LOAD SHARING IN DISTRIBUTED COMPUTER SYSTEMS

TAHIR M. AFZAL

PhD Thesis

The University of Newcastle upon Tyne

Computing Laboratory

August, 1987

ABSTRACT

In this thesis the problem of load sharing in distributed computer systems is investigated. Fundamental issues that need to be resolved in order to implement a load sharing scheme in a distributed system are identified and possible solutions suggested. A load sharing scheme has been designed and implemented on an existing Unix United system. The performance of this load sharing scheme is then measured for different types of programs. It is demonstrated that a load sharing scheme can be implemented on the Unix United systems using the existing mechanisms provided by the Newcastle Connection, and without making any significant changes to the existing software. It is concluded that under some circumstances a substantial improvement in the system performance can be obtained by the load sharing scheme.

ACKNOWLEDGEMENTS

I would like to thank Dr. Snow for his constructive advice in supervising this thesis. I am also very grateful to Professor Brian Randell for his invaluable comments on preliminary drafts of this thesis.

Several colleagues in the Computing Laboratory helped me during my research. In particular I am thankful to Robert Stroud for his help in solving the problems related to the Newcastle Connection. I wish to thank Dr. Lindsay Marshall, Andy Linton and Steve Grimes for their technical help. I am grateful to Professor Peter Lee for his comments on the final draft of this thesis.

The support of Science and Engineering Research Council during the first two years of my research is gratefully acknowledged.

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

Throughout the history of computers, efforts have been made to make them faster to meet the ever increasing demands of new applications. In the past these efforts had largely been concentrated on increasing the speeds of physical components such as processors and memories. Then the decreasing costs of processors made it possible to develop multiprocessor machines with the objective of providing fast computers.

The availability of cheap processors and memories also resulted in relatively cheap computers. Therefore, more organisations could afford several independent computers to meet their computational demands. Later, the desire to share the resources existing on different computers within an organisation, coupled with the advances in the communications technology, led to the development of what are now known as Distributed Systems.

In most distributed systems a user on one computer can normally access data and programs residing on other computers in the system. It is less common, however, to find distributed systems where some computational load of a heavily used computer is automatically and surreptitiously transferred to a less loaded computer, thus performing load sharing in the system. In this thesis we shall address this problem of load sharing in distributed computer systems.

The term **Distributed** has been used to describe a variety of computer systems. The differences in the nature, location, and the interconnection of the physical components of such systems make it virtually impossible to define distributed systems according to their physical attributes. Therefore, in section

1.1 we shall consider the fundamental objectives of distributed systems, and present a logical definition that encompasses the features necessary to achieve these objectives.

Although difficult to define, several classification schemes have been proposed for existing distributed systems. In order to understand which of these classes we shall consider for load sharing, in section 1.2 we present a classification scheme that is useful in the context of this thesis.

In section 1.3 we first define, informally, the terms **Multiprocessor System**, **Distributed System**, and **Load Sharing** as they will be used in this thesis. We shall then discuss the role of load sharing in multiprocessor and distributed systems. Finally, in section 1.4 the aims and the structure of the thesis are presented.

1.1 OBJECTIVES AND FEATURES OF DISTRIBUTED SYSTEMS

The fundamental objectives of a distributed system have been identified by Lelann [LELANN 81] as being: Increased Performance, Extensibility, Availability, and Resource Sharing. Several logical definitions of distributed systems have been suggested [LELANN 81, ENSLOW 78, JENSEN 78, BLAIR 83, TRIPATHI 80]. These definitions identify the features that help achieve the objectives of distributed systems. Enslow's definition [ENSLOW 78] is possibly the most widely accepted. Therefore, we discuss below five features that are identified by Enslow as being necessary for a system to be **Fully Distributed.** For each feature we discuss the effect it has on the four objectives of distributed systems listed above.

I - Multiplicity of Resources:

A distributed system consists of a multiplicity of general-purpose physical and logical resource components. The term **general-purpose** is important since it excludes the systems where the components are bound to specific tasks. Thus a mainframe with several I/O processors and special arithmetic units is not regarded as constituting a distributed system since these components each perform just one specific function.

Besides being general purpose, the constituent components in a distributed system should be dynamically assignable, on a short-term basis, to various system tasks. For the purpose of multiplicity, the components may be replicated, and it is not necessary for them to be homogeneous.

Jensen [JENSEN 78] has implied this requirement in **Multiplicity of Processors**, and Blair [BLAIR 83] requires it in the second property of his definition. Multiplicity of resources is related to all four objectives of distributed systems. Increased performance can be obtained if a service is provided by more than one component in the system because there will be less waiting for that service. The objective of extensibility is likely to be satisfied because if the system has been designed with the knowledge that various resource components can provide a particular service then one would expect it to be an easy task to add other resources that provide the same service. Availability is increased because a failure of one component providing a particular service need not result in the failure of the whole system since it would be possible to provide the failed service by another component, thus sharing a resource.

II- Component Interconnection:

The physical distribution of resource components in a distributed system can range from a length of connecting track on an integrated chip to the distance between two computers linked together through an international network. The important point about the interconnection in a distributed system is that the communication between the components is established by utilising two-party cooperative protocols. This means that each component is free to make its own decisions about receiving and replying to a message.

Systems whose components communicate in a master-slave manner (where the master has the full authority to force a slave to perform some task) are not regarded as distributed since they preclude the autonomous operation (criterion V) of the components. Tripathi's [TRIPATHI 80] and Lelann's [LELANN 81] definitions make an explicit requirement that the communication be based on message passing, presumably to avoid the master-slave relationship between the components.

III- Unity of Control:

In a distributed system individual processors are allowed to have their own operating systems controlling their resources. These operating systems may or may not be identical. In order to achieve the objectives of a distributed system, a high level operating system (also known as an 'executive control') is required that implements a well-defined set of policies that govern the distributed system as a whole. However, there must be no strong hierarchy existing between the high-level operating system and the local operating systems since this would violate the criterion for autonomous operation (criterion V).

What policies this high level operating system should implement depends on the nature of the system. As an example, for a general purpose system supporting interactive users, a common command language interpreter would be required. This will encourage the users to use the resources on remote machines, thus achieving the objectives of resource sharing and high performance.

IV- System Transparency:

An important characteristic of a distributed system is the degree to which its distributedness can be transparent to the user. Enslow requires that the users should be able to request services by their generic names and need not be concerned about which physical or logical component is to provide that service.

The issue of transparency must be considered for the following three different types of distributed systems:

a) Distributed Systems where various services are provided by different machines; these services, when combined, form one operating system;

b) Distributed Systems where different machines are running the same operating system;

c) Distributed Systems where different machines are running different operating systems;

Transparency required by Enslow's criterion involves two issues: transparency in location of components providing the service and, secondly, transparency in method of access.

In the first type of distributed systems a service can be provided by more than one component and, therefore transparency of location of components is

required. The issue of transparency of method of access does not arise here since there is only one operating system involved.

The second type of distributed system can be regarded as another form of the first type in which all or most of the services are replicated. In this type therefore transparency of location of component is required.

In the third type of distributed systems the transparency of both the location and the method of access is required. The user of such a system need only specify what service is required. The local operating system can then determine whether a local or a remote component is required. If the service of a remote operating system is required then the operating system should be able to access it. This level of transparency can be provided by the high level or executive operating system mentioned in the previous sub-section. The high level operating system effectively changes the third type of distributed systems into second type.

System transparency is an essential feature because it helps achieve the objectives of increased performance, availability and resource sharing by making it easier for the users to access other computers in the system. Blair [BLAIR 83] and Jensen [JENSEN 78] have also included it in their definitions.

v - Cooperative Autonomy:

Both the logical and physical components of a distributed system should interact in a manner described as **cooperative autonomy**. This means that the components operate in an autonomous fashion requiring cooperation among processes for the exchange of information as well as for the provision of services. Any component is able to refuse requests for service, even after it has accepted the message requesting that service. This could result in anarchy except for the

fact that all components adhere to a common set of system utilisation and management policies expressed by the high-level operating system. Cooperative autonomy is essential for the objectives of availability, and extensibility.

The degree to which a system incorporates the above features determines the extent to which the objectives of distributed systems are achieved.

1.2 CLASSIFICATION OF DISTRIBUTED SYSTEMS

Existing distributed systems incorporate the features discussed in the last section to a varying degree. Therefore, in order to understand exactly what type of systems we shall be considering later for load sharing, a classification scheme for distributed systems is presented in this section. This scheme has been proposed by Keefe [KEEFE 85] and is largely based on the way the individual computers in a distributed system interact with each other.

I - Network of Autonomous Systems with Explicit Network

The main characteristic of this division is that no attempt is made to provide system transparency. In order that machines of widely different architectures and different operating systems may communicate, there must be protocols for all projected activities. An example of such a system is provided by the Bell Laboratory's dial-up network of Unix† systems called UUCP [NOWITZ 80]. In this system the user needs to give different commands when dealing with remote machines; for example when copying a file from a remote machine the *uucp* command is used instead of the standard Unix command *cp*.

---

† Unix is a registered trademark of AT &T Bell Laborotories.

II- A Network of Autonomous Systems with the Network Hidden

Systems in this class are similar to those described above, except that the presence of a network is hidden from the user. This is usually achieved by introducing a layer of software between the user and the network. The user still has to know the location of a resource within his name space, but the interface to access remote services is same as that for local services. Examples of this type of system are Unix United [BROWNBRIDGE 82] (further described in chapter 4), and Cocanet [ROWE 82].

III- Integrated Loosely Coupled Systems with Autonomous Nodes

This category of distributed systems is characterised by the presentation of a uniform name space to all users, but where individual nodes of the system may function alone, albeit with only a subset of that name space available to them. Any given resource will have the same name, irrespective of its location, and the source of the request. Distributed systems that can be placed in this class are PULSE [KEEFE 85] and LOCUS [POPEK 83].

IV- Integrated Loosely Coupled Systems with Non-autonomous Nodes

Systems in this class are similar to those in the previous one, except that the nodes are not capable of operating alone. A machine removed from the network will no longer be operable, but the remainder will continue to function. Examples of this class of system are the New Mexico State University Ring-Star system [KARSHMER 83], and the Cambridge distributed computing system [NEEDHAM 80].

v - Tightly Coupled Distributed Systems

This category applies to systems consisting of several processors sharing memory on common bus. Such systems are more commonly called multiprocessor systems; the distinction we choose to make between multiprocessor and distributed systems is described in section 1.3 below. A good example of this class of systems is provided by the StarOS system [JONES 79] that was implemented at Carnegie-Mellon University for the fifty processor Cm* multiprocessor computer.

The categories mentioned above do not constitute an exhaustive list, and other systems can be classified according to their filing arrangements, homogeneity of components or the power of the processors involved. However, by considering the classes mentioned above, it is clear that different classes achieve the objectives of distributed systems to a varying degree. The systems in class 1, for example, achieve to some extent the objectives of resource sharing and extensibility. The systems in classes 2 and 3 come closest to achieving all the objectives. Although the systems in classes 4 and 5 may achieve the objectives of resource sharing and increased performance, due to the dependence of components on each other the objectives of availability and extensibility are not fully met.

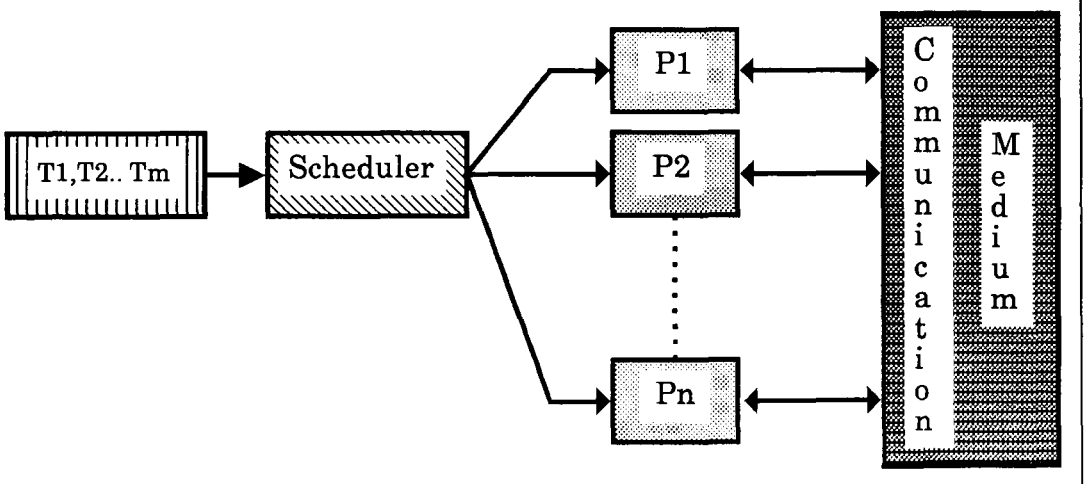
1.3 LOAD SHARING MULTIPROCESSOR AND DISTRIBUTED SYSTEMS

This thesis is primarily concerned with load sharing in distributed computer systems. However, since most early reported work on load sharing was done for multiprocessor systems, and some of the concepts employed could be applicable to distributed systems, we shall also briefly discuss load sharing in

multiprocessor systems. Therefore in this section we shall informally define the terms Multiprocessor System, Distributed System, and Load Sharing as they will be used in this thesis.

Multiprocessor System

In this thesis a Multiprocessor System will be assumed to have the generic form shown in Figure 1.1 below. A multiprocessor system consists of more than one processor (P1, P2 ..... Pn). All the tasks (T1, T2 ....Tm) waiting to be executed are allocated to the processors by **one** scheduler. Communications between the processors are established through the communication medium, which is likely to be, but not necessarily, shared memory. This definition of a multiprocessor system would cover most systems in group five of Keefe's classification scheme.



MULTIPROCESSOR SYSTEM

FIGURE 1.1

Distributed System

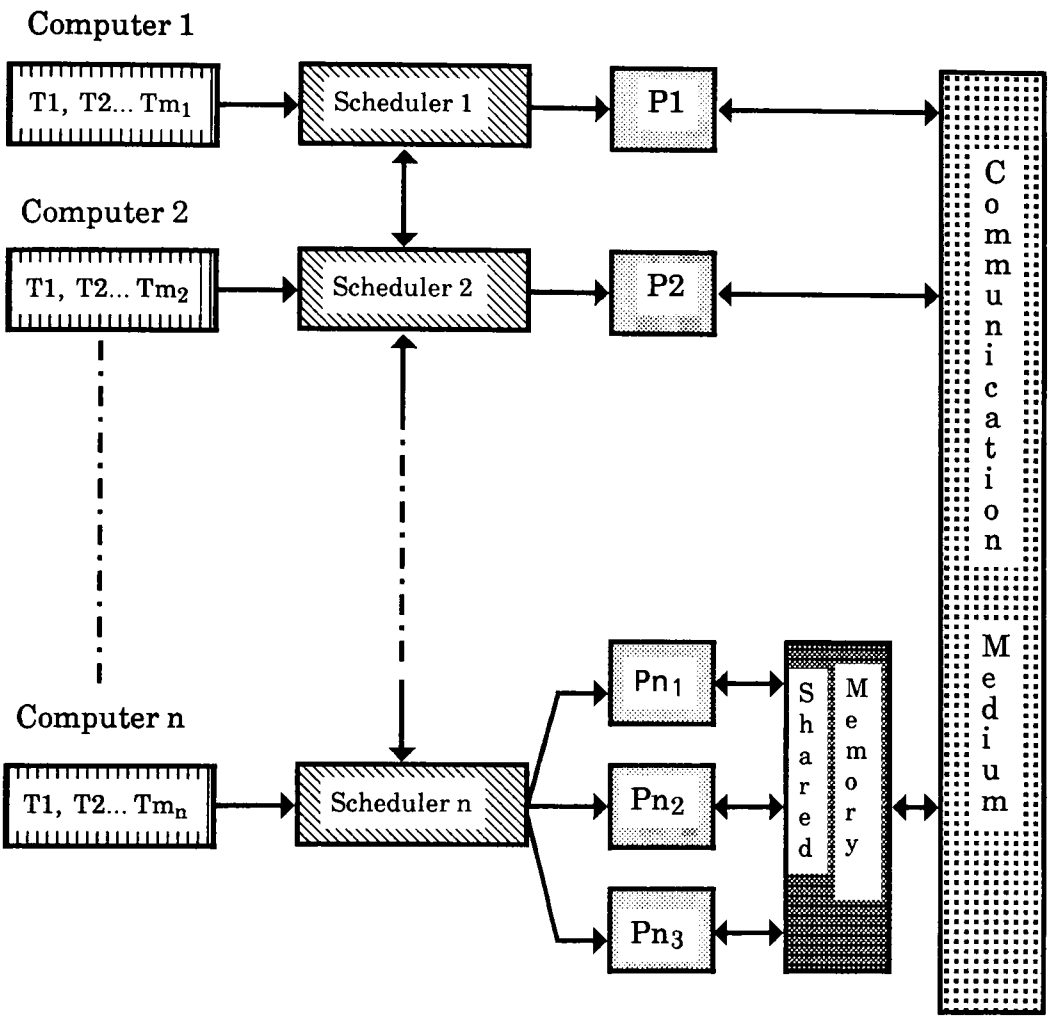
The term Distributed System, in this thesis, will refer to systems with the generic form shown in Figure 1.2. In a distributed system there is more than one computer. Note that a multiprocessor system can be part of a distributed system. Each computer has a scheduler that allocates the tasks waiting at that computer to the processor (or one of the processors if the computer is a multiprocessor). The computers communicate with each other through the communications medium. The arrows between the schedulers indicate that it is possible for a computer to *request*, but not *demand*, services on the others in the system.

Therefore, our distributed system includes the systems belonging to classes one, two and three of the Keefe's classification. Systems belonging to class four are not considered in this thesis.

Load Sharing

One of the main objectives of distributed and multiprocessor systems is to increase the system performance by utilising more resources. A resource that influences the performance of a system more than any other is the processing unit or simply the processor of a system. In a multiprocessor system there is more than one processor to execute the tasks. It is intuitively clear that the best performance will be obtained if all the processors are kept equally busy (provided that there are more tasks than processors).

Since in a multiprocessor system there is only one scheduler allocating tasks, it will be aware of the current computational load on each processor. Therefore one would expect that allocating tasks to processors in a way that



DISTRIBUTED SYSTEM

FIGURE 1.2

maximises the performance would not be difficult. However, since there can be differences in the powers and memory capacities of processors, as well as differences in the computational and communication requirements of tasks, finding an optimal allocation can be quite complicated.

In most distributed systems it is only when a computer requires remote data, or remote execution of a program, that it communicates with other computers in the system. Therefore it is common to find situations, particularly in teaching laboratories, where one computer is being used by so many students that its response time degrades to an intolerable level, while several other computers in the system are either being used very little or not at all. It is clear that in these situations the transfer of some suitable tasks from a heavily loaded computer to the less loaded or an idle computer may improve the overall performance of the system. However, several problems need to be resolved before this can be achieved; for example, how does one computer in the system find the load on others, and how can one be sure that the overhead of transferring a task to a remote computer will not outweigh the benefits?

Thus, we see that both in multiprocessor and distributed systems the problem is essentially that of sharing the computational load of a system among its available processing resources. We shall, therefore, refer to this problem as that of **Load Sharing**. The purpose of load sharing is to improve the performance of a system. The performance of a system can, however, be measured with respect to several criteria; for example, throughput, response time, and completion time. The informal definition of Load Sharing will therefore be *the distribution of a system's computational load among its available processing resources to improve one or more of the performance criteria.* The criteria being improved will be called the **Objective** of load sharing. A system can perform load sharing by implementing what will be known as a **Load Sharing Scheme**.

The role of load sharing in a system can be looked at from two different angles. Firstly, load sharing may be regarded as a feature that helps achieve the objectives of resource sharing and increased performance. In this role, load

sharing is like the feature of system transparency (section 1.1) that encourages the users to share the system resources.

Secondly, load sharing can be viewed as an extension of the responsibilities of a scheduler. For example, a scheduler in a multi-user computer switches the processor among waiting tasks to achieve some objective(s) of the scheduling scheme. This objective can, for example, be the provision of an acceptable response time to all interactive users. When performing load sharing, the scheduler would also have to consider other computers in the system before assigning a task to a processor.

1.4 AIMS AND STRUCTURE OF THESIS

In this thesis we shall study the problem of load sharing in distributed systems. In order to further understand the load sharing problem and to find how it has been resolved in the past, in Chapter Two we review several proposed load sharing schemes for multiprocessor and distributed systems. However, due to lack of common assumptions and differences in their structures it is difficult to establish meaningful comparisons between the various load sharing schemes reviewed for distributed systems. Therefore in Chapter three we identify and suggest solutions for the fundamental issues that need to be resolved when implementing a load sharing scheme on any distributed system. Two reported load sharing schemes for real distributed systems are then compared to find how each has resolved the identified fundamental issues.

The Computing Laboratory at the University of Newcastle upon Tyne has developed an architecture for distributed systems called Unix United. For an existing distributed system based on this architecture it was observed that at most times not all the constituent single-user machines were being used. Hence

there was a possiblity of improving the system performance by designing and implementing a load sharing scheme on it. In Chapter Four we first describe the general structure of Unix United systems and then consider ways of incorporating load sharing in this architecture. Chapter Five describes the details of the existing Unix United system, and how a load sharing scheme was designed for it by resolving the fundamental issues identified in Chapter three. In Chapter Six we describe the implementation of the load sharing scheme designed in the previous chapter.

In order to assess the benefits of implementing the load sharing scheme we carried out several experiments under different system conditions. These experiments are described, and their results discussed in Chapter Seven. Finally, in Chapter Eight, conclusions of the study, and suggestions for future work are outlined.

CHAPTER TWO

A SURVEY OF LOAD SHARING

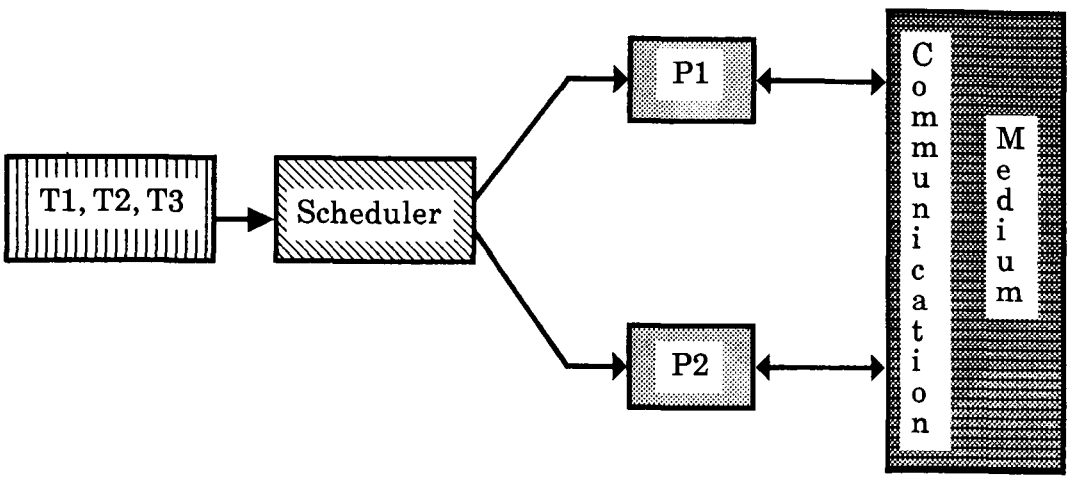
2.1 INTRODUCTION

Many researchers have studied the problem of load sharing either abstractly or for real systems, albeit under different titles such as : Load Levelling, Task Allocation, Software Allocation, Task Assignment, Load Balancing, and Task Scheduling. Although this thesis is concerned with load sharing in distributed systems, in section 2.2 the load sharing schemes for multiprocessor systems are reviewed because most of the early work on load sharing was done for multiprocessor systems. Furthermore, we were interested in comparing the load sharing schemes for multiprocessor systems with those for distributed systems that are reviewed in section 2.3. This chapter is concluded by section 2.4 which discusses the differences beteween load sharing schemes for multiprocessor and for distributed systems.

