J. Thurber and G.M. Schneider : "A Review of Methods in Distributed Processor Interconnection" Proc. of the International Conference on Communications, 1976, pp.7-17-20

2.12 M. Gien and H. Zimmerman : "Design Principles of Network Interconnection" Proc. of 6th Data Commun. Symposium, pp.109-120, 1979.

2.13 M. Gerla and L. Kleinnock : "Topological Design of Distributed Computer Networks", IEEE Trans Commun., Vol. COMM-25, pp.48-60, Jan.1977b.

2.14 H. Frank and W. Chon : "Topological Optimization of Computer Networks", Proc. IEEE, Vol.60, Nov. 1972, pp.1385-1397.

2.15 B. Stuck : "Which Local Net Bus Access is Most Sensitive to Traffic Congestion?", Data Communications, Jan. 1983

2.16 E.C. Luczak : "Global Bus Computer Communication Techniques", Proc. Computer Network Symposium, 1978.

2.17 E. Cooper : "13 Often-Asked Questions About Broadband", Data Communications, April 1982.

2.18 E. Cooper and P.K. Edholm : "Design Issues in Broadband Local Networks", Data Communications, 1983.

2.19 M.A. Dineson and J.J. Picazo : "Broadband Technology Magnifies Local Network Capability", Data Communications, Feb. 1980.

2.20 M.A. Dineson : "Broadband Local Networks Enhance Communication Design", EDN. March 1981.

2.21 G.T. Hopkins and N.B. Meisner : "Choosing Between Broadband and Baseband Local Networks", Mini-Micro Systems, June 1982.

2.22 J.H. Salter and D.D. Clark : "Why a Ring?", Proc. Seventh Data Communications Symposium, ACM, 1982.

2.23 H. Salwen : "In Praise of Ring Architecture for Local Area Networks", Computer Design, March 1983.

2.24 R.C. Dixon : "Ring Network Topology for Local Data Communications", Proceedings, COMPCON, Fall 82, IEEE 1982.

2.25 P. Heywood : "The Cambridge Ring is Still Making the Rounds", Data Communications, July 1981.

2.26 J.R. Rush : "Microwave Links Add Flexibility to Local Networks", Electronics, Jan.13, 1982.

2.27 H. Frank, I.T. Frisch, and W. Chon : "Topological Considerations in the Design of the ARPA Computer Network", AFIPS Conf. Proc. 36, June 1970, pp.581-587.

2.28 R. Binder : "A Dynamic Packet Switching System for Satellite Broadcast Channels", Proc. International Conf. on Communications, IEEE.

2.29 A.M. Dahod : "Local Network Standards : No Utopia", Data Communications, March 1983.

2.30 J. Day : "Terminal, File Transfer, and Remote Job Protocols for Heterogenous Computer Networks", In "Protocols and Techniques for Data Communication Networks", by F.F. Kuo, Prentice-Hall, 1981.

2.31 H.C. Folt : "Coming of Age: A Long-Awaited Standard for Heterogenous Nets", Data Communications, Jan. 1981.

2.32 W. Myers : "Toward a Local Network Standard", IEEE Micro, Aug.1982.

2.33 E.E. Mier : "High-Level Protocols, Standards, and the OSI Reference Model", Data Communications, July 1982.

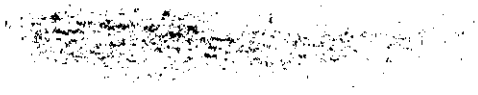
2.34 H. Zimmermann : "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", IEEE Trans. Commun. Vol. Com.28, pp.425-432, April 1980.

2.35 H.U. Bertine : "Physical Level Protocols", IEEE Trans. Communications, April 1980.

2.36 D.R. Doll : "Data Communications: Facilities, Networks, and System Design", N.Y., Wiley 1980.

2.37 N.J.A. Sloane : "A Short Course on Error Correcting Codes", Berlin: Springer Verlag 1975.

2.38 H.C. Folts : " Procedures for Circuit-Switched Service in Synchronous Public Data Networks", IEEE Trans. Commun., Vol. CDM-28, pp.489-496, April 1980a



2.39 S.W. Edge and A.J. Hinchley : "A Survey of End-to-End Retransmission Techniques", Computer Commun. Review, Vol.8, pp.1-18, Oct.1978.

2.40 A. Bochmann, and C. Sunshine : "Formal Methods in Communication Protocol Design", IEEE Trans. Commun., Vol.COM-28, pp.612-624, April 1980

2.41 E. Gelenbe, J. Labetoulle, and G. Pujolle : "Performance Evaluation of the HDLC Protocol", Computer Networks, Vol.2, pp.409-415, Sept. 1978.

2.42 L. Kleinrock, and M. Gerla : "Flow Control: A Comparative Survey", IEEE Trans. Commun., Vol. COM-28, pp.553-574, April 1980.

2.43 M. Schwartz and T.E. Stern : "Routing Techniques Used in Compute Communication Networks", IEEE Trans. Commun., Vol. COM-28, pp.539-552, April 1980.

2.44 J.M. McQuillan and D.C. Walden : "The ARPA Network Design Decisions" Computer Networks, Vol.1, pp.243-289, August 1977

2.45 I.M. Jacobs, R.Binder and E.V. Hoverstein : "General Purpose Packet Satellite Networks", Proc. IEEE, Vol.66, pp.1448-1467, Nov.1978.