2.2 MULTIPROCESSOR SYSTEMS

In order to illustrate the load sharing problem, consider an example of a multiprocessor system shown in Figure 2.1. There are two processors (P1, P2) and three tasks (T1, T2 and T3) to be scheduled. Most of the load sharing schemes for multiprocessor systems reviewed in this section make the following assumptions about such a system:

- No tasks arrive at the scheduler queue during the scheduling of tasks T1, T2 and T3;
- At the start of the scheduling the processors in the system are not executing any tasks;



A MULTIPROCESSOR SYSTEM

FIGURE 2.1

- For each Task the following information is supplied:

Execution Cost on Each Processor: The execution cost is some measure of the amount of computation required by the task. This could be, for example, the actual cost in money terms, the number of instructions to be executed, or the CPU time required. The execution cost can depend on many factors including the processor speed. Therefore, if the processors are not identical, the execution costs of a task would vary from processor to processor. The execution cost of a task does not include the cost that is incurred when a task communicates with another task(s).

Communication Costs: The communication cost is some measure of the amount of communication that takes place between two given tasks. Two assumptions are made about the communication costs. Firstly, the communication cost for two tasks executing on the same processor is negligible. Secondly, the communication cost for two tasks executing on

different processors is independent of the processors being used (this will not be true, for example, in a distributed system where the communication costs for neighbouring computers will be less than the computers that are farther apart, in networking terms).

Both the execution and the communication costs are assumed to be positive integers, and a higher value represents higher cost. How these costs are calculated, and who provides them, need not concern us at this stage.

Tables 2.1 and 2.2 give sample execution and communication costs respectively for the system shown in Figure 2.1

	P1	P2
T1	2	1
T2	3	3
T3	2	1

EXECUTION COSTS

TABLE 2.1

	T1	T2	T3
T1		3	0
T2	3		2
T3	0	2	

COMMUNICATION COSTS

TABLE 2.2

Consider a load sharing scheme with the objective of minimising the total cost of executing all three tasks. The total cost will be the sum of all execution costs plus the communication costs of tasks on different processors. Given N processors and M tasks there are N^M possible ways to schedule the tasks to processors. Table 2.3 gives the eight (2^3) possible allocations for the system in Figure 2.1 along with their total costs.

No	P1	P2	EXECUTION COST	COMMUNICATION COST	TOTAL COST
1	T1, T2, T3	-	2+3+2=7	-	7
2	-	T1, T2, T3	1+3+1=5	-	5
3	T1, T2	T3	2+3+1=6	2	8
4	T3	T1, T2	1+3+2=6	2	8
5	T2, T3	T1	3+2+1=6	3	9
6	T1	T2, T3	2+3+1=6	3	9
7	T2	T1, T3	3+1+1=5	2+3=5	10
8	T1, T3	T2	2+2+3=7	2+3=5	12

TABLE 2.3

The second allocation, where all the tasks are assigned to processor P2, results in the minimum total cost. One reason is that this allocation avoids the communication cost between the task pairs T1,T2 and T2,T3.

It might seem that load sharing to achieve minimum total cost is a trivial matter because it would always result in allocating all the tasks to a processor with minimum sum of execution costs, thus avoiding any communication costs. However, this is not necessarily true because sometimes the difference in execution cost of a given task on two different processors is large enough to make up for the communication costs. For example, consider a system consisting of two tasks (T1 and T2) and two processors (P1 and P2). The execution and the communication costs for this system are shown in Tables 2.4 and 2.5 respectively. The four possible allocations with their total costs are shown in Table 2.6.

In this case the allocation of T2 to P1, and T1 to P2 results in the minimum total cost. This rather simple example illustrates that the avoidance of communication costs does not always result in the minimum total cost.

	P1	P2
T1	6	2
T2	1	4

EXECUTION COSTS

TABLE 2.4

	T1
T2	1

COMMUNICATION COSTS

TABLE 2.5

No	P1	P2	EXECUTION COST	COMMUNICATION COST	TOTAL COST
1	T1, T2	-	6+1=7	-	7
2	-	T1, T2	2+4=6	-	6
3	T1	T2	6+4=10	1	11
4	T2	T1	1+2=3	1	4

TABLE 2.6

In the previous chapter (section 1.3) load sharing was informally defined to be the distribution of computational load throughout the system in order to achieve some objective(s). In the above two examples the objective had been the minimisation of total costs. If the execution costs for the example of Figure 2.1 represent the CPU times required for each task, and the communication costs represent the additional CPU time required for communication between a pair of given tasks, then the allocation chosen (all tasks to P2) will use the minimum number of CPU time units (ignoring the *interference* costs that result due to switching of the processor among three tasks).

The minimisation of the total CPU time does not mean, however, that the allocation of all three tasks to one processor satisfies the other objectives also. For example, the response time (for interactive tasks) and the system finishing time (the time when all the tasks in the system have been executed) may not be minimised by this allocation. If the objective of load sharing is, for example, to minimise the response time, then load sharing should involve the evaluation of a function that relates the execution and the communication costs to response time. This function will have to be evaluated for all the possible allocations if the optimal case is to be found.

Now that we have some understanding of the nature of the load sharing problem in a multiprocessor system, we can review the efforts that have been made by various researchers to solve it. Two studies (Stone's and Lo's work) will be considered in some detail, while other work is mentioned only briefly. Stone's work is looked at in detail because it has also been used in several other studies. Lo's work is considered because it illustrates how the optimal solution methods of Stone's work can be adapted for larger multiprocessor systems to give non-optimal, but still acceptable, solutions.

2.2.1 Stone's Graph Theoretic Approach:

In [STONE 77] Stone has used a graph theoretic approach to analyse the problem of finding an allocation of tasks to processors which minimises the total of execution and communication costs. Similar approaches have been adopted in [BOKHARI 79, CHU 80, WU 80, RAO 79]. The algorithm presented in [STONE 77] constructs a graph whose nodes represent either a processor or a task. The edges between the task nodes represent their communication, and are labelled with the corresponding costs. The edges between the processor and task nodes are labelled with weights given by the following equation:

$$W_{iq} = \frac{1}{N-1} \sum_{r \neq q} X_{ir} - \frac{(N-2)}{(N-1)} X_{iq} \qquad \text{Equation 1}$$

where:

W_{iq} is the weight of the edge joining task **i** to processor **q**;
N is the number of processors;
and X_{ir} is the execution cost of task **i** on processor **r**.

For a two processor system with two tasks this means that the edge between say a task node T1, and a processor node P1 is labelled with the execution cost of T1 on P2; while the edge between the task node T1 and processor node P2 is labelled with the execution cost of T1 on P1. For example, the graph obtained for the system in Figure 2.1 is shown below in Figure 2.2. Compare the weights on

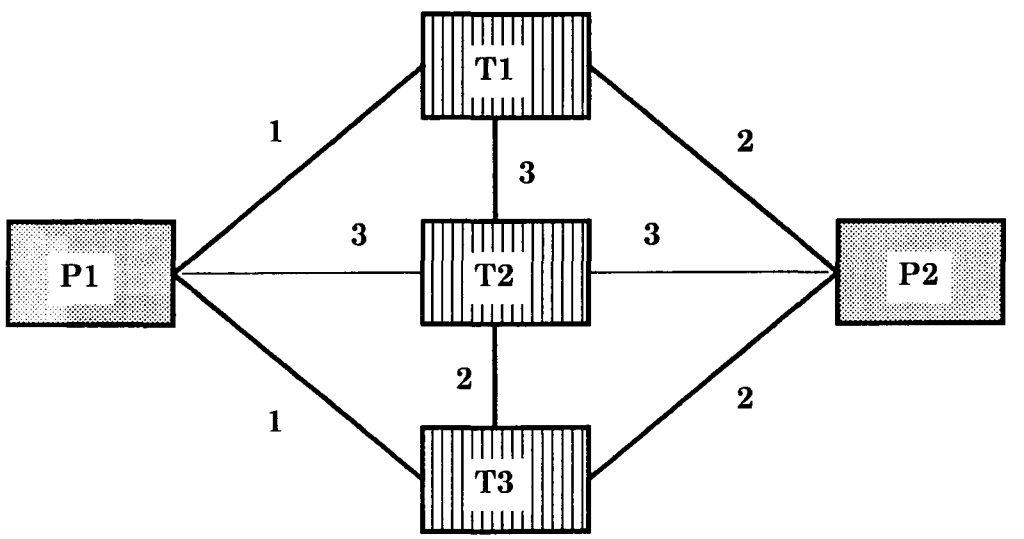


FIGURE 2.2

the edges between the processor and the task nodes with the values given earlier in Table 2.1.

In graph theory a cut set consists of a minimum set of edges whose removal from the graph would split the graph into two disjoint sub graphs. For two given nodes there can be more than one cut sets that leaves them in different sub graphs. The capacity of a cut set is the sum of weights of all the edges in it. The maximum flow minimum cut set is the one with minimum capacity [NARSINGH 74].

Stone shows that if there are N processor nodes in the graph, then an N-way cut could partition the graph into N disjoint subsets such that there is only one processor node in each subset. It is proved that the maximum flow minimum cut set of this graph specifies the optimal allocation of tasks to processors. For example, all the possible cut sets for the graph in Figure 2.2 are shown, with their capacities, in Figure 2.3. The minimum capacity of five is given by the cut set **C2** that leaves all the task nodes with the processor node P2 and none with P1.

While this method of finding an optimal allocation is simple, it has several limitations. Firstly, an extension of this method to an arbitrary number of processors would require an N-dimensional min-cut algorithm which is known to be an NP-complete problem and, therefore, computationally intractable [HARARY 69]. Secondly, this method of finding a minimum cost assumes that all the processors have enough capacity to execute all the tasks that are assigned to them. For example, an extension of this method to include a memory constraint is complex and the solution is, again, NP-complete [RAO 79]. Another drawback of this method is that it does not take into account the decrease in throughput

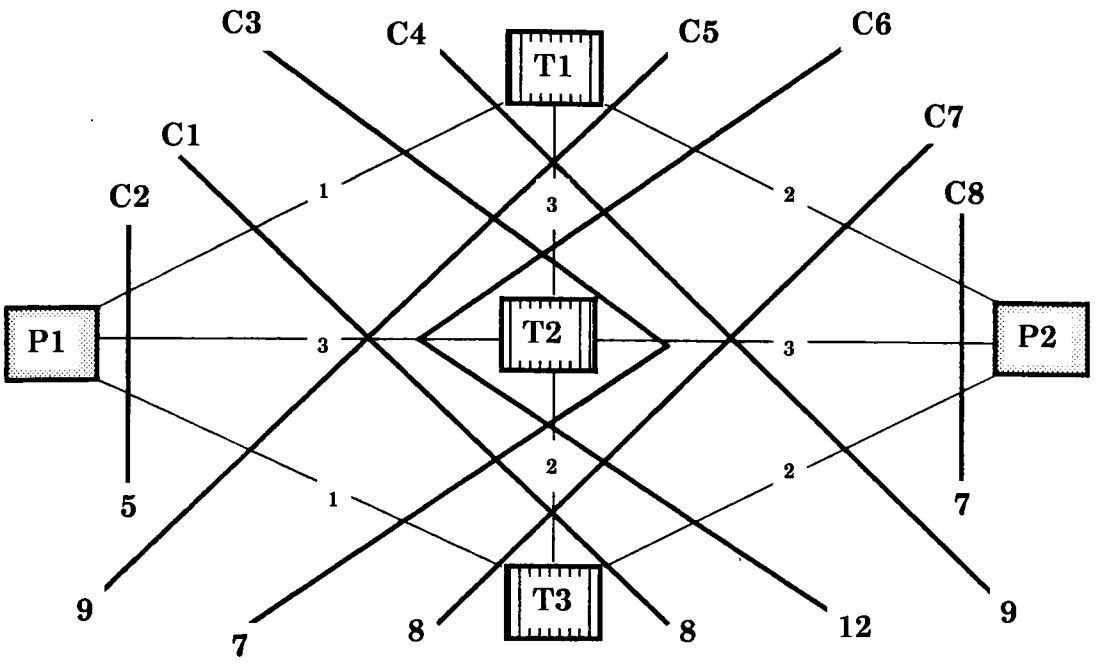


FIGURE 2.3

that results from delays occurring when a processor is switched between various tasks that are allocated to it.

2.2.2 Lo's Heuristic Algorithms:

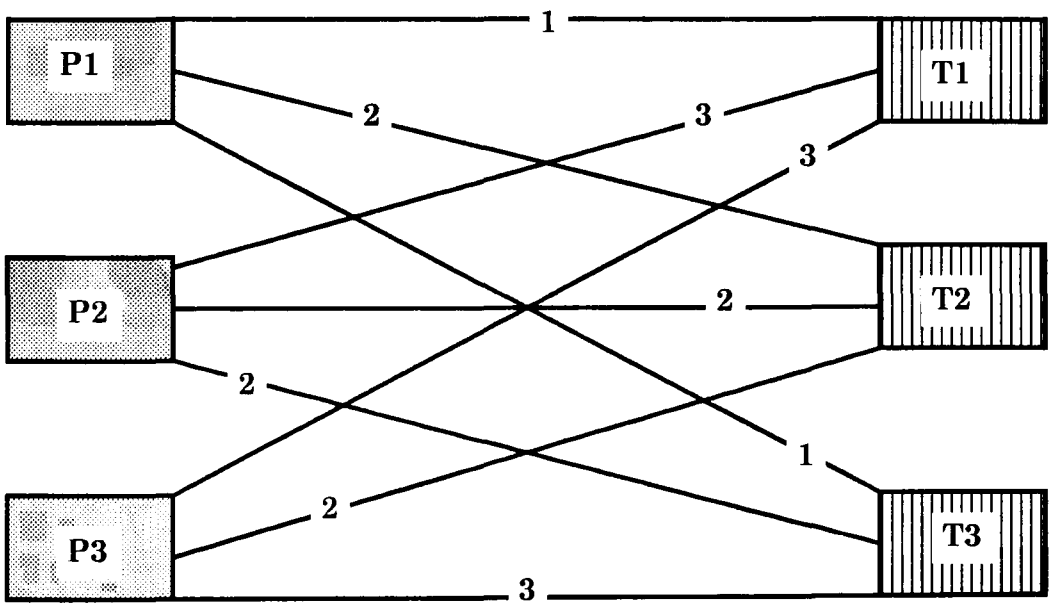
In [LO 84] Lo proposed heuristic algorithms for load sharing with the objectives of minimising the following:

- execution and communication costs;
- execution, communication, and interference costs;

- execution and communication costs, whilst respecting bounds on the number of tasks assigned to each processor.

Minimising execution and communication costs:

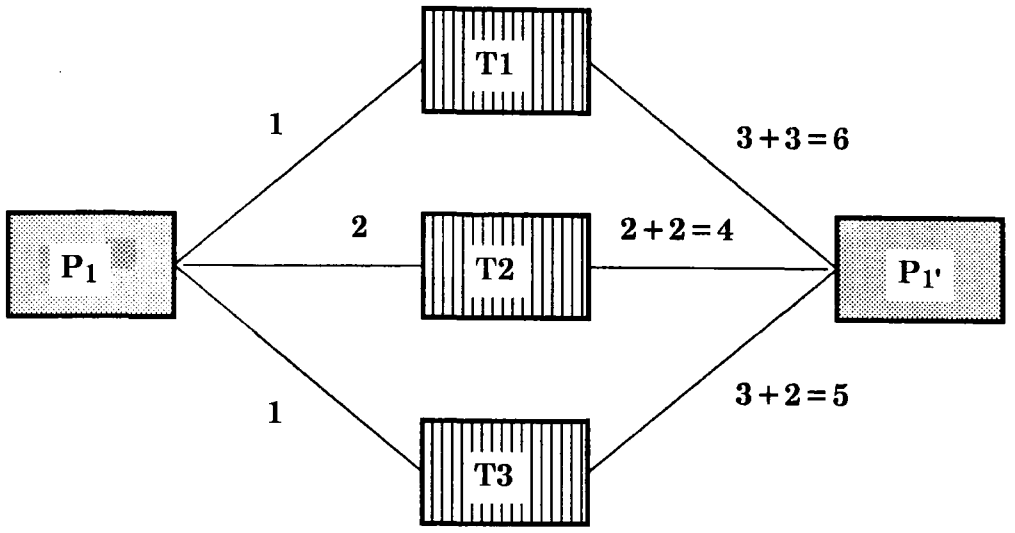
This algorithm consists of three parts: Iterative, Lump, and Greedy. The Iterative part is based on Stone's work [STONE 77]. In this part a graph of task and processor nodes is first constructed as described in section 2.2.1. A graph G_j is obtained from this graph by replacing all the processor nodes, except one special node called P_j, by one node $P_{j'}$. All the edges from each task node to all the processor nodes (except P_j) are replaced by a new edge with weight equal to the sum of weights on the replaced edges. For example, consider the graph G (Figure 2.4) with three tasks and three processor nodes. If P1 is selected as a



GRAPH G

FIGURE 2.4

special node then the graph G_1 (Figure 2.5) will be obtained.



GRAPH G_1

FIGURE 2.5

Thus a graph is obtained with only two processor nodes. The maximum flow minimum cut on this graph would partition it into two disjoint subsets A_j and $A_{j'}$. A_j would contain the processor node P_j and some Task nodes. According to the result in [STONE 77], in an optimal task allocation, the tasks in A_j are assigned to the processor j.

If there are N processor nodes in the system then the above process is repeated N times, once regarding each processor as a special node. Each time the maximum flow minimum cut algorithm is applied to find the allocation of tasks nodes to the special processor node.

If after N iterations all the tasks have been allocated then the allocation is optimal with respect to total execution and communication costs. However, if some tasks are left over, then the graph at the time of last iteration is modified

by deleting all the assigned task nodes from the graph (recall that there will be two processor nodes P_j and $P_{j'}$ in the graph). The execution costs (on Pj) of the remaining tasks are redefined as their original execution cost plus the sum of communication costs between the task and all tasks already assigned to processors other than Pj. The weight on the edges from remaining tasks to Pj are recalculated according to equation 1 in section 2.1. The maximum flow minimum cut algorithm is applied again for each processor.

This second iteration stops when either all the tasks have been assigned or when no tasks are assigned in the last iteration. If all the tasks are allocated then the allocation is optimal. Proofs of this and that no two processors will be assigned the same task and that the iteration process does indeed halt are given in [LO 83].

If it is found that, even after the second set of iterations, there are still some tasks that have not been assigned to processors then the second part of the algorithm, Lump, is applied. This part checks whether it would be cheaper to assign all the remaining tasks to one processor rather than to find an optimal N-way cut for the remaining graph. If so, the remaining tasks are all assigned to the processor that yields minimum total execution cost for remaining tasks. Otherwise, part three (Greedy) of the algorithm is applied. This part locates clusters of tasks with high communication costs. All the tasks in the same cluster are allocated to the same processor.

Minimising execution, communication, and interference costs:

When more than one task is assigned to a processor then they compete with each other for the resources of the processor (e.g. CPU time) potentially resulting in low throughput and longer completion times. In [LO 84] an algorithm is proposed that takes account of the competition for resources. For

this purpose the concept of *interference costs* is used. Interference costs are based on two main factors:

- **Processor based**: The processor based interference costs affect all the tasks on a processor and are dependent upon the amount of resources available on that processor;
- **Communication based**: The communication based interference costs apply to communicating tasks on a same processor which make use of the inter-process communication facilities (e.g. message buffers).

Therefore the interference costs between two tasks i and j, which arise when both are assigned to processor q, can be expressed as the sum of two components:

$$Iq_{(i,j)} = Iqp_{(i,j)} + Iqc_{(i,j)}$$

where:

$Iq_{(i,j)}$ = total interference cost between tasks i and j when allocated to processor q;

$Iqp_{(i,j)}$ = processor based interference cost;

$Iqc_{(i,j)}$ = communication based interference cost.

It is assumed that the interference costs are independent of the processors to which two tasks, i and j, are assigned. Therefore $Iq_{(i,j)} = I_{(i,j)}$.

In order to find an allocation of tasks to processors that minimises the total costs of execution, communication and interference, the same algorithm as in the last sub-section is used. The difference is that this time the weights of the edges between a task node and a processor node are calculated using the equation 2 (instead of equation 1 in section 2.2.1):

$$W_{iq} = \frac{1}{N-1} \sum_{r \neq q} X_{ir} - \frac{(N-2)}{(N-1)} X_{iq} + \frac{1}{2(N-1)} \sum_{1 \leq f \leq M} I_{if}$$

Equation 2

The edges between the communicating task nodes, i and j, have weight equal to their communication costs minus their interference costs ($W_{ij} = C_{ij} - I_{ij}$).

Lo found that the addition of interference costs to the algorithm caused a moderate decline in its performance in finding optimal assignments. However, it dramatically improved the ability of the algorithm to choose assignments with greater parallelism and less completion times.

Bounds on number of tasks assigned to each processor:

Restricting the number of tasks assigned to each processor allows one to take into account the finite capacity of the processors as a factor in task assignments. As a result a more realistic model is obtained which reduces the completion times of the set of tasks by deliberately utilising more processors.

In [LO 84] it is proved that for small systems in which the number of tasks is less than or equal to twice the number of processors, and in which each processor may be assigned at most two tasks, an optimal assignment can be found in polynomial time. For this purpose the graph theory concept of *matching* is used. It is proved that a maximum weight matching corresponds to an assignment which minimises the total execution and communication costs under the constraint that no processor is assigned more than two tasks.

Maximum weight matching, and thus optimal assignments, can be found in polynomial time by network flow algorithms.

For systems with M tasks, N identical processors, and bound B, $\lceil M/N \rceil \leq B \leq M$, as the maximum number of tasks per processor, the following algorithm is used to find sub optimal assignment :

- Construct a graph G with a node for each task T_i and and an edge between each pair of tasks, T_i and T_j, with weight C_{ij} (the communication cost);

- If $M \leq 2N$ then the graph matching algorithm can be applied to obtain an optimal assignment;

- If $M > 2N$, then tasks are grouped in classes using the greedy algorithm mentioned in the last section. The maximum size of a cluster is B/2. The greedy algorithm is repeated until the number of clusters is less than or equal to 2N.

A new graph G' is obtained with a node corresponding to each cluster and an edge between a pair of clusters, R_a and R_b, with weight W_{ij} such that :

$$W_{ab} = \sum_{M_i \in R_a,\ M_j \in R_b} C_{ij}$$

Since the number of nodes in this new graph G' is less then 2N, a graph matching algorithm can be used to produce an assignment of clusters to processors which minimises the total execution and communication costs of clusters, while keeping the number of tasks on each processor less than or equal to the bound B. The graph matching algorithm produces an optimal assignment

for the reduced graph G' but this assignment may be sub optimal for the original graph G. The simulation results reported by Lo show that the algorithm found optimal assignments in 82.4% of cases.

2.2.3 Other Work:

In this section brief descriptions of some further approaches to solving the problem of load sharing in multiprocessor systems are presented.

In [CHOW 79] queuing models for a simple heterogeneous multiprocessor system are presented, analysed and compared. Each model is distinguished by a task routing strategy which is designed to reduce the average task turnaround time by balancing the total load among the processors. The following three task assignment policies were analysed:

Minimum Response Time Policy: An arriving task is sent to the processor with the least value of queue length to service rate ratio.