2.46 R.E. Kahn : "The Organisation of Computer Resources into a Packet Radio Network", IEEE Trans. Commun., Vol. COM-25, pp.169-178, Jan.1977

2.47 D.D. Clark, K.T. Pogran and D.P. Reed : "An Introduction to Local Area Networks", Proc. IEEE, Vol.66, pp.1497-1517, Nov.1978.

2.48 K.J. Thurber, and H.A. Freeman : "Updated Bibliography on Local Computer Networks", Comput. Arch. News, Vol.8, pp.20-28, April 1980.

2.49 A. Grant, D. Hutchison and W. Shepherd : "A Gateway for Linking Local Area Networks and X-25 Networks", Proc. SIGCOMM 83 Symposium, 1983.

2.50 E. Eschenauer and V. Obozinski : "The Network Communication Manager: A Transport Station for the SGB Network", Computer Networks, Vol.2, pp.236-249, Sept. 1978.

2.51 J.F. Shoch and L. Stewart : "Interconnecting Local Networks via the Packet Radio Network", Proc. Sixth Data Commun. Symp., pp.153-158, 1979.

2.52 W.B. Rauch-Hindin : "Upper-Level Network Protocols", Electron Design, March 3, 1983.

2.53 J. Day : "Terminal Protocols", IEEE Trans. Commun. Vol.COM-28, pp.585-593, April 1980.

2.54 W. Diffie and M.E. Hellman : "Privacy and Authentication", Proc. IEEE, Vol. 67, pp.397-427, March 1979.

2.55 M. Gien : "A File Transfer Protocol (FTP)", Computer Networks, Vol.2, pp.312-319, Sept. 1978.

2.56 A. Lempel : "Cryptology in Transition", Computer Survey, Vol.11, pp.286-303, Dec. 1979.

2.57 M. Gasser, and D.P. Sidhu : "A Multilevel Secure Local Area Network", Proc. IEEE Symposium on Security and Privacy, 1982.

2.58 G.J. Popek and C.S. Kline : "Encryption and Secure Computer Networks", Computer Survey, Vol.11, pp.331-356, Dec.1979.

2.59 W.W. Chu and P.P. Chen : "Centralized and Distributed Data Base Systems", IEEE 1979, Calif.

2.60 J.E. Donnelley : "Components of a Network Operating System", Computer Networks, Vol.3, pp.389-399, Dec. 1979.

2.61 J.B. Dennis : "The Varieties of Data Flow Computers", First Int. Conf. Data Flow Comput., IEEE, pp.430-431, 1979.

2.62 W. Bux : "Local-Area Subnetworks: A Performance Comparison", IEEE Trans. Communications, 1981.

2.63 G.S. Christensen : "Links Between Computer-Room Networks", Telecommunications, Feb. 1979.

2.64 D.D. Clark, K.T. Pogran, and D.P. Reed : "An Introduction to Local Area Networks", Proceedings of the IEEE, Nov.1978.

2.65 I.W. Cotton : "Technologies for Local Area Computer Networks", Proceedings, Local Area Communications Network Symposium, 1979.

2.66 J.F. Hayes : "Local Distribution in Computer Communications", IEEE Communications Magazine, March 1981.

2.67 M. Killen : "The Microcomputer Connection to Local Networks", Data Communications, March 1982.

2.68 R.M. Metcalfe and D.R. Boggs : "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, Vol.19, pp.395-404, July 1976.

2.69 J.F. Shoch, Y_K. Dala and D.D. Redell : "Evolution of the Ethernet Local Computer Network", Computer, August 1982.

2.70 M. Marathe, and B. Hawe : "Predicted Capacity of Ethernet in a University Environment", Proceedings, SOUTHCOM 82, 1982.

2.71 W. Bux : "Local-Area Subnetworks: A Performance Comparison", IEEE Trans. Communications, 1981.

2.72 I.W. Cotton : "Technologies for Local Area Computer Networks", Proceedings, Local Area Communications Network Symposium, 1979.

2.73 G.M. Pfister and B.V. O'Brien : "Comparing the CBX to the Local Network - and the Winner Is?", Data Communications, July 1982.

2.74 W.W. Chu, W. Haller and K.K. Leung : "A Contention Based Channel Reservation Protocol for High Speed Local Networks", Proceedings, Seventh Conf. on Local Computer Networks, 1982.

2.75 H.C. Folts : "X.25 Transaction-Oriented Features - Datagram and Fast Select", IEEE Trans. Commun. Vol. COM-28, pp.496-500, April 1980b.

2.76 H.C. Folts : "Status Report on New Standards for DTE/DCE Interface Protocols", Computer, Vol.12, pp.12-19, Sept. 1979.

2.77 H.C. Fleming, and R.M. Hutchison, Jr. : "Low-Speed Data Transmission on the Switched Telecommunication Network", Bell Syst. Tech. J., pp.1385-1406, April 1971.

2.78 J.M. Kasson : "Survey of Digital PBX Design", IEEE Trans. Communications, July 1979.

2.79 J.M. Kasson : "The Rolm Computerized Branch Exchange: An Advanced Digital PBX", Computer, June 1979.

2.80 Institution of Electrical and Electronic Engineers : "IEEE Project 802, Local Network Standards, Draft C", May 17, 1982.

2.81 J.F. Shoch, and J.A. Hupp : "Measured Performance of an Ethernet Local Network", Communications of the ACM, Dec. 1980.

2.82 E.G. Rawson and R.M. Metcalfe : "Fibernet: Multimode Optical Fibres for Local Computer Networks", IEEE Trans. Communications, July 1978.

2.83 J.R. Jones : "Consider Fibre Optics for Local Network Designs", EDN, March 3, 1983.

2.84 A. Sheltzer, R. Hinden and M. Brescia : "Connecting Different Types of Networks with Gateways".

2.85 M.T. Liu, W. Hilal and B.H. Groomes : "Performance Evaluation of Channel Access Protocols for Local Computer Networks", Proceedings of COMPCON 82 Fall, 1982.

2.86 A.E. Joel : "Circuit Switching: Unique Architecture and Applications", Computer, June, 1979.

2.87 H.C. Folts : "Procedures for Circuit-Switched Service in Synchronous Public Data Networks", IEEE Trans. Commun., Vol. COM-28, pp.489-496, April 1980a.

2.88 W.D. Farmer and E.E. Newhall : "An Experimental Distributed Switching System to Handle Bursty Computer Traffic", Proceedings, ACM Symposium on Problems in the Optimization of Data Communications, 1969.

2.89 W. Rauch-Hindin : "IBM's Local Network Scheme", Data Communications, May 1982.

2.90 J.W. Mark : "Global Scheduling Approach to Conflict-Free Multiaccess via a Data Bus", IEEE Trans. on Communications, Sept. 1978.

2.91 H.H. Driver, H.L. Hopewell, J.F. Inquinto : "How the Gateway Regulates Information Flow", Data Communications, Sept. 1979.

2.92 D.P. Heyman : "An Analysis of Carrier-Sense Multiple-Access Protocol", Bell Syst. Techn. J., October 1982.

2.93 M. Steiglitz : "Local Network Access Tradeoffs", Computer Design, Oct. 1981.

2.94 N. Abramson : "The ALOHA System - Another Alternative for Computer Communications", Proceedings, Fall Joint Computer Conference, 1970.

2.95 L.G. Roberts : "ALOHA Packet System With and Without Slots and Capture", Computer Communications Review, April 1975.

2.96 L. Kleinrock and S.S. Lam : "Packet Switching in a Multiaccess Broadcast Channel: Performance Evaluation", IEEE Trans. Commun. Vol. COM-23, pp.410-423, April 1975.

2.97 S.S. Lam and L. Kleinrock : "Packet Switching in a Multiaccess Broadcast Channel: Dynamic Control Procedures", IEEE Trans. Commun., COM-23, pp.891-904, Sept. 1975.

2.98 L. Kleinrock and F.A. Tobagi : "Packet Switching in Radio Channels: Part I: Carrier-Sense Multiple-Access Modes and Their Throughput - Delay Characteristics", IEEE Trans. Commun., Dec. 1975.

2.99 L.Kleinrock and F.A. Tobagi : "Random Access Techniques for Data Transmission over Packet-Switched Radio Channels", Proceedings National Computer Conference, AFIPS Press, pp.187-201, 1975.

2.100 F.A. Tobagi : "Multiaccess Protocols in Packet Communication Systems", IEEE Trans. Commun. Vol. COM-28, pp.468-488, April 1980c.

2.101 D.C. Wood, S.F. Holmgren, and A.P. Skelton : "A Cable-Bus Protocol Architecture", Proceedings, Sixth Data Communications Symposium, 1979.

2.102 A.K. Mok and S.A. Ward : "Distributed Broadcast Channel Access", Computer Networks, Vol.3, pp.327-335, Nov.1979.

2.103 L.W. Hansen and M.Schwartz : "An Assigned-Slot Listen-Before-Transmission Protocol for a Multiaccess Data Channel", IEEE Trans. Commun. Vol. COM-27, pp.846 - 857, June 1979.

2.104 S.S. Lam : "A Carrier Sense Multiple Access Protocol For Local Networks", Computer Networks, Vol.4, pp.21-32, Feb.1980.

2.105 F.A. Tobagi and V.B. Hunt : "Performance Analysis of Carrier Sense Multiple Access with Collision Detection", Computer Networks, Nov. 1980.

2.106 F.A. Tobagi : "Distributions of Packet Delay and Interdeparture Time in Slotted ALOHA and Carrier Sbnse Multiple Access", Journal of the ACM, Oct. 1982.

2.107 A.B. Carleial and M.E. Hellman : "Bistable Behaviour of ALOHA-Type Systems", IEEE Trans. Commun. Vol. COM-23, pp.401-410, April 1975.

2.108 J.I. Capetanakis : "Generalized TDMA: The Multi-Accessing Tree Protocol", IEEE Trans. Commun., Vol. COM-27, pp.1476-1484, Oct.1979.