Minimum System Time Policy: On arrival of a new task, the expected time to complete the tasks already at a processor plus the new task is calculated for each processor. The new task is then assigned to the processor with smallest value for the expected completion time.

Maximum Throughput Policy: This policy assigns each arriving task to the processor that will maximise the expected sum of the individual throughputs of each processor.

The comparison of above policies for a two-processor system showed that the maximum througput policy achieves the best average job turnaround time.

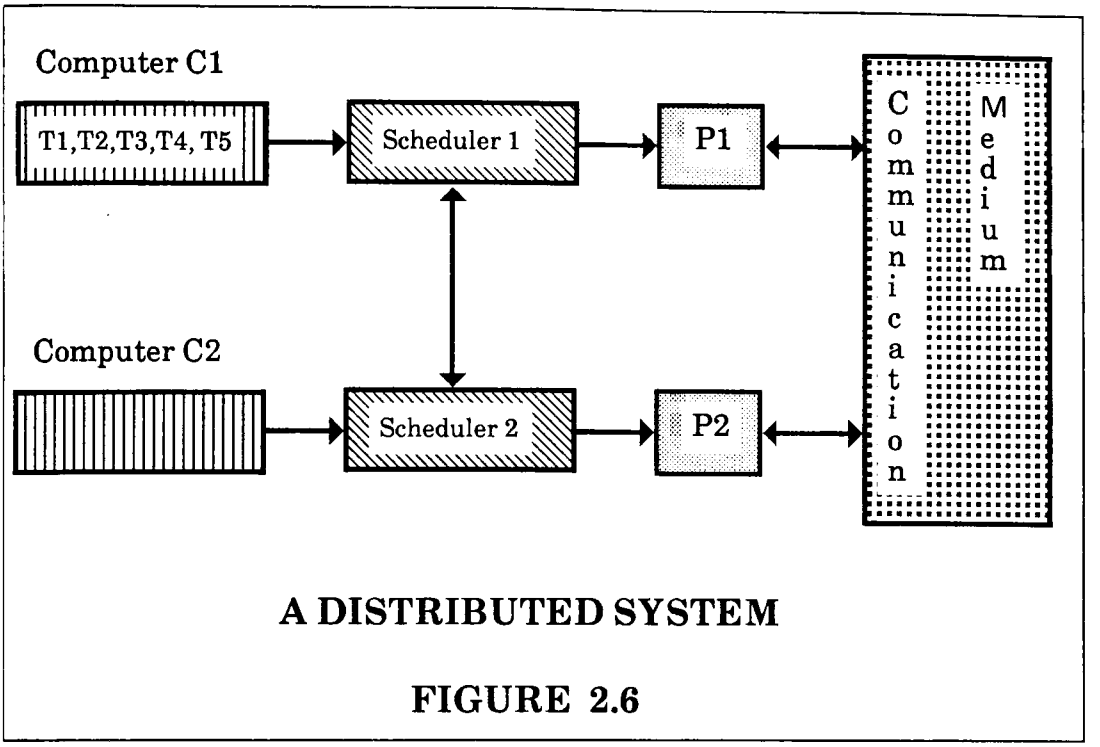
In [PRICE 84] Price and Krishanparsad have proposed a simple heuristic algorithm called the Clustering algorithm. In this algorithm pairs of tasks are

considered for allocation in order of decreasing communication costs. Thus tasks that communicate heavily are more likely to be assigned to the same processor. If neither task of a given pair is currently assigned to a processor, then the pair is assigned to the processor for which the communication costs are minimal and the processor capacity is not violated. The clustering algorithm was tested on simulated networks with uniform data links and processor capacities. It was found that the optimal solutions were achieved for the problems in which communication costs dominated execution costs.

In [NI 81] Ni and Hwang have studied optimal probabilistic load sharing policies to improve the average task turn around time (time from the submission of a task to the scheduler to its completion by one of the processors) for a multiprocessor system. In probabilistic load sharing the scheduler allocates the tasks to processors on a proportionate approach which is independent of the current load on processors. For example, the probability of assigning a task to processors can be proportional to their processing speeds. It is argued that the probabilistic load-sharing policy is easier to implement than the deterministic policies in systems with large number of processors. The scheduling overhead in the probabilistic approach is low because the current processor information is not needed. In addition, to prove optimality of a deterministic load sharing policy is a non-trivial task and more difficult than the probabilistic policies.

2.3 DISTRIBUTED SYSTEMS

In order to illustrate how the load sharing problem relates to distributed systems, consider an example of a distributed system with two computers (C1,C2), as shown in Figure 2.6.



A DISTRIBUTED SYSTEM

FIGURE 2.6

We make the following, rather unrealistic, assumptions about this system:

- Both schedulers use a non-pre-emptive scheduling scheme. Thus, a task does not give up CPU until it is finished.
- All the tasks (T1 to T5) require t seconds of CPU time to execute.
- None of the tasks need to communicate with each other.
- Time Δt is required to transfer a task from one computer to the other;
- No more tasks arrive at either computer until all five tasks have been completed.

Without any load sharing, the completion time of all tasks in the system would be $5t$. If the scheduler on C1 sends tasks T4 and T5 to execute on C2, then the completion time would be $3t+2\Delta t$. As long as Δt is less than t, the completion time would be reduced as a result of load sharing.

In a real system, however, there are several issues that have to be resolved before any load sharing could be performed, for example:

- How does C1 know that C2 has less tasks than itself?
- How is the time for task transfer established?
- What if other tasks arrive at C2 soon after it has received tasks from C1?
- In a distributed system with more than two computers, which one would be selected by C1 to execute its tasks?

These and other issues are discussed in detail in Chapter Three. In the following sub-sections, however, we first review some approaches that have already been taken to resolve the load sharing problem in distributed systems.

2.3.1 A Load Sharing Scheme for MOS:

In [BARAK 85a, BARAK 85b, SHILOH 83] a load sharing scheme has been described for MOS: A Multicomputer Distributed Operating System. MOS is a general-purpose time-sharing operating system that makes a cluster of loosely connected independent homogeneous computers behave as a single machine Unix system. The main goals of the system include network transparency, decentralised control, site autonomy, and dynamic process migration.

The dynamic process migration feature forms the basis of the load sharing scheme implemented on MOS to reduce response times. The scheme is implemented by three algorithms: *local load*, *exchange*, and *process migration*.

The local load algorithm is executed independently by each computer to monitor its own load. The number of processes ready to run and waiting for the CPU is taken every 20 milliseconds and, essentially, averaged over a 1 second period.

The exchange algorithm exchanges the load information between the computers every 1 second. Each computer maintains a small load vector L of size ℓ to store the load information. The first component of this vector (L[0]) holds the value of the local load. The remaining components hold the load values of an arbitrary subset of computers. Every second each processor:

- Updates its own load value;
- Chooses a random integer i such that $1 \leqq i \leqq n$, where n is the number of computers in the system;
- Sends the first half of its load vector to computer i;

On receiving a portion of load vector each computer merges this information with its local load vector, using the mapping :

$$L[i] \text{ --> } L[2i] \quad 1 \leqq i \leqq (\ell/2) - 1$$

and $$Lr[i] \text{ --> } L[2i+1] \quad 0 \leqq i \leqq (\ell/2) - 1$$

Where Lr are the components of the received vector.

The process migration algorithm on each computer picks up the candidate processes in a round robin fashion, and causes them to consider migration. Since it is difficult to determine in advance the computational requirements of a process, only those processes which have already used some minimum CPU time on the current machine become candidates for migration. In this way short processes, or those that have just completed migration, are prevented from migration until they have gained sufficient CPU cycles on the current computer.

The main considerations for process migration are:

- Computer load information that is supplied by the load exchange algorithm. This information is used to estimate the expected response time which a process would get on a given computer.

- Communications overhead. Each process accumulates information about its communication with different computers. If more than half of the process's communication is performed with any single computer, then the process favours running on that computer.

- When the process table of a computer becomes nearly full, migration of new processes becomes essential, and the computer raises its local load (in its load vector) above a certain value to prevent processes from other computers moving in.

The initial performance evaluation of the above scheme confirmed that it was possible to reduce the response time of MOS by dynamically load sharing the system.

2.3.2 Load Sharing with Maitre d' :

In [BERSHAD 86] a load sharing scheme called Maitre d' (French for *host* or *server*) has been described. It runs at the user level on computers (VAX 750, 780, 785, MicrovaxII) running Unix systems. All computers that are to be able to offload tasks to other computers run a *client* process that maintains the list of all server machines, including status information as to whether or not they are currently willing to accept tasks.

The computers that are prepared to receive tasks from others run a *server* process that maintains status connections with client computers and accepts remote tasks. A server declares itself available if its five minute load average is less than some threshold (receive threshold), and there are fewer than a given

number of active users logged in to it. Normally, all the machines would run both the client and the server processes, thus allowing all the computers to exchange tasks with all the other computers.

Only those application programs that are modified to run under Maitre d' participate in the load sharing scheme. These application programs first contact the local client process, asking for an available computer. If the local load is less than the send-off threshold, or no remote computers are presently available, the application is informed by the client process to perform itself locally. Otherwise, the client process carries out a round-robin traversal of its list of available servers until it finds a computer where the server has advertised a willingness to accept tasks. In this case the client passes back the internet address of this server to the application program. The application then requests the remote server to execute it.

It is reported that the statistics collected over a one year period show that the load sharing scheme has, on average, halved the response times for programs that were modified to run under Maitre d'.

2.3.3 Load Sharing to meet Real Time Constraints:

Some applications require that the tasks submitted to a distributed system must be executed within a given time limit. Therefore one of the objectives of load sharing could be to meet the real time constraints of tasks in the system.

In [RAMAMRITHM 84, ZHAO 85] Ramamrithm and Zhao have proposed an algorithm that performs load sharing in a loosely coupled distributed system. In this algorithm the schedulers have to consider two types of tasks:

I - **Periodic:** A periodic task, say with period T, is one which has to be executed once every T time units. These tasks have known start times, deadlines and computation times.

II- **Non-periodic:** A non-periodic task is one which occurs in the system just once and at an unpredictable time. Upon arrival it is characterised by its deadline and its computation time.

A task is said to be **guaranteed** if under all circumstances it will be scheduled to meet its real time requirements. The aim of the load sharing scheme is to guarantee as many tasks as possible, by utilizing the resources of the entire system. The scheduler on each computer in the distributed system consists of four components that work together and with components running on other schedulers to achieve the objective of load sharing. These four components are:

I - **The Local Scheduler** deals with the tasks that arrive directly (not from guarantee routine (described below). If the task cannot be guaranteed, it is put in the bidder's queue to be tried by another scheduler in the system.

II- **The Dispatcher** determines which of the guaranteed periodic and non-periodic tasks is to be executed next. The decision is based on earliest-deadline-first scheme. The run time cost of the dispatcher is part of the computation time of every task.

III- **The Bidder** receives requests from the local scheduler to send a task to another bidder. It also receives requests from other bidders and tries to guarantee the received tasks on the local scheduler.

IV- **The Guarantee Routine** determines if there is enough surplus processing power to execute a newly arriving task before its deadline, without effecting the tasks guaranteed already.

Note that for the above scheme to work, the bidder needs to know how much surplus processing power is available on other computers. This information can be passed in bids or on request. When bidding for a task, the resources are not reserved for it. This is because some times the surplus situation gets changed by the time a task arrives after a successful bid.

The above strategy is extended in [ZHAO 87] to handle the resource requirements of tasks in addition to deadlines. A heuristic function is developed which synthesizes various factors of real-time scheduling considerations to actively direct the scheduling process to a plausible path to follow, and prevents it from considering implausible paths. The simulation results show that in most cases the schedules found by the heuristic algorithm are optimal or close to optimal.

2.3.4 Other Work:

In [EAGER 84, EAGER 85] three types of load sharing schemes for distributed systems have been studied. In the **random scheme** a computer is selected at random, and a new task is transferred there for execution. No other system information is therefore used by this scheme. In the **threshold scheme** a computer is selected at random, but this time the selected computer is probed to determine whether the transfer of the task would increase the number of tasks already in service or waiting for service (queue length on the selected computer) above a given threshold value. If not, then the task is transferred to the randomly chosen computer, otherwise another computer is randomly selected

and probed. In the **shortest scheme** a given number of distinct computers are chosen at random, and each is polled in turn to determine its queue length. The task is then transferred to the computer with the shortest queue, unless it is greater than or equal to some threshold queue length in which case the task is executed at the originating computer. The study concluded that the threshold scheme was better than the random; although the shortest scheme used more system information, its performance was not significantly better than the threshold policy.

In [LIVNY 82] three algorithms for a load sharing scheme have been studied. In the **state broadcast algorithm** whenever the state of a computer changes because of the arrival or departure of a task, the computer broadcasts a status message that describes its new state. The decisions by all the computers in the system to transfer tasks are based on this information. In the **broadcast idle algorithm** a computer broadcasts its state only when it enters an idle state. This message alerts other computers and they start considering task transfer. In the **poll idle algorithm** a computer starts polling a subset of system computers for tasks when it enters an idle state. The simulation results showed that each algorithm was good for certain conditions, and therefore it was not possible to select one as being the *best.*

In [ZATTI 85] a multivariable information scheme to balance the load in a distributed system has been described. The objective was to minimise the response time of a particular set of jobs. However, instead of using one general load index which could give some indications of the expected response time of a new task, a multivariable information scheme is used where a separate index for each resource (CPU, free memory, I/O traffic) is maintained over the whole system. A machine is selected to execute a task by matching each of the machine's available resources with the corresponding requirements of the task.

The experiments performed with a prototype implementation show that the load sharing scheme is able to make the right choice on a set of test jobs between 55 and 80% of the times.

In [CAREY 86] an approach to query processing in a locally distributed database system has been presented which integrates query processing and load sharing. This approach uses the load information to dynamically choose from among those sites that have copies of the relations referenced by the query. Simulation results indicate that load-shared query processing can provide improvements in both query response times and overall system throughput as compared to schemes where execution sites are either statically or randomly selected.

Dynamic load sharing in locally distributed data base systems is also investigated in [YU 86]. Here the database is partitioned and distributed among multiple transaction processing systems, and a common front-end processor is employed for transaction placing. Simulation studies of four different load sharing strategies showed that dynamic strategies which considered both balancing the load and reducing remote processing requests can be superior to the optimal static strategies.

In [HAC 85] a dynamic load sharing policy is described for a distributed system consisting of a number of hosts connected by a local area network. The file system modelled in this study was that of the LOCUS system [POPEK 83] which allowed replicated files. The load sharing algorithm selects the site for process execution and also decides which file to use. The algorithm bases its decision on a periodically collected system information. The simulated experiments show that the performance of the system improved by up to 21.7% for file accesses, and up to 19.6% for the CPU intensive jobs.

Two different dynamic process assignment schemes for load balancing in distributed systems are studied in [LELAND 86]. In the *initial placement* process assignment scheme the operating system assigns a process to one of the computers (not necessarily the one on which the process was submitted) at the time of process creation. In the *process migration* scheme the operating system may reassign (*migrate*) a process to different computers while the process is being executed. In order to model the behaviour of these two schemes, the resource utilisation (CPU usage and disk accesses) of over 9.5 million Unix processes was observed. These processes were created during normal professional computing by the users in two computing laboratories. The data obtained was then used to carry out simulation studies of the two process assignment schemes. A comprehensive set of results are given in [LELAND 86], however, the final conclusions were that:

- Initial placement alone, or process migration alone, provides significant reductions in the response times of processes requiring large amounts of CPU time.

- Initial placement and process migration provide still more improvement when used together, especially as the number of processes involved increases.

- Simple heuristics for initial placement and process migration suffice to protect ordinary processes from harm while providing dramatic benefits for CPU-intensive processes.

In [WANG 85] a hierarchical taxonomy of load sharing schemes, based on the amount of system information employed, has been presented. The load sharing schemes in different classes are then compared by observing their performance under simulated conditions. It is concluded that widely varying performance is

obtained with different schemes. Furthermore, with the same level of information available, *Server Initiative* algorithms (where free servers find busy servers and offer to execute their tasks) have the potential of outperforming *Source Initiative* algorithms (where busy servers look for free servers).

In [NI 85] a distributed algorithm for load sharing has been described. It is reported that by employing the algorithm a significant system performance improvement over no load sharing effort is shown by simulating a sample five-processor distributed system. The results also indicated that the definition and measurement of processor load is not required to be very accurate and a good estimate of processor load is sufficient to improve the system performance.

2.4 CONCLUSIONS

In this chapter we have considered several load sharing schemes for multiprocessor and distributed systems. It is difficult to make direct comparison between these load sharing schemes, particularly those for distributed systems, because they are designed for different systems with different objectives. Therefore, in the next chapter we identify the fundamental issues that are common to all load sharing schemes for distributed systems. However, it is possible to make general comparison between load sharing schemes for distributed and load sharing schemes for multiprocessor systems.

We observe that most schemes for load sharing on multiprocessor systems assume that there is one scheduler that makes use of prior knowledge of tasks' computational requirements in assigning them to processors. Furthermore, many of these schemes take account of the communications costs between tasks executing on different processors. Generally, no attention is given to the additional cost of assigning a task to a particular processor, or to the fact that

for real systems it is almost impossible to determine a task's computational and communication requirements in advance.

On the other hand, most load sharing schemes for distributed systems do not assume prior knowledge of tasks' computational requirements. The decision to transfer tasks is based on the current state of the system, and is made by independent schedulers on each computer in the system. In contrast to the multiprocessor schemes, load sharing schemes for distributed systems do not generally consider the additional communications costs between tasks executing on different computers, though the costs of transferring a task from one computer to the other are usually taken into account.

The above survey also indicates that it is difficult to develop efficient and computationally feasible algorithms that will give optimal solutions even for very simple, and possibly unrealistic, models of task assignment. Heuristic methods employed by load sharing schemes for distributed systems do not guarantee an optimal solution but they have proved, in several simulated studies, to yield results that provide an acceptable compromise between the quality of a solution and the computation required to obtain it.

CHAPTER THREE

ISSUES IN LOAD SHARING DISTRIBUTED SYSTEMS

3.1 INTRODUCTION

The examination of load sharing algorithms presented in the last chapter has revealed a variety of proposed approaches to solve the load sharing problem both in multiprocessor and distributed systems. However, due to lack of common assumptions, and differences in the systems for which the algorithms were designed, it is difficult to establish meaningful comparisons between these various approaches.

In this chapter, therefore, a common set of fundamental issues regarding the load sharing problem in distributed systems are identified. Some of the issues will no doubt be applicable to multiprocessor systems as well, but the discussion in this chapter will only be concerned with distributed systems.

The issues are presented in the following section. In section 3.3 it is shown that two of the load sharing schemes for distributed systems, discussed in Chapter Two, differ from each other only in the way they resolve each of the identified issues.

3.2 LOAD SHARING ISSUES

This section identifies the issues that have to be resolved by all load sharing schemes. While discussing these issues we suggest some general solutions only because the precise solutions will depend on the characteristics of the system on

which load sharing is being implemented. The following sub-sections present each issue separately.

3.2.1 OBJECTIVE:

The objective of load sharing is generally understood to be *the improvement of system performance.* Since the performance of a system can be measured according to several criteria [FERRARI 83], one needs to be more precise in stating which criterion, or the combination of criteria, is being improved. Some performance criteria that may be improved by applying load sharing in a distributed system are:

- response time of the system;
- completion time of a given amount of computation;
- system throughput;
- ability to meet real time constraints;
- execution cost (e.g. in terms of time) of a given set of tasks.

The above list is not exhaustive. Furthermore, the improvement in one criterion may result in the deterioration of another. For example, reduction in average response times may reduce the overall throughput of the system. Which criterion is selected as an objective of a load sharing scheme depends on the purpose of the system being considered. A computing service dealing mainly with batch oriented tasks may wish to increase its throughput by load sharing. On the other hand, a computer system controlling an industrial plant may use load sharing to improve the ability of the system to meet the real time demands of the plant.

Although it is possible to imagine a distributed system where each computer has its own objective for load sharing, it is very unlikely in practice because of the complexity of algorithms to implement such a scheme. In this study, therefore, we shall assume that all the computers have same objective for participating in a load sharing scheme.

Which ever criterion is selected, the essential point is that before implementing a load sharing scheme, the designer should clearly and precisely define its objective.

3.2.2 GRANULARITY AND MECHANISM:

The discussion about sharing the computational load has so far been based on the implicit assumptions that the computational load of a system is already *partitioned* into tasks that can, somehow, be distributed across the system. In practice, the implementation of any load sharing scheme would require the designer to make decisions about the form of the tasks, and the mechanism for their transfer to other computers in the system.

The form of the tasks to be transferred is important for at least two reasons. Firstly, if the load is presented to the system in a form that is not suitable for transfer, then the load sharing scheme has to provide a facility for *packaging* the load into tasks of an appropriate form that may be send to another computer for execution.

Secondly, the performance of a load sharing scheme, and therefore the degree to which it achieves its objective, will depend on the granularity of the tasks that are exchanged between the computers. A finer granularity will, for example, allow the computers to share the load more evenly, but since there will be an overhead associated with the transfer of each task, the total overhead of

transferring a given amount of load would be higher than it would be for the tasks of coarser granularity. Thus, some balance has to be achieved between the two conflicting aims for finer granularity and lower overhead.

At the hardware level the load on a computer consists of a number of machine operations to be carried out. From the user's point of view, in a general purpose distributed system most of this load gets generated in an hierarchical manner. At the top level there are **users** who issue **commands**. The commands result in the running of **programs**. A program may consist of **modules** that may be executed concurrently. The modules would consist of **instructions** that map into machine **operations** to be performed.

In a load sharing scheme the granularity of tasks to be transferred to other computers can, at least theoretically, be placed at any stage in the above hierarchy of load generation. Thus, at the coarsest granularity one can imagine the transfer of users, say at login time, from a heavily loaded computer to an idle computer. On the other hand, at the finest level of granularity, individual machine operations could be carried out by a less busy remote computer.

From the system's point of view, however, there may be no distinction, as far as their execution is concerned, between a user, a command, a program, and a module. For example, a user may well be represented by a process that runs other processes to run commands. The commands, in turn, generate more processes to run particular programs. The modules inside programs may then be executed by separate processes. Therefore, as far as the system is concerned, the transfer of a task of any granularity to another machine effectively amounts to a remote process execution. There are basically two mechanisms by which a system can achieve remote process execution:

i) Code Transfer: In this mechanism, the complete image (machine code and its data) of a process is transferred from one machine to the other. This mechanism can, for example, be used by a load sharing scheme using the module level granularity in a completely homogeneous, or an architecturally homogeneous (identical machines running different operating systems) distributed system.

ii) Remote Call: In this mechanism, a computer with heavy load requests a computer with less load to execute a process on its behalf. The code to be executed already resides on the remote machine and is identified by the parameters passed with the request. This mechanism may be used to implement a load sharing scheme that employs a command level granularity in a completely homogeneous or a functionally homogeneous (different machines running identical operating systems) distributed system.

The level of granularity chosen for a load sharing scheme would depend on the features of the system being considered. If the load sharing scheme is considered as a part of the design of a distributed system, then one could select a particular granularity and provide mechanisms in the system to support it. Alternatively, if the load sharing is implemented as an after-thought in a distributed system that already provides some mechanisms for remote process execution, then the granularity can be selected to suit this mechanism.

3.2.3 INITIATOR:

As a result of load sharing in a distributed system, a busy computer, the source, transfers its task(s) to a less busy (or even idle) computer, the server. The distinction between the source and the server is based on the particular task being transferred. In another transaction, the roles of the source and the server may well be reversed.