2.109 D.E.Carlson : "Bit-Oriented Data Link Control Procedures", IEEE Trans. Commun. April 1980.

2.110 A.E. Joel : "Digital Switching - How It Has Developed", IEEE Trans. Commun. 1979.

2.111 E. Gelenbe, J.Labetoulle, and G.Pujolle : "Performance Evaluation of the HDLC Protocol", Computer Networks, Vol.2, pp.409-415, Sept. 1978.

2.112 I. Chlamtac and W.R. Franta : "Message-Based Priority Access to Local Networks", Computer Communications, April 1980.

2.113 V.G. Cerf and P.T. Kristein : "Issues in Packet-Network Interconnection", Proceedings of the IEEE, Nov. 1978.

2.114 C.K. Miller and D.M. Thompson : "Making a Case for Token Passing in Local Networks", Data Communications, March 1982.

2.115 L. Pouzin : "Virtual Circuits vs Datagrams - Technical and Political Problems", Proceedings of the National Computer Conference, pp.483-494, 1976.

2.116 M. Graube : "Local Area Nets: A Pair of Standards", IEEE Spectrum, June 1982.

3.1 J.von Newmann : "The Computer and the Brain", Yale University Press, 1959.

3.2 N. Wirth: "Systematic Programming : An Introduction", Prentice Hall, 1973

3.3 C.A.R. Hoare, O.J.Dahl and E.W. Dijkstra : "Structured Programming", Academic Press, 1972

3.4 W.F. Dalton : "Design Microcomputer Software like other Systems - Systematically", Electronics, Jan.19 1978, pp 97-101

3.5 M.J.Flyn : "Directions and Issues in Architecture and Language", Vol.13, pp 5-22, Oct.1980

3.6 A.S. Tanenbaum : "Implications of Structured Programming for Machine Architecture", Commun. ACM, Vol.21, pp 237-246, March 1978

3.7 J.R. Heath and S.M. Patel : "How to write a Universal Cross-Assembler", IEEE Micro, Vol.1, pp 45-66, August 1981

3.8 D.E. Knuth : "The Art of Computer Programming, Vol.1 : Fundamental Algorithms", Addison-Wesley, 1967

3.9 D.E. Knuth : " The Art of Computer Programming, Vol.2 : Seminumerical Algorithms", Addison-Wesley, 1969

3.10 E.W. Dijkstra : " GOTO Statement Considered Harmful", Commun. ACM, Vol.11, pp 147-148, March 1968

3.11 N.Wirth and H. Weber : " EULER : A Generalization of ALGOL, and Its Formal Definition, Part I", Comm. ACM, Vol.9, No.1, Jan.1966, pp 13-25

3.12 N.Wirth and H.Weber : "EULER : A Generalization of ALGOL, and Its Formal Definition, Part II", Comm.ACM, Vol.9, No.2, Feb.1966, pp 89-99

3.13 G. Morris : "Make Your Next Instrument Design Emphasize User Needs and Wants", EDN, October 20, pp 100-105, 1978

3.14 D.L. Parnas : "On the Criteria to be Used in Decomposing Systems into Modules", Commun. ACM, pp 1053-1058, Dec.1972

3.15 D.L. Parnas : "A Technique for the Specification of Software Modules with Examples", Commun. ACM, pp 330-376, May 1973

3.16 R.W. Ulrickson : "Software Modules are the Building Blocks", Electronic Design, pp 62-66, Feb.1977

3.17 R.W. Ulrickson : "Solve Software Problems Step-by-Step", Electronic Design, pp 54-58, Jan.18, 1977

3.18 A.V. Aho, J.E. Hopcroft, J.D.Ullman : "Data Structures and Algorithms", Addison-Wesley, 1983

3.19 D.E. Knuth : "An Empirical Study of FORTRAN Programs", Software-Practice and Experience, pp 105-133, 1971

3.20 N.Wirth : "Program Development by Stepwise Refinement", Commun.ACM, 14, pp 221-227, 1971

3.21 N. Wirth : "Algorithms + Data Structures = Programs", Prentice-Hall, 1975

3.22 A.V.Aho, J.E.Hopcroft and J.D. Ullman : "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974

3.23 H.S. Stone : "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", IEEE Trans. Software Eng., Vol.SE-3, No.2, pp 85-93, Jan.1977

3.24 B. Carrè : "Graphs and Networks", Oxford, 1979

3.25 F.Harary : "Graph Theory", Addison-Wesley, 1969

3.26 H.S.Stone and S.H. Bokhari : "Control of Distributed Processes", Computer, Vol.II, pp 97-106, July 1978

3.27 S.H.Bokhari : "Dual Processor Scheduling with Dynamic Reassignments", IEEE Trans. Software Eng., Vol.SE-5, pp 341-349, July 1979

3.28 L.R. Ford and D.R. Fulkerson : "Flows in Networks", Princeton University Press, 1962

3.29 H.Frank and I.Frisch : "Communication, Transmission, and Transportation Networks", Addison-Wesley, 1971

3.30 V.M. Malhotra, M.P. Kumar and S.W. Maheshwari : "An O ($|V|^3$) Algorithm for Finding Maximum Flows in Networks", Inf.Proc.Lett., Vol.7, pp 277-278, Oct. 1978

3.31 A.S. Tanenbaum : "Computer Networks", Prentice-Hall, 1981

3.32 E.L. Lawler : "Cutsets and Partitions of Hypergraphs", Networks 3, pp 275-285, July 1973

3.33 M.B. Wells : "Elements of Combinatorial Computing", Pergamon, Oxford, 1971

3.34 J.Edmonds and R.M. Karp : "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems", J.Ass.Comput.Mech., Vol.19, pp 248-264, 1972

3.35 H.Bokham : "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System", IEEE Trans. Software Eng., Vol. SE-7, pp 587-589, Nov.1981

3.36 A.C. Yao : "An O ($|E|$ log log $|V|$) Algorithm for Finding Minimum Spanning Trees", Inf. Processing Letters, Vol.4, pp 21-23, 1975

3.37 J. Edmonds : "Paths, Trees and Flowers", Canadian J. Math., Vol.17, pp 449-467, 1965

3.38 A.Van Dam, G.Stabler and R. Harrington : "Intelligent Satellites for Interactive Graphics", Proc. IEEE, Vol.62, pp 83-92, April 1974

3.39 J.Turner : "The Structure of Modular Programs", Commun. ACM, Vol.23, pp 272-277, May 1980

3.40 A.V. Aho and J.D. Ullman : "Principles of Compiler Design", Addison-Wesley, 1978

REFERENCES

4.1 J. Nievergelt and J.C. Farrar : "What machines can and cannot do", ACM Computing Surveys, Vol.4, pp 81-96, 1972

4.2 A.V. Aho, J.E. Hopcroft and J.D. Ullman : "The Design and Analysis of Computer Algorithms", Addison-Wesley, 1974

4.3 W.S. Brainerd and L.H. Landweber : "Theory of Computation", John Wiley, 1974

4.4 A.M. Turing : "On Computable numbers, with an application to the Entscheidungsproblem", Proc. London Math. Soc., Vol.42, pp 230-265

4.5 E.L. Post : "Finite Combinatory Processes", J.Symbolic Logic, Vol.1, pp 103-105

4.6 A. Church : "The Calculi of Lambda-conversion", Ann. Math., Vol.6

4.7 S.C. Kleene : "General Recursive Functions of Natural Numbers", Math. Ann., Vol.12, pp 340-353

4.8 E.L. Post : "Formal reductions of the general combinatorial decision problems", Am.J. Math., Vol.65, pp 197-268

4.9 C.E. Elgot and A. Robinson : "Random Access Stored Program Machines : An Approach to Programming Languages", J. ACM, Vol.11, 1964, pp 365-399

4.10 M. Minsky : "Computation, Finite and Infinite Machines", Prentice-Hall, 1967

4.11 A. Gill : "Introduction to the Theory of Finite-state Machines", McGraw-Hill, 1962

4.12 M. Arbib : "Theories of Abstract Automata", Prentice-Hall, 1969

4.13 E.E. Swartzlander, B.K. Gilbert and I.S. Reed : "Inner Product Computers", IEEE Trans. Comput., Vol. C-27, Jan.1978, pp 21-31

4.14 E.E. Swartzlander and B.K. Gilbert : "Arithmetic for Ultra-High-Speed Tomography", IEEE Trans. Comput., Vol. C-29, May 1980, pp 341-353

4.15 O.L. MacSorley : "High Speed Arithmetic in Binary Computers", Proc. IRE, Vol.49, 1961, pp 67-91

4.16 D.J. Kuck : "The Structure of Computers and Computation, Vol.1", John Wiley, 1978

4.17 A. Lunde : "Empirical Evaluation of Some Features of Instruction Set Processor Architectures", Comm. ACM, Vol. 20, No.3, 1977, pp 143-153

4.18 J.C. Mudge : "Design Decisions Achieve Price/Performance Balance in Mid-range Minicomputers", Computer Design, Vol. 16, No.3, August 1977, pp 87-95

4.19 H.C. Lucas : "Performance Evaluation and Monitoring", Comp. Surv., Vol.3, No.3, 1971, pp 79-91

4.20 J.C. Gibson : "The Gibson Mix", IBM Systems Development, N.Y., 1970

4.21 E. Raichelson and G. Collins : "A Method for Comparing the Internal Operating Speeds of Computers", Comm. ACM, Vol.7, No.5, May 1964, pp 309-310.

4.22 N. Bonwell : "Benchmarking : Computer Evaluation and Measurement", John Wiley, 1975

4.23 R.M. Keller : "Look Ahead Processors", Comp. Surv., Vol.7, No.4, Dec. 1975, pp 177-195

4.24 S.A. Cook and R.A. Reckhow : "Time-Bounded Random Access Machines", J.Computer and System Sciences, Vol.7, No.4, pp 354-375, 1973

4.25 M.O. Rabin : "Complexity of Computations", Comm. ACM, Vol.20, No.9, 1977, pp 625-633

4.26 A.V. Aho, J.E. Hopcroft, J.D. Ullman : "Time and Tape Complexity of Pushdown Automaton Languages", Information and Control, Vol.13, 1968, pp 186-206

4.27 J.Hartmanis, P.M.Lewis II and R.E. Stearns : "Classification of Computations by Time and Memory Requirements", Proc. IFIP Congress 65, Spartan, N.Y., pp 31-35, 1965