The discussion of load sharing so far may imply that it is always the source that takes the initiative and searches the rest of the system for a possible server. However, this need not be the case. In some schemes it could be the server that, upon becoming less busy or idle, starts searching for work and a possible source in the system.

In [WANG 85, EAGER 85] both types of scheme have been considered in some detail. It is reported that in source-initiated schemes the queues of tasks tend to form at the servers, while in server-initiated schemes the queues tend to form at sources. The study concludes that, provided the communication costs are not a dominant factor, and both schemes use same amount of information in making their load sharing decisions, the server-initiated schemes outperform the source-initiated schemes.

We believe that, at least in theory, there is also a third possibility, namely that of having a mixed scheme. In this scheme both the server and the source may initiate the load sharing. Thus when the load of a computer exceeds a certain maximum, or falls below a certain minimum, it starts searching the system for a possible server or a source respectively. Such a scheme has not been reported in the literature, possibly because of the complexity of its implementation.

In discussing the remainder of the issues, where necessary, clear distinction will be made between the source and server initiated schemes. Otherwise, a source initiated scheme will be assumed.

3.2.4 INITIATION:

In the last section we only considered the nature of the initiator of load sharing. We did not discuss, however, the timing of this initiation. In other

words, when does a computer in a distributed system invoke the load sharing scheme? There are two obvious possibilities:

i) Periodically: Every t seconds a computer starts the load sharing scheme and considers transferring one or more of its tasks to another computer. Periodic initiation can either be done independently by each computer (even allowing different values of t), or it can be done collectively by all the computers at the same time. The practical difficulties in synchronising all the computers would make the latter approach almost impossible to implement. Periodic initiation allows the possibility of transferring a task that is already executing on the local computer (pre-emptive schemes).

ii) Task-arrival: When a task arrives (or when a task completes for server-initiated schemes) the computer starts the load sharing scheme and makes decisions about the task's transfer. For example, if command level granularity is being used, then before carrying out the command issued by the user, the computer may compare its load with the load on other computers. It could then decide whether to carry out the command locally or transfer it to another computer.

It may be suggested that there are other times when a computer may initiate load sharing, such as when a computer's load exceeds certain threshold value. We believe that this is a special case of periodic initiation. In this case the first step of the scheme would be to check whether the local load is above or below a certain value. If so, the rest of the scheme is carried out, otherwise it is stopped until the next periodic invocation.

It is also possible to imagine a scheme that combines the above two possibilities. For example, the scheme is normally invoked upon task arrival;

however, if a task does not arrive for certain period of time, then the periodic method gets invoked by default.

Which of the above two methods is selected by a load sharing scheme would depend on the nature of demands being put on the system as well as the overhead of initiating the scheme. If, for example, the system is highly interactive and the load sharing scheme is relatively cheap, then the task arrival method is likely to be employed.

3.2.5 INFORMATION DEPENDENCY:

After its initiation on a particular computer, the load sharing scheme has to make the following two decisions:

- Should one or more of its tasks be transferred to another computer?
- If so, to which computer should they be sent?

The rules that determine the answer to the first question are called the **transfer policy**, while the rules used to answer the second question are known as the **location policy** [EAGER 84]. Both these policies may use information about the computational requirements of the task being considered, and the current state of the system. Generally it is not possible to determine in advance the precise computational requirements of a task. Therefore, a load sharing scheme is likely to use information about system state only. Hence, in this section we consider the types of system information that may be employed by the transfer and location policies of a load sharing scheme.

In [CHOU 82, NI 85, LELAND 86] load sharing schemes have been classified as either being **static** or **dynamic.** In static schemes the processor on which a given task must execute is predetermined, independently of the current state of

the system. For example, tasks can be assigned to computers according to probabilities fixed in proportion to their processing speeds.

In dynamic load sharing schemes, the computer to execute a given task is selected according to the current state of the system. For example, a dynamic scheme may choose to assign the new task to the computer with the shortest queue of waiting tasks.

We believe, however, that the above definitions of static and dynamic classes are not satisfactory for at least two reasons. Firstly, all load sharing schemes, including static, make use of some system information, otherwise load sharing would be impossible. In the probability-based example given above, the static scheme has to know the correspondence between the probability values it generates and the computers in the system.

It may be argued that the static schemes use information that remains fixed, while the dynamic schemes use information that varies during the operation of the system. Unfortunately, even this argument does not resolve the issue. Although the correspondence between the probability values and the computers may remain constant, the availability of all the processors in the system cannot be guaranteed to remain constant (unless the system is 100% reliable, or a crash of one computer stops the whole system). Thus, both static and dynamic schemes need to know about the availability of computers in the system.

The second objection to the classification of load sharing schemes into static and dynamic schemes is that it does not reflect the amount of system information that is used by the dynamic schemes to select a computer. In our shortest-queue example above, the dynamic scheme could have also taken account of the communications overhead, and the processing speeds of the

available computers, but this would not have been indicated by our classification.

In [WANG 85] a taxonomy based on the system information used by load sharing algorithms has been presented. The algorithms have been divided into classes 1 to 7. Algorithms in a higher numbered class use all the information employed by the algorithms of next lower class as well as some extra information. We think that, although useful for comparing the performance of algorithms in its different classes, this taxonomy cannot be used to classify various algorithms incorporated in several reported load sharing schemes. This is because algorithms in different schemes use different types of system information, but not necessarily in the hierarchical manner that is required by the above linear taxonomy.

Instead of attempting to classify the load sharing schemes according to the nature of system information employed by their transfer and location policies, we list below different types of information, a subset of which may be used by any load sharing scheme:

i) Processor Availability: As discussed earlier, this information is used, either explicitly or implicitly, by all load sharing schemes. All this piece of information tells is whether or not a computer is working, and therefore available to be considered by the transfer and location policies of a load sharing scheme. It must be stressed that this information is just concerned with the availability of the computer, and does not tell us, for example, that the computer is too busy to consider tasks from other computers.

ii) Random Choice: A heavily loaded computer may randomly select another computer to share its load. This random choice may or may not be weighted with the other features of the receiving computers such as their processing speeds.

iii) Processing Speeds: In systems that are not fully homogeneous, the processing speeds of the computers may be one of the factors that is taken into account by the transfer and location policies of a load sharing scheme. For example, provided that everything else is the same, a computer may prefer to share its load with a faster computer. The speeds of the computers will, of course have to be represented in a form that makes their comparison possible.

iv) Processor Functionality: In a distributed system it is possible that certain computers are dedicated to specific tasks, such as file handling or graphics calculations. In such an environment a load sharing scheme would have to use the information regarding the functionality of computers in the system. Thus, a computer wishing to transfer a task would then have to consider whether the candidate computer has the functionality required to perform that task.

v) Communication Costs: If the computers in the system are connected by different speed links, or if the system is not fully interconnected, then the communication costs, for example, in terms of time used, can be different for different pairs of computers. In a tightly coupled computer system the differences in these costs may not merit attention, but in a loosely coupled system, particularly one spread over a wide area, these costs can influence the performance of load sharing scheme quite significantly. Although it is difficult to establish exact communication costs, especially under varying traffic conditions, the load sharing scheme may require the participating computers to take account of some relative measure of communication costs in establishing a computer to share their load.

vi) History: Some load sharing schemes, such as [BERSHAD 86], require the computers to consider the load sharing decisions they have made in the past before making a new decision about sharing their load. For example, if a

computer has already sent a task to a particular computer in the system, then it may not consider the same computer for another task until the previous task has been completed. The concept of using history can, indeed, be extended so that the computers learn from their past decisions. So, for example, if the choice of a particular computer in the past was found not to be beneficial, then a computer may decide to exclude that computer from the list of possible computers that may share its future load.

vii) Current Load: The current load on computers is the most important factor that a load sharing scheme could require a computer to consider before selecting another computer to share its load. If a task can be performed locally, there is no point in a computer sending this task to another computer which has more load than itself! Many indices can be used to express the load existing on a machine at a given time, for example: CPU utilisation, the length of the ready queue, the stretch factor (defined as the ratio between the execution time of a process on a loaded machine and its execution time on the same machine when it is empty) [FERRARI 86], paging activity, terminal I/O, disk transfers, and more complicated functions of these simple variables.

How often the values for different types information are calculated and updated is a separate issue, and discussed later. For the time being we are simply saying that each computer could maintain some indication of load on itself and other computers in the system. This information could then be used by the transfer and location policies of the load sharing scheme.

We are now in a position to give better and more applicable definitions of static and dynamic load sharing schemes. A static load sharing scheme makes use of only that system information (system *knowledge* would be a better term) that it **assumes** remains constant during system operation. For example, if a load sharing scheme assumes that the computer speeds and availabilities

remain constant (quite reasonable assumptions!), and then assigns tasks to computers according to probabilities that are fixed in proportion to their speeds, then that scheme would be regarded as being static.

On the other hand, if a load sharing scheme makes use of even one piece of system information that it **knows** can vary during system operation, then that scheme would be dynamic. To overcome the second objection to the earlier definitions, we must state clearly the system information that is used by the load sharing scheme either explicitly or implicitly.

Which of the seven types of system information mentioned above are used (either as varying or constant quantities) by the transfer and location policies would depend on the configuration and the attributes of the system, as well as the objective of the load sharing scheme being considered. One general point that can be made, however, is that the use of more information may allow the transfer and location policies to make better decisions, but it will, no doubt, carry more overhead of maintaining and using that information.

Furthermore, system information, particularly load information, has a tendency to become out of date very quickly, hence its validity is always in question. Therefore, the benefit to be gained by using certain type(s) of information must be assessed against its overhead and the cost of possible inaccuracies.

3.2.6 COMPUTER CONNECTIVITY:

The discussion in previous sub-sections may have implied that, both for transfer of its tasks and for exchange of information, a computer considers all

other computers in the system. This is not necessarily true in either case. To discuss this issue further, we define the following terms:

The **send connectivity (SC)** of a computer is a set of all computers in the system, excluding itself, that it considers likely to execute its tasks.

The **receive connectivity (RC)** of a computer is a set of all computers in the system, excluding itself, that are likely to send it their tasks for execution.

The **other computers (OC)** of a computer is a set of all the other computers in the system excluding itself.

To understand these terms further, consider the distributed system of six computers shown in Figure 3.1. A single headed arrow going from computer 3 to

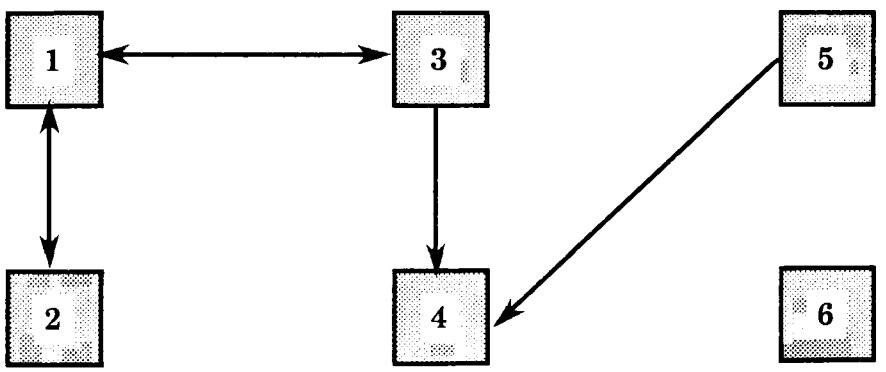


FIGURE 3.1

computer 4 means that computer 3 can send tasks to computer 4 and, conversely, computer 4 can receive tasks from computer 3. A double-headed arrow between computers 1 and 2 means that computer 1 can send tasks to, as

well as receive tasks from computer 2, and vice versa. The corresponding SC, RC, and OC sets for each computer are shown in Table 3.1 .

COMPUTER	SC	RC	OC
1	{2, 3}	{2, 3}	{2, 3, 4, 5, 6}
2	{1}	{1, 3}	{1, 3, 4, 5, 6}
3	{1, 2, 4}	{1}	{1, 2, 4, 5, 6}
4	{ }	{3, 5}	{1, 2, 3, 5, 6}
5	{4}	{ }	{1, 2, 3, 4, 6}
6	{ }	{ }	{1, 2, 3, 4, 5}

TABLE 3.1

Note that arrows between computers only indicate their connectivities for the sake of load sharing. A physical connection may well exist between all of the computers.

We note that SC and RC can be proper subsets of OC, and need not be equal. Although it is very likely that if a computer can send tasks to another then it will be able to receive tasks from it as well, but this is not necessary (see SC and RC for 3). In fact, it is possible for a computer in a distributed system not to participate in the load sharing scheme at all (computer 6).

In a dynamic load sharing scheme, system information may have to be exchanged between computers. Typically, a computer will require the information about the computers in its SC, while it needs to supply, either voluntarily or on demand, information about itself to computers in its RC.

Indeed, it is the cost of exchanging the system information that generally restricts the RC and SC of a computer in a large distributed system to be subsets of their OC. In other words, computers do not consider all other computers for

task transfers because that would imply having to exchange system information with all of them. If there are N computers in a system, then each computer would have to exchange system information with N-1 other computers. If each computer has information to exchange with all the others, and assuming equal cost (C) for each such exchange, then the total cost of exchange would be:

$$C \times N \times (N\text{-}1) \text{ or } C \times (N^2 - N)$$

Thus, the total cost is proportional to the square of the number of computers. Therefore, the overhead will increase quite rapidly as the number of computers participating in the load sharing scheme increases. By considering only a subset of all the computers in the system, the transfer and location policies of a load sharing scheme may not make the best possible decisions, but it would keep the overhead of load sharing under control.

The SC and RC sets of a computer will be determined by the characteristics of the system, such as the communication costs between pairs of computers. Some sophisticated load sharing schemes may even allow these sets to change during the operation of a system. For example, when a computer does not receive any information, either for a long time or when an explicit request is made, from one of the computers in its SC, it may then decide to replace that computer with another in its OC.

3.2.7 INFORMATION MEASUREMENT AND EXCHANGE:

The transfer and the location policies of a load sharing scheme use the system information to make their decisions. In this section we consider the issues of measurement of system information on each computer, and exchange of this information between computers.

The component of a load sharing scheme which measures the system information can be invoked independently on each computer in the following ways:

- **Periodically** every t seconds;
- **Irregularly**, on the request of a load sharing scheme;
- **Combination** of the above two; i.e. periodically as well as when an additional request is made by the load sharing scheme.

The component of the load sharing scheme that exchanges the information of a computer with the computers in its RC could be invoked in the following ways:

- **Periodically**, every T seconds; normally T would be larger than t above;
- **On Request** from another computer to supply the system information;
- **On becoming Idle**, a computer starts informing the others in its RC of its idleness;
- **Combination** of the above; e.g. information is exchanged periodically as well as when requested by another computer.

Although the methods for measuring and exchanging information are similar, it is not essential that the same methods are adopted in both cases. For example, the information on each computer could be measured periodically, but exchanged with other computers only when requested. Furthermore, it is not even necessary to exchange exactly the same information as measured locally. The values exchanged could, for example, be the average of values measured periodically on the local computer since the last exchange.

Besides the features of the system, the choice of methods for resolving the above two issues will also depend on the method of initiating the load sharing scheme. For example, if the load sharing scheme is initiated periodically, then it would make sense to select periodic methods of information measurement and exchange.

Both the information measurement, and information exchange schemes suffer from the problem of information-ageing. This means that by the time the information is actually used by the transfer and the location policies of a load sharing scheme, particularly at a remote computer, the information may already be out-of-date. There is no complete solution to this problem. However, by measuring the information just before it is required, and using weighted averages over a given period rather than instantaneous values, the effects of information-aging can, to some extent, be reduced.

We have not discussed in this section the exact methods of measuring different types of information (e.g. load) mentioned in section 3.2.5, because they would depend on the way other issues have been resolved as well as on the particular system being considered. Nevertheless, it should be mentioned that the extent of improvement in the system performance by load sharing is greatly influenced by the accuracy and speed of these methods. Accurate values would lead to better decisions by transfer and location policies. Fast methods of measurement, on the other hand, would either reduce the overhead of load sharing or, for a given overhead, allow a higher frequency of load measurement, thus reducing the effects of information-ageing.

3.2.8 DESIRABLE FEATURES:

Throughout the discussion of the above issues, we have maintained that their solution depends on the features of the particular system being considered. There are, however, some features that are desirable in any load sharing scheme on any system. In [ALONSO 83] six features of a good load sharing scheme are given:

i) Stability: If an idle computer announces its availability to computers in its RC, then it may get inundated with tasks from all possible sources. Unless catered for, it is also possible that a task may keep *hopping* from one computer to the other, but not getting executed at all. The load sharing scheme should avoid situations that can potentially lead the system into an unstable state. Stability and distributed scheduling algorithms have been discussed in [STANKOVIC 85].

ii) Implementability: The implementation of a load sharing scheme should not require major changes to the existing system. The load sharing scheme should, therefore, not alter the basic attributes of the system.

iii) Cost: The total cost of running a whole load sharing scheme should be as small as possible. The objective of load sharing can then be achieved to a better degree.

iv) Autonomy: The servers in the system should be free to accept or reject requests from the source computers. In case of rejection, the source should be able to *recover*, and either try another server or execute the task locally.

v) Transparency: The users of the system should not be aware that some of their tasks are being executed remotely. This is essential if the user is to be saved from having to learn new system commands to perform load sharing.

vi) Tunability: The load sharing schemes should be tunable to the changing environment. The tuning could either be automatic, or could be done by the system's manager.

The issues mentioned in the earlier sections must be resolved in a way that retains these six desirable features of a load sharing scheme.

3.3 MOS and MAITRE D' REVISITED

In this section we reconsider two load sharing schemes, MOS and Maitre d', that were mentioned in Chapter Two. These two schemes have been chosen because, unlike other schemes that are either entirely theoretical or only partly simulated, these two have been implemented on real systems. Our aim in looking at these two schemes is to discover the way both of them have resolved the issues that were identified in the last section. This has been done in Table 3.2 below.

Note that the two schemes have adopted contrasting solutions for some of the issues. For example, MOS has used periodic initiation, while Maitre d' has employed Task (command) arrival initiation.

3.4 CONCLUSIONS

In this chapter we have looked at the fundamental issues that arise if a load sharing scheme is implemented in a distributed system. For each issue, possible ways of resolving that issue are suggested. In the end two existing load sharing

ISSUES	MOS	MAITRE D'
OBJECTIVE	Improvement of the response time.	Even distribution of load amongst computers
GRANULARITY AND MECHANISM	Process Level Code transfer	Command Level Remote Call
INITIATOR	Source	Source
INITIATION	Periodic	Task Arrival (when command given)
INFORMATION DEPENDENCY	Computer Loads and CPU times already used by processes	History, least recently used available computer is selected. Local load > Threshold
CONNECTIVITY	At each computer a randomly changing subset of OC is maintained.	Fully Connected, each computer considers every other.
LOAD MEASUREMENT AND EXCHANGE	Local measurement every 20 msec; Random exchange every 1 sec.	Server informs all others of its availability. Only local load measured at command issue time.
DESIRABLE FEATURES	Transparent, Stable, Tunable, required changes to existing system.	All features satisfied to an extent.

TABLE 3.2

schemes were compared according to the way they had solved each of the issues discussed earlier. We conclude that the solution to a load sharing issue depends on three factors: the characteristics of the distributed system being considered, the desirable features of a load sharing scheme that are to be retained, and the way some of the other issues have already been resolved.

CHAPTER FOUR

THE NEWCASTLE CONNECTION AND UNIX UNITED SYSTEMS

4.1 INTRODUCTION

The Newcastle Connection is a software subsystem developed at the University of Newcastle upon Tyne. This subsystem has been used in construction of distributed systems which are commonly known as Unix United systems [BROWNBRIDGE 82, RANDELL 84, BLACK 86]. In the next chapter we shall describe the design of a load sharing scheme for one such Unix United system. Therefore, in this chapter we describe the general structure of Unix United systems, and the role of Newcastle Connection in their construction.

In section 4.2 we shall briefly examine the external and internal characteristics of a Unix United system. We then argue in section 4.3 that Unix United systems embody the essential features of Enslow's definition (discussed in Chapter One) of a distributed system. The position of load sharing within the structure of a Unix United system is then discussed in section 4.4. This chapter is concluded in section 4.5. In this and the following chapters we assume that the reader is familiar with the design and terminology of Unix systems. Further details about the Unix system can be found in [BACH 86, DUNSMUIR 85, BROWN 84, FOXLEY 85, BANAHAN 82, RITCHIE 78, RITCHIE 84].

4.2 UNIX UNITED SYSTEMS

A Unix United system is a distributed system composed out of a set of inter-linked Unix systems, each with its own storage and peripheral devices, accredited set of users and system administrators [BROWNBRIDGE 82]. The naming structure for files, devices, commands and directories of each component Unix system are joined together into a single naming structure, in which each Unix system is, to all intents and purposes, just a directory. An example is given in Figure 4.1 of a simple Unix United system consisting of just two machines.

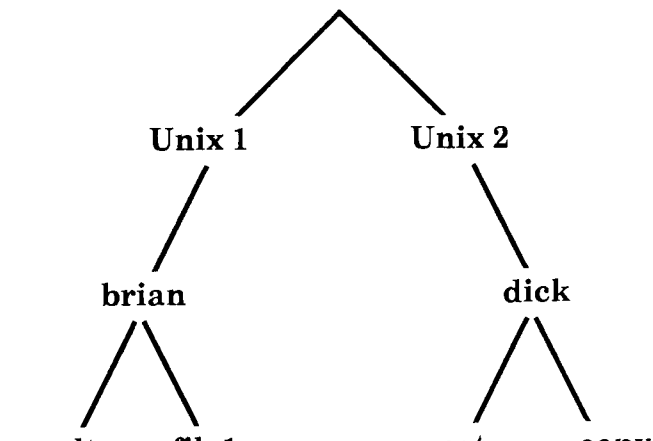


FIGURE 4.1

It must be noted that Unix1 and Unix2 are two completely separate, autonomous Unix systems residing on distinct machines. The tree-structured

nature of the Unix naming scheme makes it easy to create a Unix United tree of which the original systems are subtrees (as shown in Figure 4.1).

In a Unix United system a user can access any file on any machine in the system (assuming the user has the access permission for that file). For example, in the system shown in Figure 4.1, a user working in the directory *dick* on Unix2 can issue the following command (the root directory '/' is on Unix2):

```
cp /../Unix1/brian/result  copy
```

to copy the file called *result* (residing in the directory *brian* on Unix1) into a file called *copy* residing on the local machine Unix2.

Similarly, a user is able to invoke a program on a remote machine and then be able to get the results returned into a file on his machine. For example, if the user working in the directory *brian* of the system shown in Figure 4.1 issues the Unix command (the root directory '/' is at Unix1):

```
/../Unix2/dick/sort  file1  >  result
```

then the program *sort* residing in the directory *dick* on Unix2 will be applied to the file *file1* (on Unix1), and the result will then be put in the file *result* in the directory *brian* on Unix1.

In a Unix United system execution always takes place at the machine where the program resides. Therefore, in the above example the data in file *file1* in directory *brian* on Unix1 will be accessed remotely from the directory *dick* on Unix2 where the program *sort* resides. The execution will take place on Unix2 machine and the results passed back to directory *brian* on Unix1.