4.28 R.L. Graham : "Bounds on Multiprocessing Timing Anomalies", SIAM Journal of Applied Math, Vol.17, No.2, pp 416-429, 1969

4.29 M.R. Garey, R.L. Graham and J.D. Ullman : "Worst-case Analysis of Memory Allocation Algorithms", Proc. of the 4th Annual ACM Symposium on Theory of Computing, pp 143-150, 1972

4.30 D.E. Knuth : "Big Omicron and Big Omega and Big Theta", SIGACT News, No.2, pp 18-24, 1976

4.31 D.S. Johnson : "Approximation Algorithm for Combinatorial Problems", Proceedings of the 5th Annual ACM Symposium on Theory of Computing, pp 38-49, 1973

4.32 E. Horowitz and S. Sahni : "Computing Partitions with Applications to the Knapsack Problem", J. ACM, Vol.21, No.2, pp 277-292, 1974

4.33 R.V. Book : "Comparing Complexity Classes", J.Computer and System Sciences, 1974

4.34 M. Davies : "Computability and Unsolvability", McGraw-Hill, 1958

4.35 S.L. Hantler and J.C. King : "An Introduction to Proving the Correctness of Programs", ACM Computing Surveys, Vol.8, No.3, pp 331-353,1976

4.36 D.E. Knuth, "The Art of Computer Programming, Vol. 1 : Fundamental Algorithms", Addison-Wesley, 1968

4.37 M.H. Halstead : "Elements of Software Science", North Holland, 1977

4.38 E.I. Jury : "Theory and Application of the Z-Transform Method", John Wiley, 1964

4.39 T.C. Wesselkamper : "Computer Program Schemata and the Processes They Generate", IEEE Trans. Soft. Eng. Vol.SE-8, No.4, July 1982, pp 412-419

4.40 J.L. Peterson : "Petri Net Theory and the Modelling of Systems", Prentice-Hall, 1981

4.41 D.E. Knuth : "Structured Programming with GOTO Statements", ACM, Computing Surveys, Vol.6, pp 261-301, 1974

4.42 E.A. Ashcroft and Z. Manna : "The Translation of GOTO programs into WHILE programs, in Proceedings International Federation for Information Processing Congress, North-Holland, Amsterdam, pp 250-255, 1971

4.43 C. Bohm and Y. Jocopini : "Flow Diagrams, Turing Machines and Languages with Ohly Two Formation Rules", Comm. ACM, Vol.19, pp 366-371, 1966

4.44 T.G. Price : "A Note on the Effect of the Central Processor Service Time Distribution on Processor Utilization in Multiprogrammed Computer Systems", J.ACM, Vol.23, No.2, pp 342-346, April 1976

4.45 E.D. Lazowska : "The Use of Percentiles in Modelling CPU Service Time Distributions", in Computer Performance, K.M. Chandy and M.Reiser, North-Holland, 1977, pp 53-66

4.46 R.M. Brown, J.C. Browne and K.M. Chandy : "Memory Mangement and Response Time", Comm. ACM, Vol.20, No.3, pp 153-165, March 1975

4.47 W.Chiu, D.Dumont and R.Wood : "Performance Analysis of a Multi-programmed Computer System", IBM J. of Res. and Dev., Vol.19, No.3, pp 263-271, May 1975

4.48 D.P. Gaver : "Probability Models of Multiprogramming Computer Systems", J.ACM, Vol.14, No.3, pp 423-428, 1967

4.49 D.F.Towsley, J.C.Browne and K.M.Chandy, "Models for Parallel Processing Within Programs : Application to CPU-I/O and I/O-I/O Overlap", Comm.ACM, Vol.21, No.10, pp 821-831, Oct.1978

4.50 T.J.Teorey and T.B. Pinkerton : "A Comparative Analysis of Disk Scheduling Policies", Comm. ACM, Vol.15, No.3, pp 177-183, Dec.1975

4.51 W.Faller : "An Introduction to Probability Theory and Its Applications", 3rd Ed., John Wiley, 1968

4.52 W.J. Stewart : "A Comparison of Numerical Techniques in Markov Modelling", Comm.ACM, Vol.21, pp 144-151, 1978

4.53 V.L. Wallace and R.S. Rosenberg : "Markovian Models and Numerical Analysis of Computer System Behaviour", AFIPS Conf. Proc. 28, pp 141-148, 1966

4.54 L. Kleinrock : "Queueing Systems Volume I : Theory", John Wiley, 1975

5.1 N. Abramson : "The ALOHA System - Another Alternative for Computer Communications", Proc., Fall Joint Computer Conf., 1970

5.2 L.G. Roberts : "ALOHA Packet System with and without slots and capture", Computer Communications Review, April 1975

5.3 L. Kleinrock and S.S. Lam : "Packet Switching in a Multiaccess Broadcast Channel : Performance Evaluation", IEEE Trans. Commun., Vol. COM-23, pp 410-423, April 1975

5.4 L.W. Hansen and M. Schwartz : "An assigned-slot Listen-Before-Transmission Protocol for a Multiaccess Data Channel", IEEE Trans. Commun., Vol. COM-27, pp 846-857, June 1979

5.5 F.A. Tobagi : "Multiaccess Protocols in Packet Communication Systems", IEEE Trans. Commun., Vol. COM-28, pp468-488, April 1980c.

5.6 L.Kleinrock and F.A. Tobagi : "Packet Switching in Radio Channels; Part I: Carrier-sense Multiple-Access Modes and Their Throughput-Delay Characteristics", IEEE Trans. Commun., Dec.1975

5.7 L.Kleinrock and F.A. Tobagi : "Random Access Techniques for Data Transmission over Packet-Switched Radio Channels", Proc. National Computer Conference, AFIPS Press, pp 187-201, 1975

5.8 F.A. Tobagi and V.B. Hunt : "Performance Analysis of Carrier Sense Multiple Access with Collision Detection", Computer Networks, Nov. 1980.

5.9 S.S. Lam : "A Carrier Sense Multiple Access Protocol for Local Networks", Computer Networks, Vol.4, pp 21-32, Feb.1980

5.10 W. Feller : "An Introduction to Probability Theory and Its Applications", 3rd Edition, Vol.1, Wiley, 1968

5.11 A. Papoulis : "Probability, Random Variables and Stochastic Processes", McGraw-Hill, 1965

5.12 L. Kleinrock : "Queueing Systems, Vol. II : Computer Applications", Wiley, 1976

5.13 B. Stuck : "Which Local Bus Access is Most Sensitive to Traffic Congestion", Data Communications, Jan.1983

5.14 L. Kleinrock : "Communication Nets.", New York, Dover, 1964

5.15 A.B. Carleial and M.E. Hellman : "Bistable Behaviour of ALOHA-Type Systems", IEEE Trans. Commun., Vol. COM-23, pp 401-410, April 1975

5.16 R.M. Metcalfe and D.R. Boggs : "Ethernet: Distributed Packet Switching for Local Computer Networks", Comm-n. ACM, Vol.19, pp 395-404, July 1976

5.17 L. Kleinrock : "Queueing Systems, Vol. 1: Theory," John Wiley, 1975

5.18 D.V. Widder : "The Laplace Transform", Princeton University Press, Princeton, 1946

5.19 G. Doetsch : "Guide to the Applications of Laplace Transforms," Van Nostrand, 1961

5.20 E. I. Jury : "Theory and Applications of the Z-Transform Method" John Wiley, 1964

5.21 J.D.C. Little : "A proof of the Queueing Formula L = λW", Operations Research, Vol.9, pp 383-387, 1961

5.22 J.F. Shoch and J.A. Hupp : "Measured Performance of an Ethernet Local Network", Comm. ACM, Dec.1980

REFERENCES

6.1 J.M. McKinney : "A survey of Analytical Time-Sharing Models", Computing Surveys, Vol. 1, pp 105-116, 1969

6.2 L. Kleinrock and E.G. Coffman : "Some Feedback Queueing Models for Time-shared Systems", Proc. of the 5th International Teletraffic Congress, pp 91-92, June 1967

6.3 E.G. Coffman and L. Kleinrock : "Feedback Queueing Models for Time-Shared Systems", JACM, Vol.15, pp 549-576, 1968

6.4 M. Greenberger : "The Priority Problem and Computer Time-sharing", Management Science, Vol.12, pp 888-906, 1966

6.5 W. Chang : "Single-server Queueing Processes in Computing Systems", IBM Syst. J., pp 36-71, 1970

6.6 E. Fuchs, and P.E. Jackson : "Estimates of Distributions of Random Variables for Certain Computer Communications Traffic Models", Comm. ACM, Vol.13, pp 752-757, 1970

6.7 I. Adivi and B. Avi-Itzhak : "A Time-sharing Queue with a Finite Number of Customers", J. ACM, Vol.16, pp 315-323, 1969

6.8 I. Adivi : "Computer Time Sharing Queues with Priorities", J. ACM, Vol.16, pp 631-645, 1969.

6.9 W.J. Gordon and G.F. Newell : "Closed Queueing Systems with Exponential Servers", Operations Research, Vol.15, pp 254-265, 1967

6.10 L. Kleinrock and G. Coffman : "Distribution o- Attained Service in Time-shared Systems", J. of Computer System Science, Vol.1, pp 287-298, 1967

6.11 L. Kleinrock and R.R. Muntz : "Processor-sharing Queueing Models of Mixed Scheduling Disciplines for Time-shared Systems", J. ACM, Vol.19, pp 464-482, 1972

6.12 R.R.P. Jackson : "Queueing Systems with Phase Type Service", Operations Research, Vol.5, pp 109-120, 1954

6.13 S.H. Fuller and F. Baskett : "An Analysis of Drum Storage Units", JACM, Vol.22, pp 83-105, 1975

6.14 T.J. Teorey and T.B. Pinkerton : "A Comparative Analysis of Disk Scheduling Policies", Comm. ACM, Vol 15, pp 177-183, 1972

6.15 F. Baskett, K.M. Chandy, R.R. Muntz and F. Palacios : "Open Closed, and Mixed Networks of Queues with Different Classes of Customers", J ACM, Vol.22, pp 248-260, 1975