In both the examples given above the format of the commands would be similar even if Unix1 and Unix2 were two directories residing on one machine. Thus, a Unix United system hides from its users the presence of more than one machine in the system. This system transparency is achieved by adding to each standard Unix system which is part of the Unix United system, a layer of software called the Newcastle Connection. Figure 4.2 shows the position of the

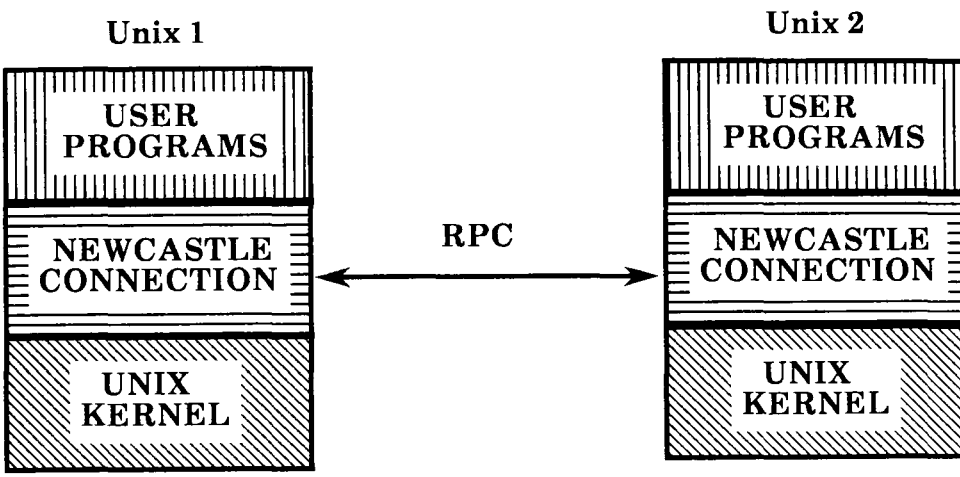


FIGURE 4.2

Newcastle Connection layer for the system in Figure 4.1.

The Newcastle Connection layer hides from the user all the issues concerning network protocols and inter-process communication. The user programs make calls to the Newcastle Connection layer as though the calls are being directed to the kernel, while to the Unix kernel the Newcastle Connection appears as a user program. The Newcastle Connection layer filters out system calls that have to be redirected to the Newcastle Connection layers on remote

machines (and thence to the remote kernel), and accepts system calls that have been directed to it.

To communicate with the other machines in the system, the Newcastle Connection layer makes use of a Remote Procedure Call (RPC) mechanism [SHRIVASTAVA 82]. As shown in Figure 4.2, the logical communication between the machines in the system takes place through their Newcastle Connection layers. However, this communication actually occurs at the hardware level, and for this purpose the kernel includes means for handling low level communication protocols.

4.3 ENSLOW'S CRITERIA AND UNIX UNITED SYSTEMS

In this section we assess Unix United as a distributed system by determining the extent to which it possesses the five features identified by Enslow in his definition of fully distributed systems (discussed in Chapter One).

1 - Multiplicity of resources:

The first Unix United system was based on a set of three PDP 11/23s and two PDP 11/45s all running Unix V7 and connected by a Cambridge Ring. Since these machines are all general purpose physical resources, while the facilities provided by Unix V7 (e.g. file systems, editors, compilers etc.) on each machine can be regarded as multiple logical resources, the first requirement that a distributed system is composed of a multiplicity of general purpose physical and logical resources is satisfied.

The second part of the first feature is that the multiple resources should be dynamically assignable. A user on Unix United can run programs on various remote machines. If the user decides to perform a task in the background then

he need not wait for one machine to finish his task before starting another program on the same machine or a different remote machine. Thus a user on Unix United can dynamically use different resources of the system.

The Newcastle Connection has also been run on ICL's PERQ machines over Ethernet at the University of Newcastle upon Tyne, and elsewhere it has been run between VAX and 68000 machines, thus making Unix United systems heterogeneous in terms of physical resources. However, it is still required that all the machines in a Unix United system be running either Unix (various versions can be supported simultaneously) or Unix look-alike systems. Even though the definition of distributed systems allows machines in the system to have different operating systems, the homogeneity in Unix United does not exclude it from the definition. Therefore it can safely be concluded that the Unix United satisfies the first criterion of the definition.

II- Component Interconnection:

All communication between machines in the Unix United system is performed by the means of a Remote Procedure Call (RPC) protocol. The mechanism for RPC is part of the Newcastle Connection layer. This layer filters out system calls that have to be redirected to the Newcastle Connection layer on a remote machine, and accepts system calls that have been directed to the local machine.

The RPC uses the term *client* for the sender of the request, and *server* for the intended receiver. The important point about this protocol, as far as the definition of distributed systems is concerned, is that the server is quite free to make its own decision, depending on the local situation, as whether to perform the requested service or not. In other words, Unix1 can call Unix2 and vice

versa. Hence the system is symmetrical and there is no master-slave relationship between the system components. It can, therefore, be said that Unix United satisfies the second requirement of the definition of distributed systems.

III - Unity of control:

This criterion requires that a distributed system has a high level operating system or an executive control that defines and supports a unified set of policies in order to achieve the overall objectives of the system. In Unix United it is difficult to pinpoint the existence of such an operating system. The need for a high level operating system is more apparent in heterogeneous distributed systems. For example, if the components of a system are running different operating systems, then it might be necessary for the high level operating system to provide a command interpreter, so that the system's components are compatible and transparent to the user.

In Unix United all the machines run same operating system, therefore there is no need for a special command interpreter. Other policies to achieve, for example, reliability, load sharing, and security are (or can be) part of the Newcastle connection layer. Therefore one could argue that an implicit executive control does exist in Unix United that has been hidden inside the connection layer.

An important point about the requirement of a high level operating system is that it should not be a centralised block of code with strong hierarchical control over the system. Since the connection layer is replicated on each machine in the Unix United system, the implementation of the high level operating system can be viewed as being distributed in the system. The common

goals of the system are thus achieved by mutual cooperation of components rather then being enforced by one particular component.

It can, therefore, be concluded that, as far as the Unix United system consists of machines running the same operating system, the third feature of the definition of distributed systems is satisfied.

IV - System Transparency:

This criterion requires that a distributed system should be totally transparent to the user, and the user should be able to request services by generic names and not be aware of their physical location. In Unix United if a user in the directory *dick* on Unix2 issues the following command:

```
cp /../unix1/user/tma/result filename
```

meaning that the file called *result* on Unix1 in the directory of user *tma* is to be copied into a file called *filename* in the current directory, then the Newcastle Connection will recognise this as a command requiring the use of RPC and make a call to the connection layer on Unix1. So, the details of communication with Unix2 are hidden from the user and he need not be concerned about the protocol used to transfer the contents of file *result.*

It might be argued that the system is not completely transparent to the user because he needs to use *../Unix1/..* in his command. However, when using *../Unix1/..* in his command, as far as the user is concerned the Unix1 is just another directory, similar to other directories which reside on his own machine. Therefore the user's view of Unix2 is that of another directory and not of

another machine. In fact there is no way for the user to tell that Unix1 is a machine because every test made will report that it is a directory.

If the user wants to compile a program in a particular language, and the user's machine does not have a compiler, then the user can give a command for his program to be compiled on a remote machine. The Connection layer recognises the remote request and passes the program to the remote machine that possesses the compiler. The result of the compilation is then passed back to the user as if the compilation had been performed on the local machine.

There is one area where Unix United might not appear to be transparent. If a user has registered with Unix1, then he can log-on to Unix1 and be able to use the facilities of other Unix systems without having to log-on and give the password to each system accessed. But, if Unix1 crashes then that user would be unable to use any of the facilities of the system. This inability might suggest that Unix-United is not fully distributed. This is not the case for two reasons. Firstly, the definition does not require that the user of one system be able to log-on to another system. Secondly, this facility could be provided by making each Unix system have a copy of user identifiers and passwords for all the other systems.

V - Cooperative Autonomy:

This criterion requires that the logical and physical components in a distributed system should interact in a manner described as *cooperative autonomy*. In Unix United the servers on a particular machine can decide to deny the services to a client on another machine. Thus mutual cooperation between the client and the server, rather than master-slave arrangement, is used.

In Unix United each machine runs a full Unix system and is capable of providing all the facilities of Unix without being connected to the network. Similarly if every machine in the network has been connected to all the others, then the removal of one machine will only degrade the performance of the system and not shut down the whole system. Therefore the components in the system are autonomous and the final requirement of the definition is satisfied.

It has been shown that the Unix United satisfies all the features required by Enslow's definition. This does not mean that all the objectives of a distributed system are achieved by the Unix United systems. Extra layers can, however, be added to achieve these objectives. For example, if higher performance is required then a load sharing scheme can be implemented on a Unix United system. Therefore, it can be concluded that Unix United provides a distributed framework on which layers can be added to achieve the objectives of a distributed system.

4.4 LOAD SHARING IN UNIX UNITED SYSTEMS

The complexity of implementing large and sophisticated computing systems can be reduced significantly by ensuring that the system is constructed out of a well-chosen set of largely independent components which interact in well-understood ways.

In [RANDELL 85] Unix United systems have been presented as systems that have been developed using this philosophy and as a system structuring technique that distinguishes the functionality of the system from its desirable attributes. These attributes are then provided by separate components. For example, the functionality of Unix United systems is the provision of a Unix

environment. The desirable attribute of distribution is provided by the component called the *Newcastle Connection* as shown in Figure 4.2. Similarly, additional reliability has been included in one of the Unix United prototypes by adding a transparent software component (the Triple Modular Redundancy, TMR layer) on top of Newcastle Connection layers as shown in Figure 4.3 [RANDELL 85].

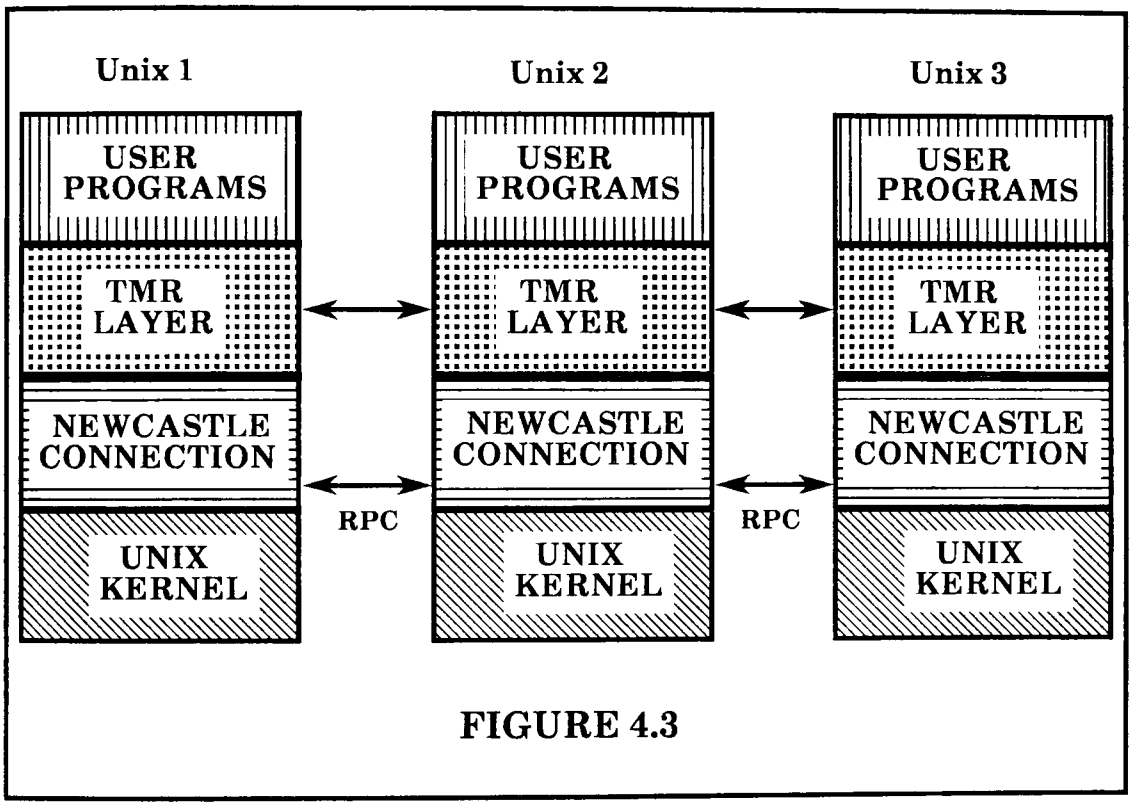


FIGURE 4.3

Similarly, the presence of the attribute of distributedness in Unix United systems makes it possible to increase their performance by load sharing. In the context of Unix United systems load sharing can, therefore, be regarded as a desirable attribute that is independent of the functionality of the system. It is proposed, therefore, that this attribute, like the attributes of distributedness and reliability, should be added to Unix United systems by another independent

component. This component will be implemented by adding to each Unix machine in the system a software layer called *Load Sharing* layer.

The load sharing layer on each machine will perform two main functions:

(1) Collect system information from the other machines;

(2) Use this information to divide the computational load in the system to achieve certain objectives.

In order to perform both these functions, the load sharing layer on a particular machine will need to communicate with the other machines in the system. Since the Newcastle Connection layer provides mechanisms for this communication, and also mechanisms for initiating computation on a remote machine, it seems natural to place the load sharing layer on top of the Newcastle Connection layer. If highly reliable load sharing (i.e. load sharing that can survive hardware errors and crashes) is required then the load sharing layer should be placed above the TMR layer (Figure 4.4).

It must be pointed out, however, that placing the load sharing layer above the Newcastle Connection layer is not the only way to incorporate load sharing in Unix United systems. The load sharing layer could, for example, be placed below the Newcastle Connection layer in which case it will need to interface directly with the RPC mechanisms. Alternatively, load sharing can be made part of the Newcastle Connection layer, or indeed it could be implemented inside the Unix kernel (these possibilities are further considered in chapter Six). In these cases substantial changes would have to be made to the Newcastle Connection layer, or the Unix kernel. Therefore, we believe that placing the load sharing layer above the Newcastle Connection layer provides the simplest way of incorporating load sharing in Unix United systems.

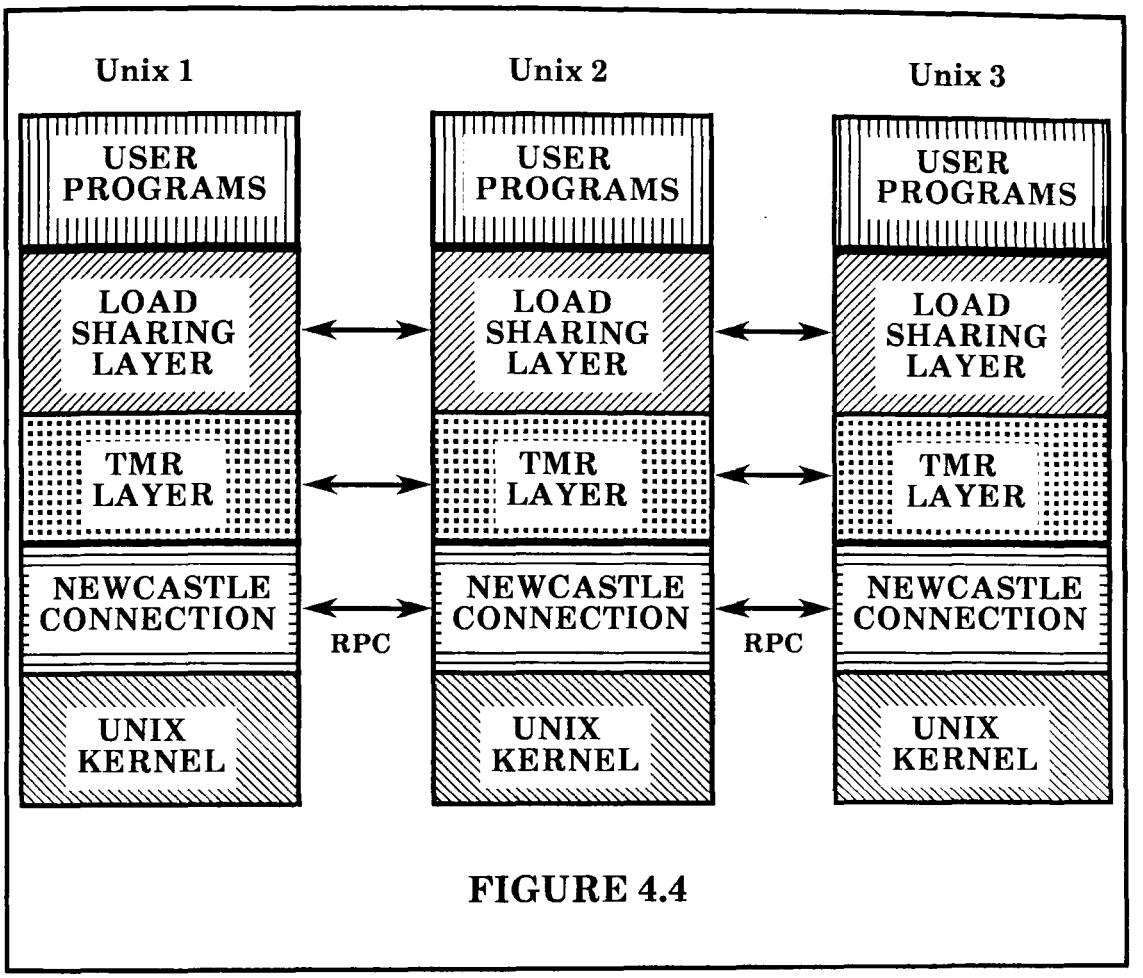


FIGURE 4.4

4.5 CONCLUSIONS

In this chapter we have looked at the general structure of Unix United systems, and the role of the Newcastle Connection in these systems. We have argued that the Unix United systems provide a framework on which the objectives of distributed systems can be achieved by implementing additional layers. Finally, we considered possible ways of incorporating load sharing in the existing structure of Unix United systems.

CHAPTER FIVE

A LOAD SHARING SCHEME FOR A UNIX UNITED SYSTEM

5.1 INTRODUCTION

In this chapter we shall describe the design of a load sharing scheme for an existing Unix United system by resolving the issues identified in Chapter Three. In section 5.2 we describe the Unix United system to be considered for load sharing. The load sharing issues are then resolved for this Unix United system in section 5.3, and this chapter is concluded in section 5.4.

5.2 THE PERQS' UNIX UNITED SYSTEM

The Unix United system considered for load sharing consists of four ICL's Perq1 (named Tyne, Tweed, Catcleugh and Kielder) single-user, multi-process computers [ICL 84]. These computers are linked together by a high speed (10 Mb/sec) Ethernet [METCALFE 85] as shown in Figure 5.1.

All the machines run same version of a Unix look-alike operating system called PNX. Therefore, this Unix United system is fully homogeneous (identical machines running the same operating system). Each computer has its own internal disk storing its file system, and is capable of running on its own. We shall refer to this Unix United system of four Perqs as the *Perq system.*

The Perq system was set up specifically for carrying out load sharing experiments, and therefore it was not possible to see how it would have been

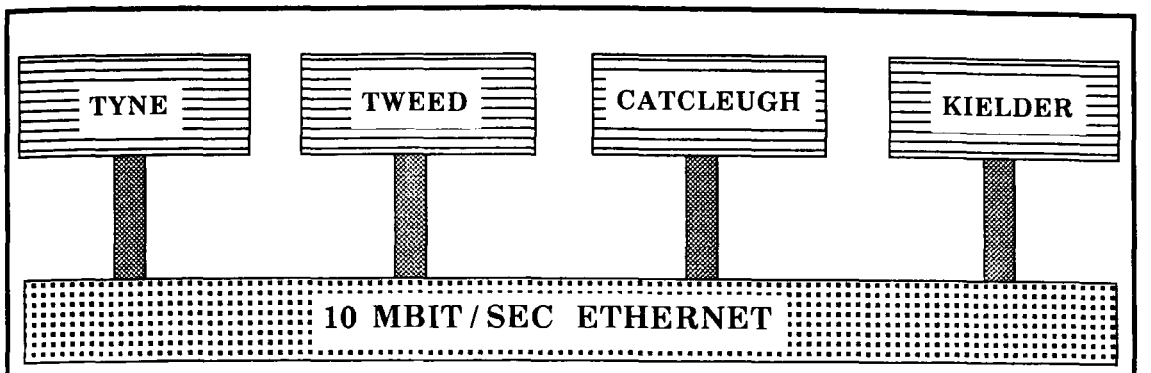


FIGURE 5.1

used had it been available for general use. However, it is reasonable to assume that the computers in the Perq system would have been used in very much the same way as the other single-user, multiprocess Unix computers in the department. The observation of the usage of these other computers revealed two potentially significant points. Firstly, it was very rare to find all machines being used at the same time. In fact, most of the time more than half the machines were idle. Secondly, it was common for the users of these machines to have several programs running simultaneously. This was encouraged by the following factors:

- the ability to run programs in the *background*;

- the ability to create *pipes* between different programs where the output generated by one program is used as an input for another;

- and most importantly, the ability to create *windows* that allows a user to split a VDU screen into several independent virtual screens. The user is able to start a program in one window, and then move into next window to start another independent program.

These three facilities are also available on all the computers in the Perq system.

The above two observations indicated that it would be common to find the Perq system in situations where one computer is running more than one program, while at least one other computer is not being used at all. The load sharing scheme we have designed for the Perq system is intended to exploit this situation.

5.3 RESOLVING THE LOAD SHARING ISSUES

In the following subsections we describe how the fundamental load sharing issues, identified in Chapter Three, were resolved by keeping in mind the characteristics and the expected usage of the Perq system.

5.3.1 OBJECTIVE:

The most important criterion of the system performance for the user of the Perq system is the time taken to complete his task(s). We shall refer to this time as the *Completion Time*. Since there can be more than one user of the system, and reducing the completion time for one user may increase the completion time for another, the objective of the load sharing scheme for the Perq system was chosen to be the reduction of the sum of the completion times for all the users of the system. To be more precise about this objective, we define the following terms:

O_{st}, The observation start time: This is, effectively, the time on the clock on the wall, and not on any particular computer in the Perq system. At this time the system is not executing any task.

O_{ft}, The observation finish time: Again, this is the time on the clock on the wall. At this time the system has finished executing all the tasks of all the users.

OP, The observation period: $OP = O_{ft} - O_{st}$.

N, The total number of users using the system during OP. The first user to login is the user number one, and the last user to login is the user number N.

T_{is} : The time, on the computer where user i is logged on, when the system starts executing the tasks of user i.

T_{if} : The time, on the computer where user i is logged on, when the system finishes executing the tasks of user i.

C_i : The completion time for user i, $C_i = T_{if} - T_{is}$.

TC : Total completion Time,

$$TC = \sum_{i=1}^{N} C_i$$

The objective of the load sharing scheme for the Perq system was to reduce TC. The effectiveness of the load sharing scheme can be determined by comparing the values of TC obtained with and without the use of the load sharing scheme. This comparison will, of course, be meaningful only when in both situations the system is required to perform identical tasks for the same users in same conditions.

5.3.2 GRANULARITY AND MECHANISM:

Most of the computational load on a computer in the Perq system gets generated by the users' requests for program execution. The Perq system provides, through the Newcastle Connection, a mechanism for transferring a

program from one computer to the other, and also to execute a program residing on the remote computer. Therefore, it was appropriate to choose a program as the granularity of the load sharing scheme for the Perq system.

Recall that in Unix United systems a program is always executed on the computer where it resides, irrespective of the computer which requested its execution (Chapter Four). Hence, for the purpose of load sharing, a heavily used computer will first transfer a program, originally intended to be executed locally, to a free machine in the system, and then use the existing mechanism for its remote execution. However, if the program being considered for remote execution is a standard Unix utility, then it will be available on all the computers, thus avoiding the need for program transfer. The heavily loaded machine would simply use the existing mechanism to execute the program already resident on the remote machine.

Therefore, it was decided that initially the load sharing scheme for the Perq system would only consider transferring the execution of standard Unix utility programs. Besides avoiding the overhead of transferring the programs to the remote computer, this approach makes this load sharing scheme applicable to systems where the programs on one computer cannot be executed on the other, but the computers run the same operating systems and utilities.

One would expect a CPU intensive program to be better suited for remote execution than an I/O intensive program that needs data resident on the local machine. The load sharing scheme therefore had to be selective in which programs to consider for remote execution. After implementing the load sharing scheme, we experimented with different types of programs to discover which

ones are best suited for remote execution (This is covered in more detail in Chapter Seven).

The essential point at this stage is that the granularity for the load sharing scheme for the Perq system was chosen to be the standard Unix utility programs. The remote program execution is carried out by using the existing mechanism provided by the Newcastle Connection. This mechanism is based on the *remote call* method for remote process execution described in section 3.2.2.

In order to understand how this mechanism is used by the load sharing scheme, let us first consider how a computer in the Perq system (and in other Unix United systems) carries out a command typed in by the user. At login time, a program called *shell* is started by the computer for the user. The user interacts with the system through shell. The shell is, therefore, responsible for reading, interpreting, and carrying out the commands typed in by the user. For example, if the user types in the command *cat file1* then, in simple terms, the shell performs the following functions:

I - It reads the command *cat file1*.

II - It searches for a program called *cat* in some predefined directories. If it does not find the program *cat*, it gives an error message and waits for the user to type in the next command.

III - If the program *cat* is found, then the path name of the program is generated by shell. Thus, if the program *cat* is found in the directory */bin*, then the pathname */bin/cat* will be generated. The name of the program (*cat*) is in the first element of an array called *argv*. The parameters to the command, like *file1* in this case, are then put in the successive elements of the array argv.

IV - The shell then issues the system call *fork*. This system call makes an identical copy of shell program and its data. This *child* shell starts executing simultaneously with, and independently of, the original or the *parent* shell. Normally, after starting the child shell, the parent shell would wait for the child shell to finish before prompting the user for the next command.

V - Soon after its birth, the child shell issues the system call *exec*. The path name of the program, and a pointer to the array argv are passed as parameters to *exec*. The system call *exec* overwrites the child shell with the program identified by the pathname. In our example, the child shell will be replaced by the *cat* program residing in the directory */bin*.
Note that if the pathname referred to a remote program (e.g */../tyne/bin/cat*), then the exec system call would get trapped by the Newcastle Connection which will arrange for it to execute remotely on Tyne.

VI- When the cat program terminates, the parent shell is informed so that it may prompt the user for the next command.

To achieve remote execution the load sharing scheme changes the pathname before it is passed to the system call *exec* as a parameter. In our last example the pathname was */bin/cat*. If according to the load sharing scheme the program cat should be executed remotely on the machine catcleugh then the pathname will be changed to */../catcleugh/bin/cat*. The Newcastle Connection will recognise this *exec* as the one requiring remote execution, and take the necessary steps. The system call *exec* can be issued by other system and user programs to start the execution of another program. Therefore a method of detecting the occurence of *exec* system calls is required so that the load sharing scheme may be invoked. The method used by our scheme is described in Chapter Six.

5.3.3 INITIATOR:

The mechanism for remote program execution in the Perq system allows a computer to invoke the execution of a program on another computer. However, this mechanism cannot be used to transfer a program, that has already started executing, to complete its execution on another computer. Therefore the decision regarding the transfer of a program for remote execution has to be taken before starting its execution. In other words, this decision has to be made when the request is made for the execution of a program.

Since the running of a program puts load on a computer on which it executes, and more up-to-date load information is available locally, it was appropriate that the computer on which the program would have executed (in the absence of any load sharing) should consider whether to execute the program locally or remotely. Thus, if the command *cat myfile* is issued on Tyne then the load sharing scheme will be invoked on Tyne. However, if the command */../tweed/bin/cat myfile* is issued, still from tyne, then the load sharing scheme on Tweed will decide whether to execute the program locally (on Tweed) or select another computer in the system for its execution.

Therefore, the load sharing scheme for the Perq system is source initiated. The word *source* in this context means the computer where the program would have executed, not necessarily where the request for its execution is made.

5.3.4 INITIATION:

In the previous section we explained that in the Perq system the decision to transfer a program has to be made when a request is made for the execution of a

program. Therefore, the load sharing scheme will be initiated at the task-arrival time (see section 3.2.4).

The task-arrival initiation can result in a very high overhead. However, in the Perq system this is unlikely to happen for two reasons. First, the rate at which commands arrive on a single user machine is fairly low. Hence the load sharing scheme will not be invoked too often. Secondly, the transfer policy of the load sharing scheme is designed such that the computationally expensive parts of the scheme are not performed until it is clear that the program being considered is suitable for remote execution (section 5.3.5). Thus the expensive parts of the load sharing scheme will be executed even more infrequently.

5.3.5 INFORMATION DEPENDENCY:

In the Perq system a computer can be in one of the following states:

1- **Unavailable (U):** In this state the computer is not operating, i.e. it is switched off.

2- **Free (F):** In this state the computer is operating but no user is logged on it;

3- **Occupied (O):** In this state a computer is operating, and a user is logged on it. An Occupied computer is said to be *busy* if at the time of observation it has more than a given number of processes waiting for the CPU, or a fast event such as disk I/O;

4- **Hosting (H):** In this state a free computer is executing task(s) for another computer. Note that only a previously free computer can be hosting. An occupied computer executing tasks for another computer remains occupied.

Each computer in the system maintains a vector which stores the states of the computers. How these vectors are initialised and updated is described in section 5.3.7. For now we assume that these vectors have been initialised to some particular values. For example consider the Perq system shown in Figure 5.2.

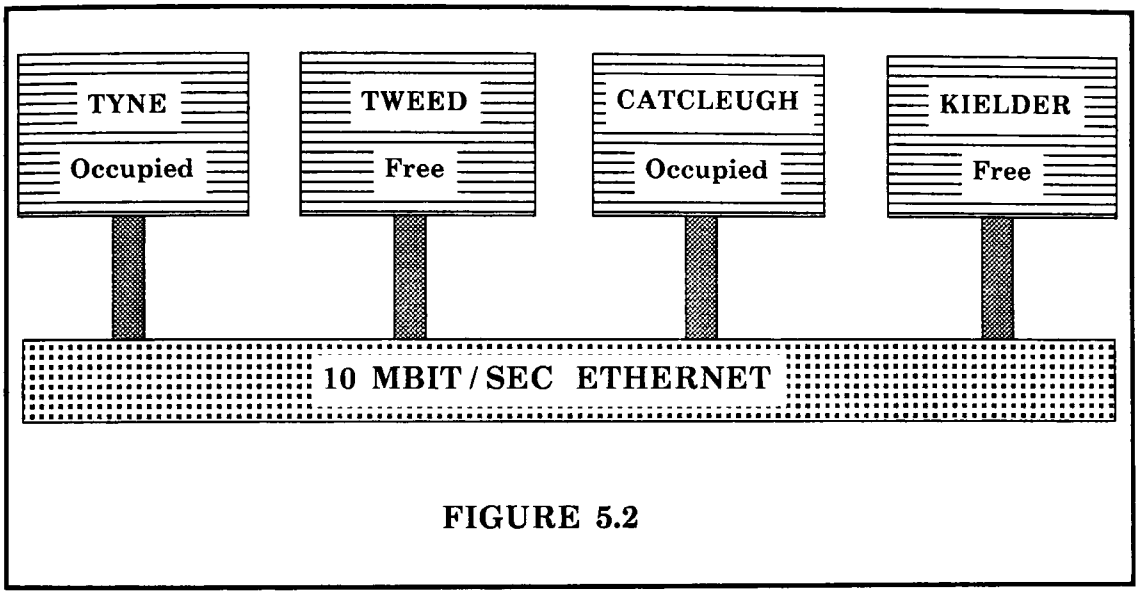


FIGURE 5.2

Tyne and Catcleugh are in the state occupied, while Tweed and Kielder are free. The corresponding state vectors are shown in Figure 5.3. Note that the state vectors on Tyne and Catcleugh are fully up to date and show the current state of the system. The state vectors on Tweed and Kielder are not up to date. The load sharing scheme is designed keeping in mind that it would not be possible for all the state vectors to represent the current system state.

The zeroth component of the state vector always stores the local state, hence avoiding the need for each computer to know its offset in the state vector. The order for storing other computer states is different in each vector. Thus, the

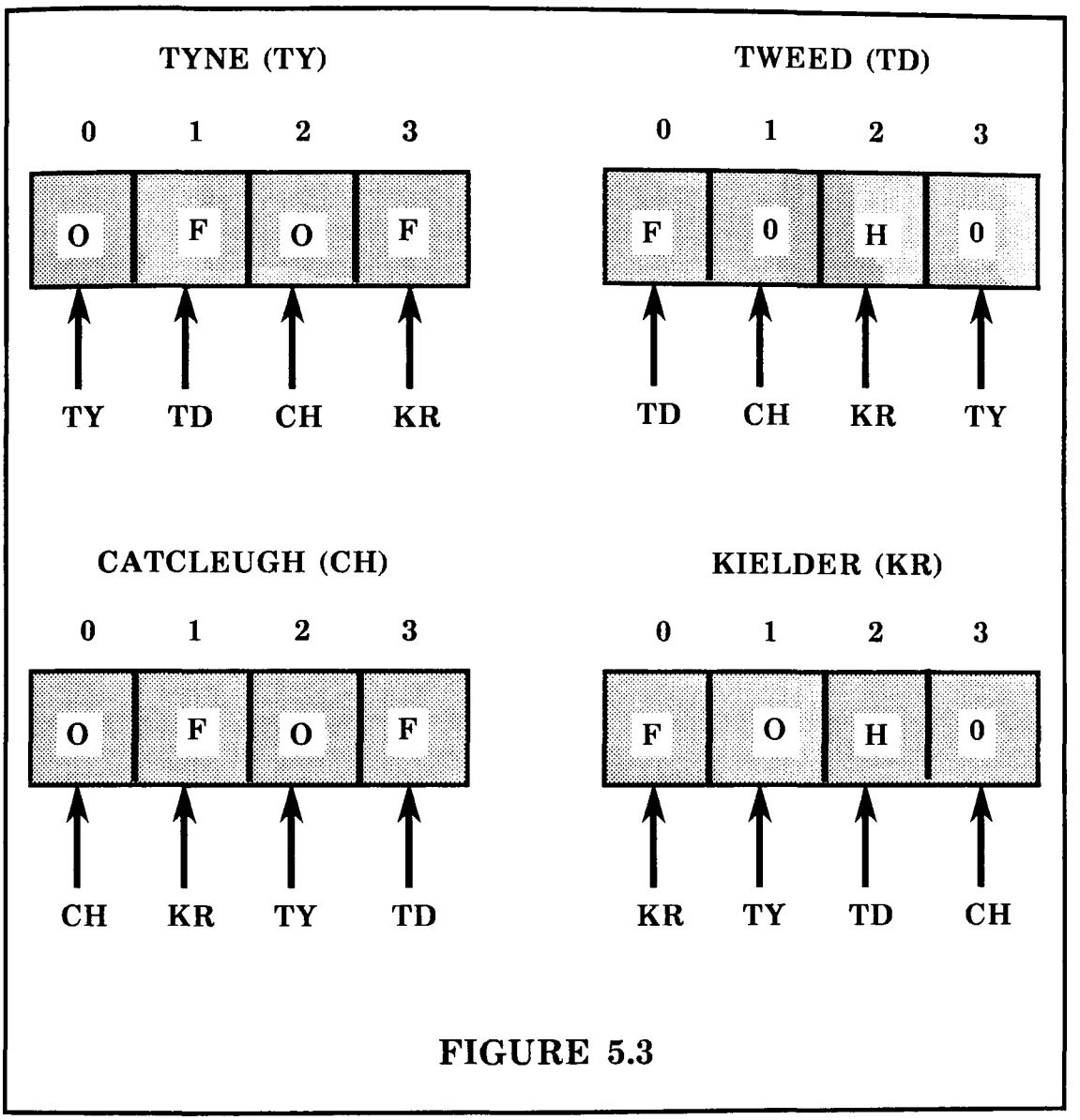


FIGURE 5.3

second component of the state vector on Tyne holds the state of Tweed, but on Catcleugh it holds the state of Kielder. This arrangement can, in some system states, avoid the same free computer being selected by more than one occupied computer to execute their program (see the location policy later in this section).

The state vectors are used by the transfer and the location policies of the load sharing scheme. We describe these two policies separately below.

TRANSFER POLICY

In the Perq system the commands on an occupied computer can either require local program execution, or a remote program execution. In the absence of any load sharing scheme the program is executed on the machine for which the request has been made. For example, on Tyne the command

```
cat myfile
```

would result in the program *cat* being executed locally on Tyne. On the other hand the following command (again issued on Tyne)

```
/../kielder/bin/cat myfile
```

would result in the program being executed remotely on kielder. For the local requests the transfer policy of the load sharing scheme for the Perq system has to decide whether to execute the program locally, or to select a computer by using the location policy and execute the program there.

The remote requests are passed on to the requested machine which then decides whether to execute the program or send it to another computer for execution. This arrangement requires that thare are two transfer policies on each computer. The L-transfer policy considers the requests made locally (either for local or remote execution) while the R-transfer policy considers the requests received directly from other computers (not as a result of load sharing).

The algorithm implementing the L-transfer policy is shown below using a pseudo-pascal notation:

```
IF ( remote request ) THEN
        invoke R-transfer policy on the remote computer with this request.
ELSE IF (program not suitable for load sharing)
            OR ( local machine not busy) THEN
        execute it locally.
ELSE BEGIN
        Try to locate a free computer;
        IF (could not find a free computer) THEN execute locally.
        ELSE BEGIN
            make request for remote execution;
            wait for it to finish;
            IF (erronous finish) THEN re-execute locally;
         END;
END;
```

First of all the L-transfer policy decides whether the request is for local or remote execution. If it is for a remote computer then the R-transfer policy on that computer is invoked with the original request as a parameter. Otherwise, if the program is not suitable for loadsharing, or the local computer is not busy then the request is carried out locally. On the other hand, if the program is suitable for load sharing and the local computer is found to be busy then the location policy (described in the next section) of the load sharing scheme is invoked.

If the location policy cannot find a free computer then the request is carried out locally. However, if the location policy finds a free computer then a request is made for remote execution. It is important to note that this request will not be further considered by the transfer policy on the remote computer. This is essential to avoid a situation where a request keeps circulating among

computers without ever getting executed. If the remote execution is not successfull then the program is re-excuted locally.

The R-transfer policy is very similar to the L-transfer policy. The only difference is that the R-transfer policy does not check if the request is local or remote. Instead it starts off by determining the programs suitability for load sharing, and then performs same checks as the L-transfer policy.

LOCATION POLICY

The location policy for the load sharing scheme for Perq system works in two phases. First, it searches the local state vector for a computer in free state. This search starts from the component two of the state vector and the first computer in free state is selected. So, in the example of system state shown in Figures 5.2 and 5.3, the location policy executing on Tyne would select Tweed, while the location policy executing on Catcleugh would select Kielder. This illustrates that in some system states (as indicated in the state vectors) this method of search would prevent a same computer being selected by two busy computers.

If the search for a free computer fails then a similar search is carried out for a hosting computer, and the first hosting computer is selected. If either free or a hosting computer is found, then the location policy enters the second phase; otherwise the location policy returns the local computer as its choice for executing the program.

The second phase of the location policy makes a direct request to the selected computer to find its current state. If the current state of the selected computer is found to be free then the selected computer is returned to the transfer policy as being chosen for executing the program. On the other hand, if the current state

of the selected computer is found not to be free then its state in the local state vector is updated to its current value. Phase one of the algorithm is then repeated to select preferably free or a hosting computer; the phase two is repeated for the newly selected computer. This process continues until a remote free computer is found, or all the free or hosting computers in the local state vector have been investigated, in which case the local computer is returned to the transfer policy as being chosen to execute the program.

In the absence of any free computers in the local state vector, the hosting computer is selected for further investigation because in the Perq system the hosting state is temporary. Therefore, one expects that a computer in the hosting state is likely to have gone back to the free state (In section 5.3.7 we will find that the other computers are not informed of the state transition from hosting to free). Note, however, that a remote computer is chosen to execute a program only when its **current** state is found to be free, irrespective of whether it was shown to be free or hosting in the local state vector.

The decision to investigate another free or hosting computer after the current state of the earlier selection is found not to be free, is suitable in the Perq system because of the small number of computers involved and the high probability of finding a free computer. Besides, an unsuccessful investigation is not entirely waste of time because as a result the local computer becomes aware of the latest state of the selected computer.

In Chapter Three (section 3.2.5) we identified different types of information that may be used by a load sharing scheme. Our load sharing scheme for the Perq system implicitly uses the following types of information:

- **Processor Availability:** If the state of the computer is Unavailable, then that computer does not participate in the load sharing scheme.

- **Processor Functionality:** By considering only suitable programs for load sharing we are ensuring that the selected computers will be able to execute the program.

- **Current Load:** We consider whether the local computer is busy by observing its current load. Furthermore, the states of the computers effectively reflect their loads.

We decided to ignore the communications costs and the processor speeds in our load sharing scheme for two main reasons. First, we wanted to keep the initial load sharing scheme as simple as possible. Secondly, these two types of information depend on the type of program being executed and therefore not easy to measure.

Finally, since our load sharing scheme makes use of system information that it knows to vary during the system operation, we can classify it as being dynamic.

5.3.6 COMPUTER CONNECTIVITY:

The Perq system consists of four computers, which is a relatively small number. Therefore, the Send Connectivity (SC) and the Receive Connectivity (RC) of all the computers is the Other Computers (OC) participating in the load sharing scheme. Another reason for this decision is that the method of

information measurement and exchange (section 5.3.7) is such that all the other computers are informed of a very few changes in the state of a computer.

5.3.7 INFORMATION MEASUREMENT AND EXCHANGE:

In section 5.3.5 we identified the four states (U, O, F, H) of a computer in the Perq system. The information measurement and exchange in our load sharing scheme is therefore concerned with detecting, on each computer, the transitions from one state to another, updating the local state vector, and then if necessary informing the other remote computers.

A state transition on a computer in the Perq system is caused by an event, for example when a computer is switched ON and starts operating its state changes from **U** to **F**. If a user now logs on this computer (another event), its state will change from **F** to **O**. Note that a computer cannot get into state **O** straight from state **U** and therefore the transition **O** to **U** is invalid.

We list below all the possible state transitions and for each indicate, with reasons, whether it is valid or invalid in the Perq system. For each valid transition we consider the event(s) that may cause it, and decide whether to inform the local and remote computers about this change in the state. The decision to inform the remote computers must be made bearing in mind that the exchange of information puts an extra load on the sender as well as on the receiver computer. Hence, our aim is to avoid unnecessary information exchange.

1- **U to F** (valid): This state transition takes place when a switched OFF computer is switched ON. In this case it is necessary to update the first component of the local state vector, and to inform every other computer because

unless other computers know that a computer has become free they do not consider it for execution of their programs. The current states of the computers that can be contacted at this stage are obtained and the local state vector updated. The computers that cannot be contacted are assumed to be unavailable.

2- U to U (invalid): A switched OFF computer cannot be switched off again!

3- U to O (invalid): A switched OFF computer must become free before it can get into any other state.

4- U to H (invalid): A switched OFF computer cannot start hosting.

5- F to H (valid): This state transition is made when a free computer starts executing a task of another computer. In this case only the local state vector is updated to show that the local machine has started hosting. There is no need to inform any other computer of this change because an occupied computer that may mistakenly believe it to be free will be updated when its locate policy tries to confirm the current state.

6- F to O (valid): This transition takes place when a user logs on a free computer. In this case it is necessary to change the first component of the local state vector to occupied. Again, there is no need to inform any other computer of this change because an occupied computer that may mistakenly believe it to be free will be updated when its locate policy tries to confirm the current state.

7- F to F (invalid): No event can cause this transition.

8- **F to U** (valid): This transition is made when a free computer is switched OFF. In this case there is no need to inform any other computer about this change, because they will all discover its unavailability when they try to contact it.

9- **H to H** (valid): This transition is made when an already hosting computer receives another task from a remote computer. In this transition no action need be taken since the first component of the local state vector is already set to hosting.

10- **H to O** (valid): This transition takes place when a user logs in on an already hosting computer. The first component of the local state vector is changed to occupied. There is no need to inform any other computer of this change because an occupied computer that may mistakenly believe it to be hosting will be updated when its locate policy tries to confirm its current state.

11- **H to U** (invalid): We are assuming that a computer can only be switched OFF when it is in the free state.

12- **H to F** (valid): This transition takes place when a hosting computer finishes all the tasks of remote computers. The first component of the local state vector is updated to show a free state. There is no need to inform any other computer of this change because an occupied computer that may mistakenly believe it to be hosting can still consider it for load sharing and find its current state.

13- **O to H** (valid): This transition occurs when a user logs off from a computer that is still executing some task(s) for the remote computer(s). The first component of the local state vector is updated to show the hosting state, and all

other computers informed of this transition so that they may, in future, consider this computer as a candidate to execute their tasks.

14- O to O (valid): A user on a computer can login as another user without necessarily logging off first. The system remains in state **O**, therefore there is no need to inform anyone of this transition.

15- O to U (invalid): We are assuming that a computer can only be switched OFF when it is in free state **F**.

16- O to F (valid): This transition is caused when a user logs off from an occupied, non-hosting, computer. The first component of the local state vector is updated to show the free state. All other computers are informed of this transition so that they may, in future, consider this computer as a candidate to execute their tasks.

The above state transitions, and the actions that need be taken are summarised in Table 5.1. Note that in all the valid transitions mentioned above, the only times all the other computers are informed is when there is a transition into a free or hosting state from either being unavailable or occupied state. This global exchange is essential because unless the remote computers know that a particular computer is free or hosting, their location policy will not select it for further investigation about executing the tasks.

It may be argued that it is only necessary to exchange this information with the occupied computers because they are the only ones that can possibly send any tasks. However, to be able to inform just the occupied computers, a computer needs to know which computers are occupied; thus a global exchange of information is required whenever a user logs on any computer. This will be an

COMPUTERS TO BE INFORMED		CURRENT STATE			
		F	H	O	U
NEXT STATE	F	INVALID	LOCAL	LOCAL AND OTHERS	LOCAL AND OTHERS
	H	LOCAL	NONE	LOCAL AND OTHERS	INVALID
	O	LOCAL	LOCAL	NONE	INVALID
	U	NONE	INVALID	INVALID	INVALID

TABLE 5.1

absurd solution where global information exchange is made to avoid a global exchange!

Since exchanging information with the non-occupied computers will not put any load on the occupied computers themselves (ignoring the effects it may have on the availability of the Ethernet) the overhead of the global exchange will not

reduce the performance of the occupied computers any more than if the exchange was only with the occupied computers. Furthermore, the event of a computer becoming free is not too common in Unix United systems (particularly those with only four computers!), so we would expect that there will not be too many instances when a global exchange of information is necessary.

One can imagine more sophisticated schemes for information exchange in the Perq system. For example, each computer could keep a record of the last state information it supplied to a particular computer, which need not be same as its present state. Upon a state transition it could then inform only those computers that have some grossly wrong idea of its state. For a Perq system with a large number of computers, such a scheme could reduce information exchange with remote computers, albeit at the expense of more local processing. We have, however, preferred our scheme for its simplicity and low overhead.

Comparing our information measurement and exchange policy with the possibilities mentioned in Chapter Three (3.2.7), we find that our information measurement is irregular. The information exchange policy is irregular as far as the potential receiver of the tasks is concerned; from the point of view of the sender of the tasks, this information is supplied on request.

5.3.8 DESIRABLE FEATURES:

The desirable features of our load sharing scheme are discussed in Chapter Eight after we have described its implementation in Chapter Six and discussed the results of experiments in Chapter Seven.