6.16 J.R. Jackson : "Networks of Waiting Lines", Operations Research Vol.5, pp 518-521, 1957

6.17 J.R. Jackson : "Jobshop - Like Queueing Systems", Management Science, Vol.10, pp 131-142, 1963

6.18 C.H. Sauer and K.M. Chandy : "Approximate Analysis of Central Server Models", IBM J. of Research and Development, Vol.19, pp 301-313, 1975

6.19 W.M. Chow : "The Cycle Time Distribution of Exponential Central Server Models", AFIPS Conf. Proc. 43, 1978

6.20 D.F. Towsley : "Queueing Network Models with State Dependent Routing", JACM, Vol 27, pp 323-337, 1980

6.21 K.M. Chandy, J.H. Howard and D.F. Towsley : "Product Form and Local Balance in Queueing Networks", JACM, Vol 24, pp 250-263, 1977

6.22 K.M. Chandy : "The Analysis and Solutions for General Queueing Networks", Proc. 6th Annual Princeton Conf. on Ihformation Science and Systems, pp 224-228, 1972

6.23 K.M. Chandy and C.H. Sauer : "Computational Algorithms for Product Form Queueing Networks", Comm. ACM, Vol.10, 1980

6.24 J.P. Buzen : "Computational Algorithms for Closed Queueing Networks with Exponential Servers", Comm. ACM, Vol.16, pp 327-531, 1973

6.25 A.A. Scheer : "An Analysis of Time-shared Computer Systems", MIT Press, 1967

6.26 R.R. Muntz and J. Wong : "Asymptotic Properties of Closed Queueing Network Models", Proc. of 8th Ann. Conf. on Information Sciences and Systems, Princeton University, 1974

6.27 F.R. Moore : "Computational Model of a Closed Queueing Network with Exponential Servers", IBM J. of Research and Development, pp 567-572, 1972

6.28 M. Posner and B. Bernholtz : "Closed Finite Queueing Networks with Time Lags and with Several Classes of Units", Operations Research, Vol.16, pp 977-985, 1968

6.29 J.D.C. Little : "A Proof of the Queueing Formula L = λW", Operations Research, Vol.9, pp 383-387, 1961

6.30 L. Kleinrock : "Queueing Systems, Vol. 2 : Computer Applications", Wiley, 1976

6.31 H. Kebayashi : "Application of the Diffusion Approximation to Queueing Networks I: Equilibrium Queue Distributions", J. ACM, pp 316-328, 1974

6.32 D.P. Gaver and G.S. Shedler : "Processor Utilization in Multi-programming Systems via Diffusion Approximations", Operations Research, Vol.21, pp 569-576, 1973

6.33 M. Reiser and C.H. Sauer : "Queueing Network Models: Methods of Solution and Their Program Implementation", in "Current Trends in Programming Methodology, Vol.3 : Software Modelling and Its Impact on Performance", by K.M. Chandy and R.T. Yeh, Prentice-Hall, 1978

6.34 M. Reiser and S.S. Lavenberg : "Mean Value Analysis of Closed Multi-chain Queueing Networks", J. ACM, Vol.27, pp 313-322, 1980

6.35 G.S. Fishman : "Principles of Discrete Event Simulation", Wiley, 1978

6.36 D.E. Knuth : "The Art of Computer Programming, Volume 2 : Semi-numerical Algorithms", Addison-Wesley, 1969

6.37 M.A. Crane and D.L. Iglehart : "Simulating Stable Stochastic Systems II : Markov Chains", J. ACM, Vol.21, pp 114-123, 1974

6.38 G.S. Fishman and L.R. Moore : "Estimating the Mean of a Correlated Binary Sequence", J. ACM, Vol.26, pp 82-94, 1979

6.39 M.A. Crane and A.J. Lemoine : "An Introduction to the Regenerative Method for Simulation Analysis", Springer-Verlag, 1977

6.40 W. Feller : "An Introduction to Probability Theory and Its Implications", Wiley, 1968

6.41 D.L. Iglehart and G.S. Shedler : "Regenerative Simulation of Response Times in Network of Queues", J ACM, Vol 25, pp 449-460, 1978

6.42 S.S. Lavenberg and D.R. Slutz : "Introduction to Regenerative Simulation", IBM J.of Research and Development, Vol.19, pp 458-463,1975

6.43 C.H. Sauer : "Confidence Intervals for Queueing Simulations of Computer Systems", Performance Evaluation Review, Vol.8, pp 45-55,1979

6.44 P. Heidelberger and P.D. Welch : "A Spectral Method for Confidence Interval Generation and Run Length Control in Simulations", Comm. ACM, Vol.24, pp 233-245, 1981

6.45 K. Jensen and N. Wirth : "PASCAL User Manual and Report", Springer-Verlag, 1974

6.46 C.H. Sauer and K.M. Chandy : "Computer Systems Performance Modelling", Prentice-Hall, 1981

6.47 Computer Centre Documentation, University of Loughborough

6.48 W.R. Franta and K. Maly : "An Efficient Data Structure for the Simulation Event Set", Comm. ACM, Vol.20, pp 596-602, 1977

6.49 F.A. Haight : "Queueing with Balking, II", Biometrika, Vol.47, pp 285-296, 1960

6.50 C.J. Ancker and A.V. Garfarian : "Some Queueing Problems with Balking and Renezing, II", Operations Research, Vol.II, pp 88-100, 1963.