5.4 CONCLUSIONS

In this chapter we have developed a load sharing scheme for the Perq system. In the design of a load sharing scheme many initial decisions can be rather subjective. We have, however, tried to be as objective as possible about our decisions and based them on the structure and the expected use of the system being considered. The only way to be sure that one has made proper decisions, or to discover the wrong decisions, is to implement and evaluate the scheme. Such an implementation is described in the next chapter.

CHAPTER SIX

IMPLEMENTATION

6.1 INTRODUCTION

In this chapter we shall describe how the load sharing scheme, developed in Chapter Five, was implemented on the Perq system. During this implementation following major problems had to be resolved :

- Trapping *execs* and establishing any given programs suitability for load sharing;

- Establishing whether a local machine is busy;

- Representing the state vectors;

- Detecting state transitions;

In sections 6.2 to 6.5 we consider possible solutions for each problem, and conclude this chapter in section 6.6.

6.2 TRAPPING *EXEC* AND DETERMINING PROGRAM'S SUITABILITY

The load sharing scheme for the Perq system requires that the transfer policy is invoked just before a program starts executing. As explained in section 5.3.2 all new programs are started by the *exec* system call. Therefore, the transfer policy had to be invoked just before any *exec* system call was issued. Thus, we needed to detect or trap the occurence of *exec* calls. After trapping an *exec* call, the transfer policy establishes whether the requested program is

suitable for load sharing. We considered the following four ways of trapping *exec* system calls and establishing the suitability of requested programs for load sharing:

1- From *Shell*:

As explained in section 5.3.2, on Unix systems a program called *shell* is responsible for running requested programs by issuing the *exec* system call. By inserting code to invoke the transfer policy before every occurence of *exec* system call in the *shell* program, one could invoke the load sharing scheme prior to execution of programs requested by *shell*. The transfer policy could then find out whether the request is for the local or a remote computer. If it is remote then the R-transfer policy on the remote computer is invoked. If the request is for local execution then the name of the requested program can be compared with the names of programs suitable for load sharing. This list of suitable programs could either be built in the code of the transfer policy or could be read from an external file every time it is invoked.

Since each user of a Unix system can arrange to have a personal *shell* program the above technique for trapping *execs* would have allowed users to select either a load sharing mode (by running the modified *shell*) or a normal mode. Besides requiring changes to a standard Unix system program (*shell*) this technique had a disadvantage of failing to trap *exec* calls made from within other user and system programs.

2- Changing *exec*:

The code of the *exec* system call could itself be extended by adding at its front the code for the transfer policy. This way the calls to *exec* from all the system and user programs would first invoke the transfer policy. The transfer policy

could then determine the computer to be used for execution and the suitability of program in the same way as described above for the shell method.

Since all the users use same *exec*, with this technique everyone will be forced to work in a load sharing mode. It must be pointed out, however, that *exec*, like other system calls, is part of the Unix Kernel. Hence, changing it was not only difficult but also undesirable since we wish to implement load sharing scheme as a layer on top of Newcastle Connection, and not as an integrated part of the Unix system itself.

3- Changing *exec* at an entry point:

Like Unix itself the load sharing scheme was implemented using a system's programming language called C [KERNIGHAN 78, KELLEY 84]. However, C cannot make a direct Unix system call (which is a TRAP instruction into the Unix system Kernel). Instead, the C programmer calls an external routine which makes the system call on his behalf. One way of trapping the *exec* calls was to make a copy of the external routine responsible for making *exec* system calls, and rename the new copy. The original routine was then modified so that it first executed the transfer policy and then called the renamed copy of the original routine. The transfer policy could then establish the computer to execute the program and its suitability for load sharing in the same way as described for the *shell* method.

This method is used by the Newcastle Connection to trap all system calls [STROUD 83]. A disadvantage of this method is that all the programs that needed to make use of the load sharing scheme have to be recompiled to link them to the new versions of the system call routines.

4- Linking suitable programs:

A file on a Unix system can have more than one name. This is achieved by creating a link between the original file and the new name. This facility could be used to trap the *execs* for the execution of suitable programs. Each suitable program is saved under a new name. The original name is then linked to a program that first executes the transfer policy and then executes the saved program. In this method there is no need to determine whether the request is for local or remote execution because if the requested program is suitable for load sharing then the transfer policy will, by default, get executed on the machine where the program resides.

The suitability of the requested program for load sharing does not have to be established since only the suitable programs would be linked to the program that first invokes the transfer policy. Therefore, the unsuitable programs will not be delayed unnecessarily by the transfer policy. However, this method requires two invocations of system call *exec*. The first *exec* is used to start the program that invokes the transfer policy; this program in turn issues the second *exec* to execute the requested program.

Among the four methods of trapping *execs* described above we decided to implement the last method of relinking the suitable programs. It was selected because of its simplicity, ease of implementation and the features mentioned above. We expect that the additional *exec* call required by this method is compensated by the absence of a need to find out either the place of execution or the suitability of the requested program for load sharing.

6.3 ESTABLISHING WHETHER A LOCAL MACHINE IS BUSY

The transfer policy of the load sharing scheme for the Perq system considers a program for remote execution only if the local machine is found to be busy. Being 'busy' in in this context means that the current load of the machine exceeds a predefined threshold value. Therefore, the problem was how to measure the load on a machine, and what threshold value to use.

The load on Unix systems is ultimately executed by user and system processes. Therefore, the measurement of load involved looking at the number and states of processes currently resident on a machine. Each process on the machine has an entry in what is known as a *process table.* This entry contains various pieces of information regarding that process. We considered the following methods for measuring the load on the machine:

1) Number of processes:

In this method one assumed that the load is proportional to the number of processes currently resident on the machine. Therefore, one would simply count the number of entries in the process table, and if this count exceeds the threshold number then the machine is regarded as being busy.

2) Ready to run processes:

At any instant the processes on a machine could be in one of several states. For example, a process could be ready to run and waiting for the CPU to become available, it could be waiting for a user to type in a command, it could be waiting for a disk I/O to complete, or it could have been put to *sleep* by another

process. Thus the total number of entries in the process table could give a misleadingly high value for the current load on the machine. A better idea of the load on the machine would be obtained if one only counted the number of ready-to-run process.

3) Number of active processes:

It was possible that at the instant of observing the process table, a number of processes were waiting for a fast event, such as disk I/O, to complete. These processes could soon become ready-to-run soon after the load value was calculated. Thus simply counting the number of ready-to-run processes could give a deceptively low value of the current load.

Each process has a *priority* value assigned to it by the system. The priority value is available in the process entry in the process table. The processes that await the completion of fast events are assigned a priority value higher than a certain value. Thus it is possible to identify such processes. Therefore an even better method for estimating load would be to count the number of active processes which includes ready-to-run as well as processes awaiting completion of fast events.

4) Complex methods:

Several pieces of information are available in a process entry in the process table. These include CPU and system time used by a process, the nature of the particular event awaited by the process, the owner of the process, etc. A complex method could incorporate this information in estimating the load on the machine.

Among the methods described above, we decided to use the number of currently active processes as an indication of load on the machine. This method was chosen because it was simple, easy to implement, and gave a reasonable estimate of the load on the machine. The threshold value to be used with this method is really a tunable parameter of the load sharing scheme. This threshold value was adjusted between two and four during the experimentation (Chapter Seven).

6.4 REPRESENTING THE STATE VECTORS

The load sharing scheme for the Perq system requires that each computer maintains a state vector which stores local and other computers' states. During the operation of the load sharing scheme a computer needs to access local and remote state vectors. Therefore, whatever form was used to represent state vectors, facilities had to be provided for access (reading and writing) to it by local and remote computers.

A possible way of representing the state vectors was to start a process called *state server* when the computer is switched ON. The state server on each computer would maintain the local state vector, and upon requests from local and remote computers supply and/or update the values in the state vector. To do this, some form of inter-process communication was needed. On Unix systems *pipes* and *signals* are used for inter-process communication. Unfortunately, pipes can only be established between related processes. Since the computers in the Perq system were to be switched ON independently, it would not have been easy to establish pipes between the state server and programs that needed its services. Note that some Unix systems provide *named pipes* and *sockets* [LEFFLER 83] for establishing communication among processes. These means of

communication would have been useful in implementing this representation of state vectors, but they were not available on the Perq system and therefore this method was not used.

Another way of representing the state vectors was to change the kernel so that it stored the states in some known locations in the memory. The local and remote computers could then manipulate these locations. This approach requires that every time a computer was added to the load sharing scheme, the kernel has to be modified and recompiled. As we wanted to avoid changes to the kernel, this approach was not implemented either.

Instead we implemented the state vectors as files. Each computer maintains its state vector in a known file. The local and remote computers can read from this file to find the current state of the local computer, as well as write to it to update their own values in the vector. Each state value is simply stored as a character **O, F, H, or U**. Since only four machines are involved each state file is only four bytes long. On each computer the first byte contains the local state, while the remaining bytes contain the states of other computers.

As explained in section 5.3.5 each computer stores the states in a different order. This order is given in a file on each computer. From this order, each computer can calculate which byte in another computer's state file contains its state, and therefore is able to modify it upon a state transition such as **O to F**. We were somewhat concerned about the possibility of more than one computer attempting to update a state vector simultaneously. So, a simple experiment was set up to determine whether the state values are corrupted in such a situation. It was discovered that as long as each computer updated different byte of the file all state values were updated properly.

However, if two computers try to update the same byte of a state vector then the order in which it is done becomes important. For example, Tweed might find the current state of the Tyne to be occupied (by reading first byte in the state file on Tyne) but has not yet updated the local state vector (by writing in the byte corresponding to Tyne in the state file on Tweed). In the meanwhile Tyne becomes free and somehow manages to update its value on Tweed to free. If now Tweed overwrites the latest value by the outdated value (O) then it will not consider Tyne as a possible candidate to execute its tasks. This problem can be prevented by making use of locking mechanisms to prevent any attempts to modify a byte while some other computer is already doing so. The computer wishing to update can try again after some time.

In our implementation we have assumed that the above race condition does not occur. Indeed it is very unlikely to occur when only four machines are involved. Furthermore, the updates are carried out in a way that attempts to avoid wrong values being passed on to state vectors. In the above example, upon becoming free Tyne would update its own state vector before updating its state value on other computers; thus making the above race situation more unlikely.

To manipulate the state vectors we developed the following primitive functions:

- local_update (machine, state)

This function updates the state vector on the local computer. The contents of the byte corresponding to computer 'machine' are changed to value 'state'. Integer value zero is returned upon successful completion; otherwise -1 is returned.

- **rem_update (machine, state)**

This function updates the state vector on a remote computer. On computer 'machine' the contents of the byte corresponding to computer that called the function are changed to value 'state'. Integer value zero is returned upon successful completion; otherwise -1 is returned.

- **local_get (machine)**

This function reads a state value from the local state vector. The value stored in the byte corresponding to the computer 'machine' is returned. If the value cannot be read for any reason, NULL value is returned.

- **remote_get (machine)**

This function reads a state value from a remote computer. From the state file on computer 'machine' the value of the first byte, corresponding to the state of computer, machine is returned. If the value cannot be read for any reason, NULL value is returned.

6.5 DETECTING THE STATE TRANSITIONS

The state vectors used by the transfer and the location policy of the load sharing scheme contain the states of the computers. As described in Chapter Five (5.3.7), local and sometimes remote state vectors have to be updated at the occurrence of some valid state transitions. In this section we consider each such valid transition separately and explain how it is detected in our implementation.

U to F:

This state transition takes place when a switched off computer is switched on. Before prompting the user to login, the PNX system executes commands stored in a file called **/etc/rc**. The system manager can put commands in this file that need to be executed every time the computer is switched on. For example the Newcastle Connection is initialised and started this way.

After the command to start the Newcastle Connection, we have added another command to run a program called **u_to_f.** This program changes the first component of the local state vector to F (by using local_update), and then tries to update (by using remote_update) the byte corresponding to the local computer in the state vectors of all other computers. The current states of the other computers are obtained and the local state vector updated accordingly. Recall that the computers that cannot be contacted are assumed to be unavailable (U).

F to O, H to O, and O to O:

These state transitions take place when a user logs on a free, hosting or an occupied computer respectively. A user is logged on by a program called *login* that gets started by another program called *init* after the computer has executed the commands contained in /etc/rc (as described above in section 6.5.1). Ideally the program *login* needs to be modified so that when a user manages to log on successfully, it changes the local state vector to show that the current state of the local machine is occupied (O).

However, as we wanted to avoid making changes to standard programs during this experimental implementation, we took another approach to detect

these transitions. After successfully logging in a user, the program *login* starts the *shell* for the new user. when the shell is invoked by the login program (as opposed to when started by a user) it first executes the commands contained in file called '*.profile*' which resides in the home directory of the user. We put the command to execute a program called **fho_to_o** in the *.profile* files of all the users. Thus whenever a user logs in, the program 'fho_to_o' gets executed. This program changes the first component of the local state vector to show that the local computer is now occupied.

The use of this approach to detect the transitions F to O, H to O, and O to O led to another problem. The creation of a window while running winit, the window manager, leads to invocation of another shell that first carries out commands in '.profile'. To avoid the program fho_to_o being executed every time a window was created, the command to execute fho_to_o is carried out only if winit is not running.

O to F, O to H:

These two transition can take place when a user logs off from a computer. On some Unix systems a special command exists (normally called 'logout') to log off a user. This command results in the execution of a program that logs off the user. The transitions O to F, and O to H can be detected by modifying this program. Unfortunately, on PNX a user logs off by typing in a control character (CTRL z) that terminates the shell program that was started by login when the user logged on. To detect the termination of this shell one needs to change the *init* program that awaits the terminations of shells started by login.

The init program is the very first to start when the computer is switched ON. Therefore one needs to be very careful in making any changes to it. An

error in init can result in the computer not starting up at all, thus preventing any corrections of init. For this reason, and because we have avoided making changes to standard programs where possible, we did not adopt this approach to detect the transitions O to F, and O to H.

Fortunately, in shell it is possible to trap the terminating signal (that gets generated when CTRL z is pressed) and invoke the execution of commands contained in a file. For example, by issuing the command

```
trap $HOME/.logout 0
```

one can cause the commands contained in file *.logout* (in the home directory of the user) to be executed when the current shell is terminated. We included the above trap instruction in the .profile file of all the users. So, when the login shell of a user is terminated the commands in file *.logout* are carried out. The .logout file contains a command to run a program called **o_to_fh**. This program checks whether the local computer is executing tasks for a remote computer. If it is, then the local state vector is changed to show that the local computer is hosting, otherwise it shows it to be free. The program o_to_fh then attempts to inform all the other computers about the transition that has taken place by modifying their state vectors accordingly.

F to H, H to F :

The state transition F to H takes place when a free computer receives a request from a process on another computer to execute a program. In contrast, the state transition H to F takes place when a computer finishes a request made by a remote computer, and there is no other remote request being carried out. In the Perq Unix United system the remote requests are carried out by a process called 'USRV'. Since a computer can be executing requests of several remote

processes, there can be a number of USRVs executing on a computer at any time.

Recall that a remote request can either be made directly by the user or system programs, or it can be made by the transfer policy of the load sharing scheme on another computer. In both cases a free computer gets into a hosting state upon arrival of the request; and can possibly get back to free state upon its completion. Therefore we need to detect the arrival and completion of both types of requests.

An obvious way to achieve this is by modifying USRV so that before carrying out a remote request it invokes a function called **f_to_h.** This function checks whether the current state of the machine is free. If so, it changes the local state to hosting; otherwise the local state remains unchanged. In either case the value of a counter to keep the current number of remote requests is incremented by one. USRV then executes the remote request and awaits its completion. When the remote request is completed, USRV invokes a function called **h_to_f.** This function decrements the value of the counter. If the value of the counter becomes zero, and the current state of the local computer is hosting (i.e it has not become occupied in the meanwhile) then the local state is changed to free. Otherwise the local state remains unchanged.

In order to avoid making changes to the USRV program, we implemented a different method to detect state transitions F to H, and H to F. We assumed (only for this experimental implementation) that the remote requests will only be made by the transfer policies; hence no remote requests are made directly by the user and system programs. With this assumption, the functions f_to_h, and h_to_f mentioned above are executed by the transfer policy before making the remote request and after the completion of the remote request respectively.

Another reason for making the above assumption is that the version of the Newcastle Connection running on the Perq system does not allow a remotely executing program to exec another remote program. It is however possible to add this facility to the Newcastle Connection [MARSHALL 87, STROUD 86].

6.6 CONCLUSIONS

In this chapter we have described how the load sharing scheme was implemented on the Perq system. Throughout this implementation we have avoided making any changes to the existing programs. As a result our implementation may not be the most efficient way (in terms of speed) of providing the Perq system with our load sharing scheme, but it works! Furthermore, if our load sharing scheme can prove to be beneficial with this implementation, we can be certain that with faster implementations, that make changes to the existing software, the load sharing scheme will perform even better. The performance of our implementation of the load sharing scheme is experimentally assessed in the next chapter.

CHAPTER SEVEN

EXPERIMENTS, RESULTS, AND DISCUSSION

7.1 INTRODUCTION

In this chapter we describe the experiments carried out to evaluate our load sharing scheme and discuss the results obtained. The essential method used in these experiments was to execute a given workload in a controlled system environment both with and without the use of the load sharing scheme. Thus two values of TC, the total completion time (as defined in section 5.3.1), are obtained. In order to measure the benefit achieved by load sharing, the term **Gain** is used and defined as follows:

$$\text{GAIN (G)} = \left[\frac{\text{TC without load sharing } (TC_w)}{\text{TC with load sharing } (TC_s)} - 1 \right] * 100\ \%$$

Note that if TC_s and TC_w, the total completion times with and without load sharing respectively, are equal then G, the gain, will be 0%. A negative value of G indicates the loss of performance as a result of load sharing the system.

So far in our discussion of the load sharing scheme for the Perq system we assumed that the transfer policy only selects suitable programs for remote execution. Therefore we need to find out what type of programs are suitable for remote execution using our load sharing scheme. To do this we investigated three types of programs: CPU intensive, I/O intensive, and Mixed. We were also

interested in observing the effect upon the gain of increasing computational requirements of each type of program. Therefore, it was necessary to be aware of the exact behaviour of the programs that represented three different types of programs. If the existing Unix utility programs were used as an example then it would have been difficult to change the computational requirements of these programs linearly. Therefore, we decided to use our own simple programs (described later in the appropriate sections).

To be able to make meaningful comparisons between their results, the experiments need to be carried out in a controlled and known environment. For example the values of TC_w and TC_s should be obtained by executing identical programs using identical data. Furthermore, the state transitions made during one experiment should also be made during the other. It must be realised, however, that on a real system it is almost impossible to guarantee absolutely identical behaviour of the system during successive experiments. For example, it would be difficult to control the scheduler so that it allocates CPU to processes in identical manners during the two experiments, or to ensure that the input or output data is placed in identical blocks in order to eliminate the effects of different disk arm movements!

Therefore, we shall assume that the uncontrollable factors in the system environment contribute, at least on average, the same amount of time during the measurement of TC_s and TC_w. We believe that the environment for the load sharing experiments on the Perq system was largely determined by the following controllable factors:

- The system state at the start of the experiment;.
- the state transitions during the experiment;
- the pattern of load generation and completion at each computer;

- and the threshold value used by the transfer policy of the load sharing scheme.

It is obvious that different combinations of the above factors can generate numerous possible environments to carry out the experiments. The environments used, the experiments carried out, and the results obtained for CPU intensive, I/O intensive and Mixed programs are described below in sections 7.2 , 7.3 and 7.4 respectively. In section 7.5 we describe the experiments and present the results for two real Unix utility programs.

Each experiment was repeated at least three times (in most cases five times) and the results given here use the average value. In all the experiments the difference between the successive readings was very small.

7.2 A CPU INTENSIVE PROGRAM

The following program (in pseudo-pascal notation) consisting of three nested **FOR** loops was used as an example of a CPU intensive program:

```
Begin
   For loop1 := 1 to LIMIT do
      For loop2 := 1 to LIMIT do
         For loop3 := 1 to LIMIT do
            count := count+1;
   writeln (count);
End.
```

The maximum value of the control variable, **LIMIT**, was passed as a parameter to the program, thus providing means of controlling the computation required. Table 7.1 shows different values of LIMIT with the corresponding

LIMIT	COMPUTATION TIME (C) (seconds)	LIMIT	COMPUTATION TIME (C) (seconds)
10	1.0	140	54.0
25	1.4	160	80.8
40	2.4	180	116.6
50	3.4	200	158.0
60	5.2	210	179.6
70	7.8	220	206.0
80	11.0	230	240.6
100	20.4	240	272.0
120	34.4	250	307.4

Computation Times for a CPU Intensive Program

TABLE 7.1

values of the time taken to complete the above program when executed on Tweed in the absence of any other user-level program. The Unix *time* facility was used to measure the times. These times indicate the computation time (C) required by the program.

The above program was repeated on all the other computers with the value of LIMIT being 250. The average value obtained on different computers is shown in Table 7.2. It is clear that all computers take almost identical time to

COMPUTER	COMPUTATION TIME FOR LIMIT = 250
CATCLEUGH	309
KIELDER	307
TYNE	301
TWEED	302

AVERAGE COMPUTATION TIMES ON DIFFERENT COMPUTERS

TABLE 7.2

execute the CPU intensive program. Hence one can attribute any differences in the values of TC_s and TC_w to load sharing rather then the differences in the speeds of the computers involved.

The values of the controllable factors of the environment used during the CPU intensive experiments were as follows:

1) **Initial system state:** The load sharing scheme is evaluated for different initial states of the system. Therefore, the experiments for the CPU intensive program were carried out under the following initial states:

a) One computer is occupied, and all the others are unavailable. Since no free computers are available we shall refer to the gain as **G0**.

b) One computer is occupied, one is free, while the other two are unavailable. The gain obtained in this case is denoted by **G1**.

c) One computer is occupied, two are free, and one is unavailable. The gain in this case is **G2**.

All the experiments were carried out from the occupied computer. Furthermore, the state vectors of the occupied and the other computers involved were fully updated before the start of the experiment.

2) State Transitions: In order to interpret the results clearly, and be able to attribute any differences in the gains only to initial system state, we decided that there would be no external state transitions during all the experiments. Any state transitions occurrring as a result of load sharing itself were, of course, allowed.

3) Load Generation: Since no external state transitions were to be allowed, any load on the system was generated by a Control program running on the occupied computer. This program received two parameters: one indicating whether to take measurements with or without the load sharing scheme, and the other representing the value of LIMIT to be used by the CPU intensive program.

The Control program noted the current time in a known file and then forked four child processes at the interval of three seconds. Each child process then executed the CPU intensive program with the supplied value of LIMIT. The

parent process then awaited the completion of all four child processes, and then made a note of the current time.

The output of each child process was directed to a separate file. When measuring TC_w, the completion time without load sharing, this output consisted of the final value of **count** generated by the CPU intensive program. However, when measuring TC_s, the completion time with load sharing, the output also consisted of the load value and the identity of the machine chosen by the load sharing scheme to execute the program. This enabled us to tell which processes were executed remotely.

The reason for using a three second interval between forking of child processes was that it allowed sufficient time for the load sharing scheme to update the local state vector before another process made use of that vector. Furthermore, in a real interactive environment one presumably does not expect two requests to be made for load sharing programs within three seconds!

We decided to fork four child processes because there were four machines involved in the experiment, and we were interested in the value of gain when each machine executed one program.

4) Threshold Value: The threshold value of two was used by the load sharing scheme to decide whether the local machine is busy or not. Thus if there were more than two processes executing on the local machine then the machine was regarded as being busy, and an attempt was made to find another computer to execute the program.

The value of two was chosen because there was always one active process on the local machine (the child process trying to find the load); however the parent process (taking the measurements) also had an entry in the process table and its status (i.e. sleeping, running, or waiting) could not be known in advance.

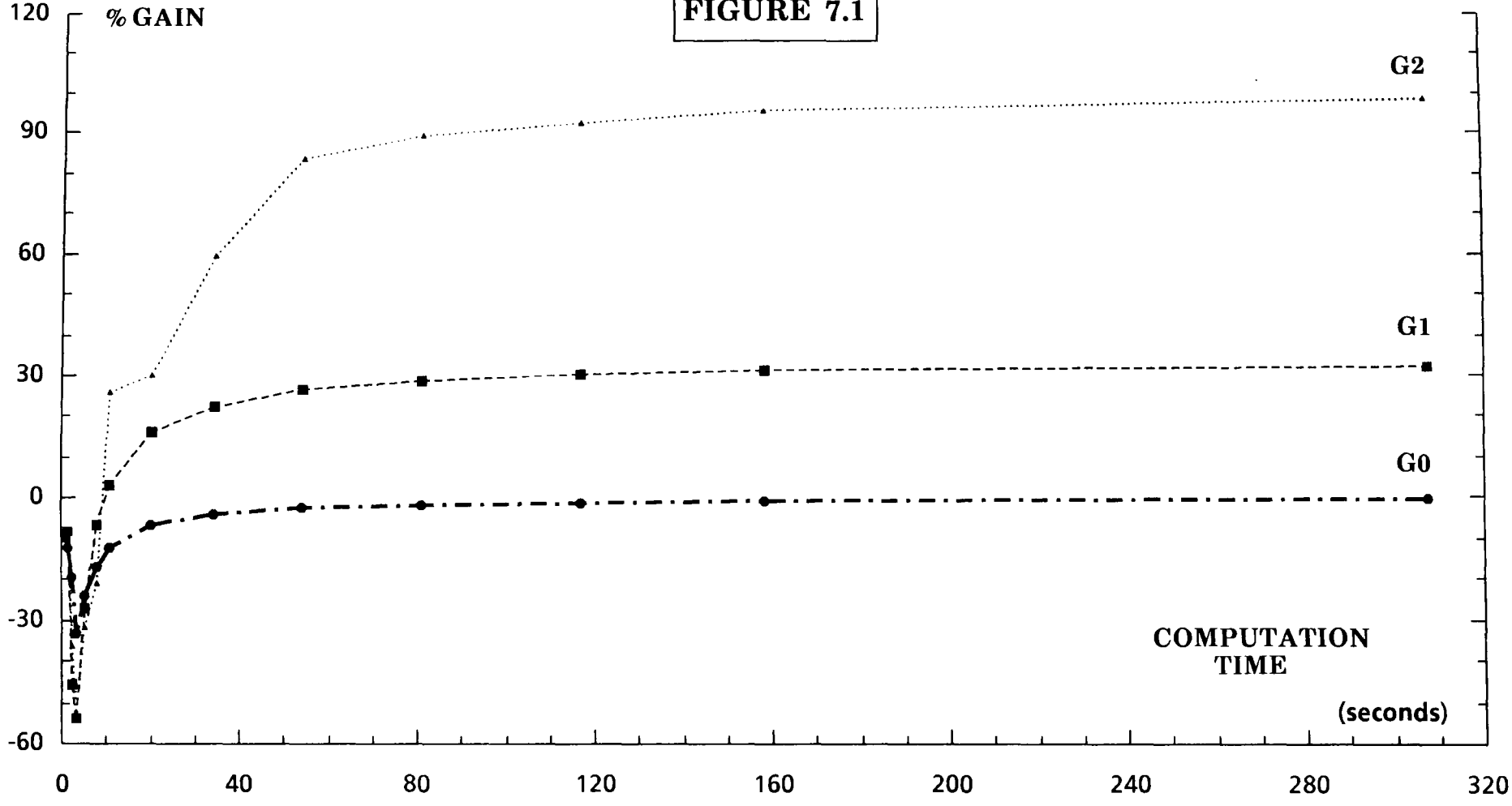
Therefore, if one found more than two active processes in the process table than at least one CPU intensive program was being executed locally.

It is clear from the above description of the controllable factors that during the investigation of CPU intensive program only the initial state of the system was varied, while the other factors remained unchanged. We measured the values of TC_w and TC_s under different initial state conditions for all the values of LIMIT indicated in Table 7.1. Thus we calculated gains G0, G1, and G2 for each value of LIMIT under three different initial states. The values of these gains have been plotted against the computation (C) in Figure 7.1.

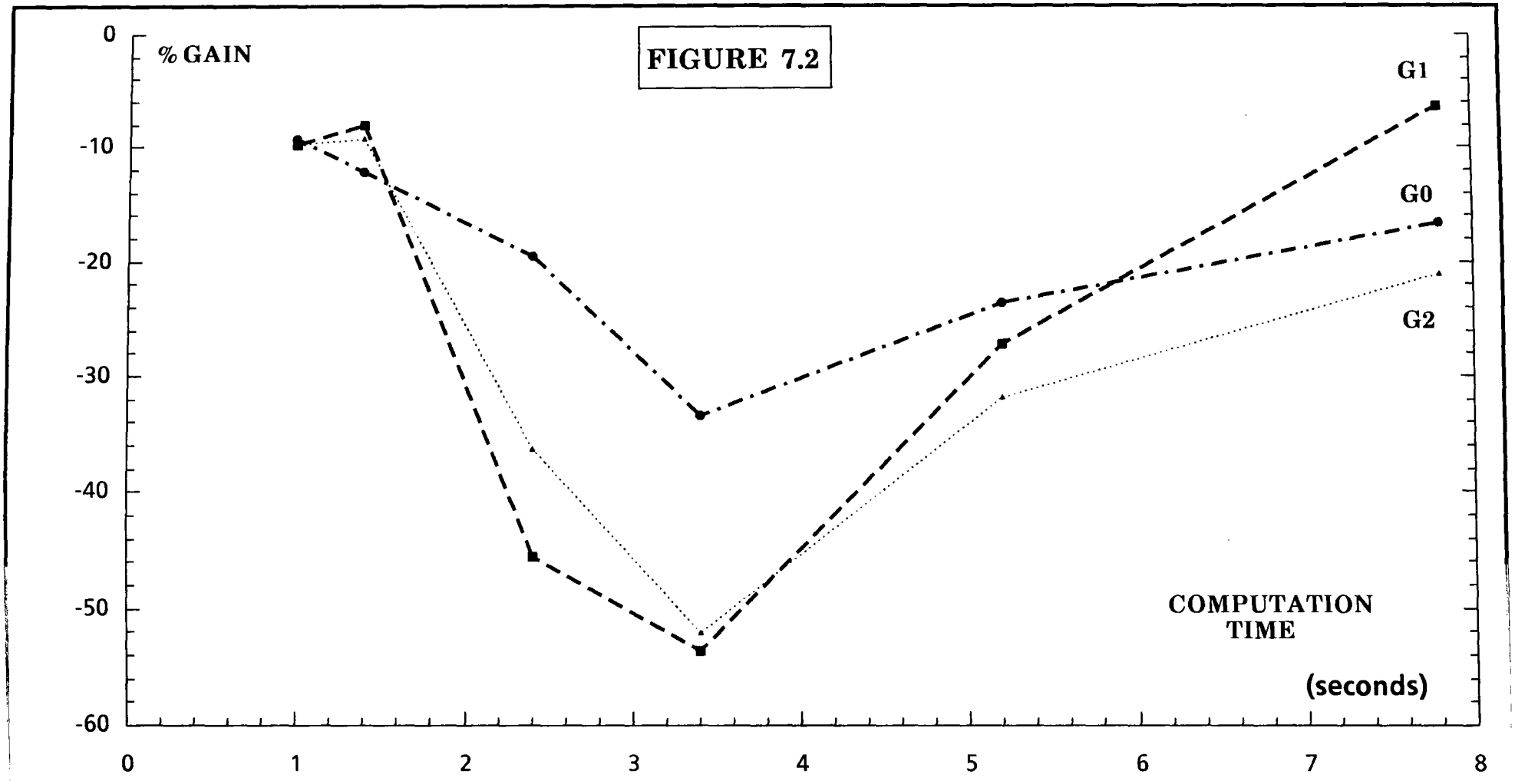
We shall first discuss the transient parts of the graphs that occur before the computation time of approximately 8 seconds. This transient part is shown more clearly in Figure 7.2. We note that here all three gains start off being negative and then become much worse before recovering. When the computation time is low, the overhead of the load sharing scheme is comparatively high, and therefore initially we get negative gain. To understand the further drop in gain (before its recovery) we need to recall the way the load sharing scheme works. If the program is suitable for load sharing then the first thing that the load sharing scheme does is to find out whether the local machine is busy.

Now, when the value of Computation time is less than approximately 2.2 seconds, the first child process completes before the next one is forked. Thus no child process finds the local machine to be busy, and the load sharing scheme does not do any more work. Hence at these values of computation the overhead of load sharing is smaller, and therefore the value of gains is less negative.

As the value of the computation time becomes greater than about 2.2 seconds, the forked child processes begin to discover that the local machine is



GAINS FOR A CPU INTENSIVE PROGRAM



TRANSIENT "GAINS" FOR A CPU INTENSIVE PROGRAM

busy. This results in execution of more expensive parts of the load sharing scheme to find a free computer in the local state vector. In the case of G0 no free computer is found, and the load sharing scheme stops there. So, we note that the G0 becomes further negative but not as much as G1 and G2.

In the case of G1 and G2 the child processes find free machines in the local state vector, and execute even more expensive part of the load sharing scheme to confirm the free state of a remote computer. Although these processes are then executed remotely, they do not require enough computation to make up for the extra work they have already done to get there. As a result the gain becomes even more negative.

Fortunately, in the case of CPU intensive programs the full overhead of load sharing remains constant irrespective of the computation required by the program. So as the computation time increases, the time spent on the remote computer begins to make up for the earlier loss and we get positive gains.

One would expect that as the value of computation becomes very large compared to the overhead of load sharing, the gains would reach their theoretical steady state values. In our experiments for a CPU intensive program this value of steady state gain is given by the following expression:

$$\left[\frac{\text{Total number of programs executed}}{\text{Number of programs executed locally}} - 1 \right] * 100\ \%$$

So, for G0, G1, G2 the steady state gain is 0%, 33.3%, and 100% respectively. We note in figure 7.1 that for large computations the gains G0, G1, and G2 asymtotically approach their theoretical values.

7.3 AN I/O INTENSIVE PROGRAM

The following program (in pseudo pascal notation) was used as an example of an I/O intensive program:

```
For count := 1 to FACTOR do begin
    Get to the start of I/O files;
    While not end of file do begin
        Read 100 bytes from input file;
        write 100 bytes to output file;
    end;
end;
```

The maximum value of the control variable, **FACTOR**, was passed as a parameter to the program, thus providing means of controlling the I/O intensive computation.With this program the size of the output file does not become very large, and same input file could be used to provide different amount of computation. Table 7.3 shows different values of FACTOR, with the corresponding completion times when an input file of 900 bytes was used on Tweed and Catcleugh (the two computers used in the I/O intensive experiments). There is approximately 10% difference in the values obtained for Tweed and Catcleugh, possibly due to different layout of data on the disks. As the later results will confirm, this difference is negligible compared to the gain or loss in the performance of the system due to load sharing.

FACTOR	Tweed	Catcleugh
8	2.65	2.83
16	2.77	2.70
32	2.80	3.01
64	4.70	4.68
128	9.02	8.75
256	14.78	12.25
512	22.77	23.82
1024	59.65	55.22
2048	130.78	144.47
4096	266.40	234.08

COMPUTATION TIMES FOR AN I/O INTENSIVE PROGRAM

TABLE 7.3

The values of the controllable factors of the environment used during the I/O intensive experiments were as follows:

1) **Initial system state:** In the case of I/O intensive program, we were interested in observing the effects of the locations of I/O files on the gain of the load sharing scheme. Therefore the experiments for I/O intensive programs were carried out in the following initial system states.

a) One computer is occupied, the others are unavailable. All I/O files are located an the Occupied computer. Since the I/O files are on the same computer as the control program we refer to the gain as G_I0.

b) One computer is occupied, one is free and the other two are unavailable. All the I/O files are located on the occupied computer. The gain in this case is G_I1.

c) Two computers are occupied and two are unavailable. The I/O files for one program are located on the occupied computer where the control program runs, and for the other program they are located on the other occupied computer. Since the I/O files for one program are remote from the control program, the gain is called G_r0.

d) One computer is occupied, one is free, and the other two are unavailable. The I/O files for one program are located on the occupied computer, and for another program they are located on the free computer. The gain in this case is called G_r1.

2) **State transitions:** No state transitions, except those occurring as a result of load sharing scheme, were allowed during the experimentation with the I/O intensive program.

3) **Load Generation:** The load was generated by the Control program running on the occupied computer. This program received two parameters: one indicating whether to take measurements with or without the use of load sharing scheme, and the other other representing the value of **FACTOR** to be used.

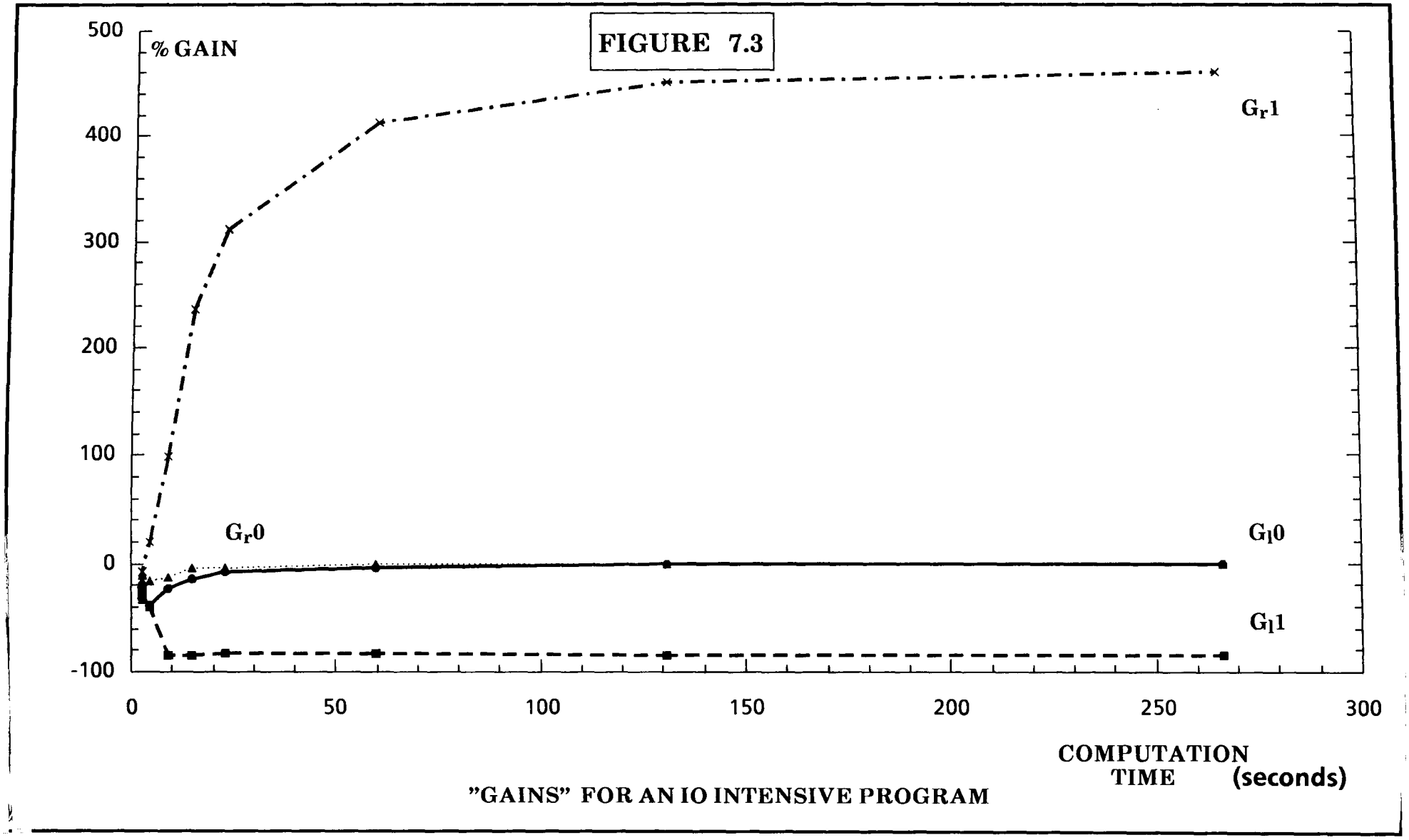
The control program made note of the current time and then forked two processes at interval of three seconds. Both child processes then execute the I/O intensive program with the supplied value of FACTOR. The control program awaited the completion of both processes, and made note of the current time. Note that both processes use separate but identical I/O files in order to avoid contention.

4) **Threshold Values:** Since only two processes were being forked by the control program the threshold value of one was used. Thus, if more than one program is *active* then the computer is regarded as being busy.

The gains obtained for the I/O intensive program are plotted against the computation (shown for Tweed in table 7.3) in Figure 7.3. Considering the transient parts of the graphs (shown more clearly in Figure 7.4) we note that initially all graphs are negative. This is because the overhead of load sharing is high compared to the computation required by the program. G_l0 and G_r0 become further negative before recovery. The reasons for this behaviour are related to the way the load sharing scheme works and have already been explained for the CPU intensive program.

In the case of G_l1 one I/O intensive program gets executed remotely on previously free computer, while its I/O files are located on the occupied computer. This means that the remotely executing program has to access its data over the Ethernet. This remote access of data carries its own overhead, making the gain G_l1 even more negative. The overhead of remote data access is associated with each read and write system call. However, since the ratio of remote to local read or write system call remains constant irrespective of the computation required, the gain G_l1 becomes permanently negative. It reaches its steady state value of approximately -90% after the initial load sharing overhead has been *offset* by the computation required (about 10 seconds).

In the case of G_r1, one program is executed on a previously free computer where its I/O files are residing. There are two reasons for very high (note the change in scale of Y-axis in Figure 7.3) values for G_r1. First, one program gets executed remotely; thus two CPUs are employed instead of one. Secondly, and



"GAINS" FOR AN IO INTENSIVE PROGRAM

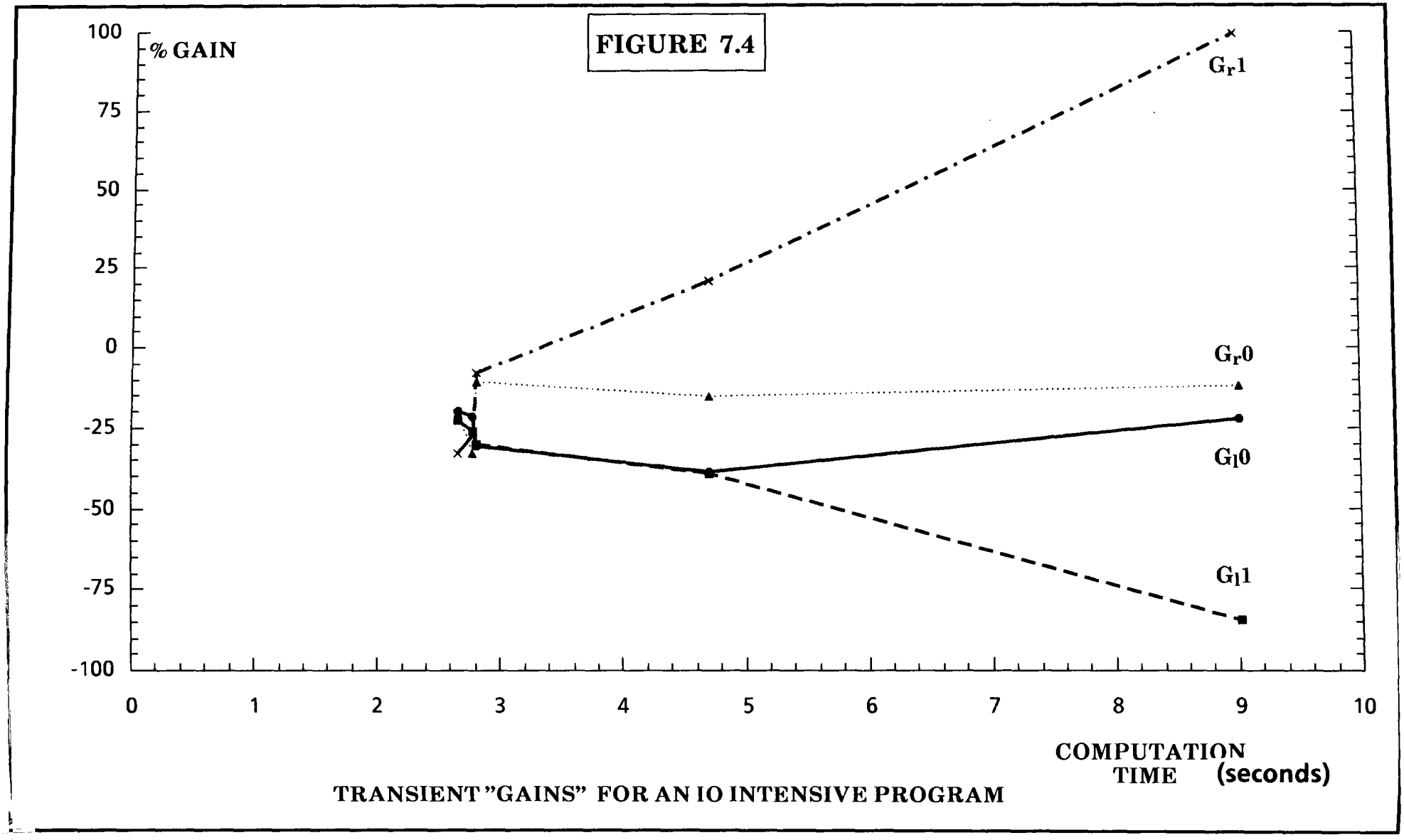


FIGURE 7.4
% GAIN
100
75
50
25
0
-25
-50
-75
-100
0
1
2
3
4
5
6
7
8
9
10
G_r1
G_r0
G_l0
G_l1
COMPUTATION TIME (seconds)
TRANSIENT "GAINS" FOR AN IO INTENSIVE PROGRAM

more importantly, in the absence of any load sharing both programs were executed on the occupied computer; therefore one program has to access its I/O files over the Ethernet. By executing that program where its I/O files were residing, the load sharing scheme avoided a substantial overhead of remote I/O system calls. As a result we note that once the computation required by the program exceeds the initial overhead of load sharing, the gain becomes steady at about 500%.

7.4 A MIXED PROGRAM:

A mixed program contains both CPU intensive and I/O intensive parts. As an example we used the following program (in pseudo pascal notation) :

```
Begin
        Do CPU intensive part;
        Do I/O intensive part;
        Do CPU intensive part;
        Do I/O intensive part;
End;
```

The CPU and I/O intensive parts are same as the ones described for CPU and I/O intensive programs described in sections 7.2 and 7.3 respectively. Therefore, the mixed program received two parameters: LIMIT to control the computation required by each CPU intensive parts; and FACTOR to control the computations required by each of the I/O intensive parts.

For the experiments the values of LIMIT and FACTOR were chosen such that when the above program was executed on its own the contributions (to total computation time) of CPU and I/O intensive parts were roughly equal. Table 7.4

shows the combinations of LIMIT and FACTOR, along with the corresponding total computation.

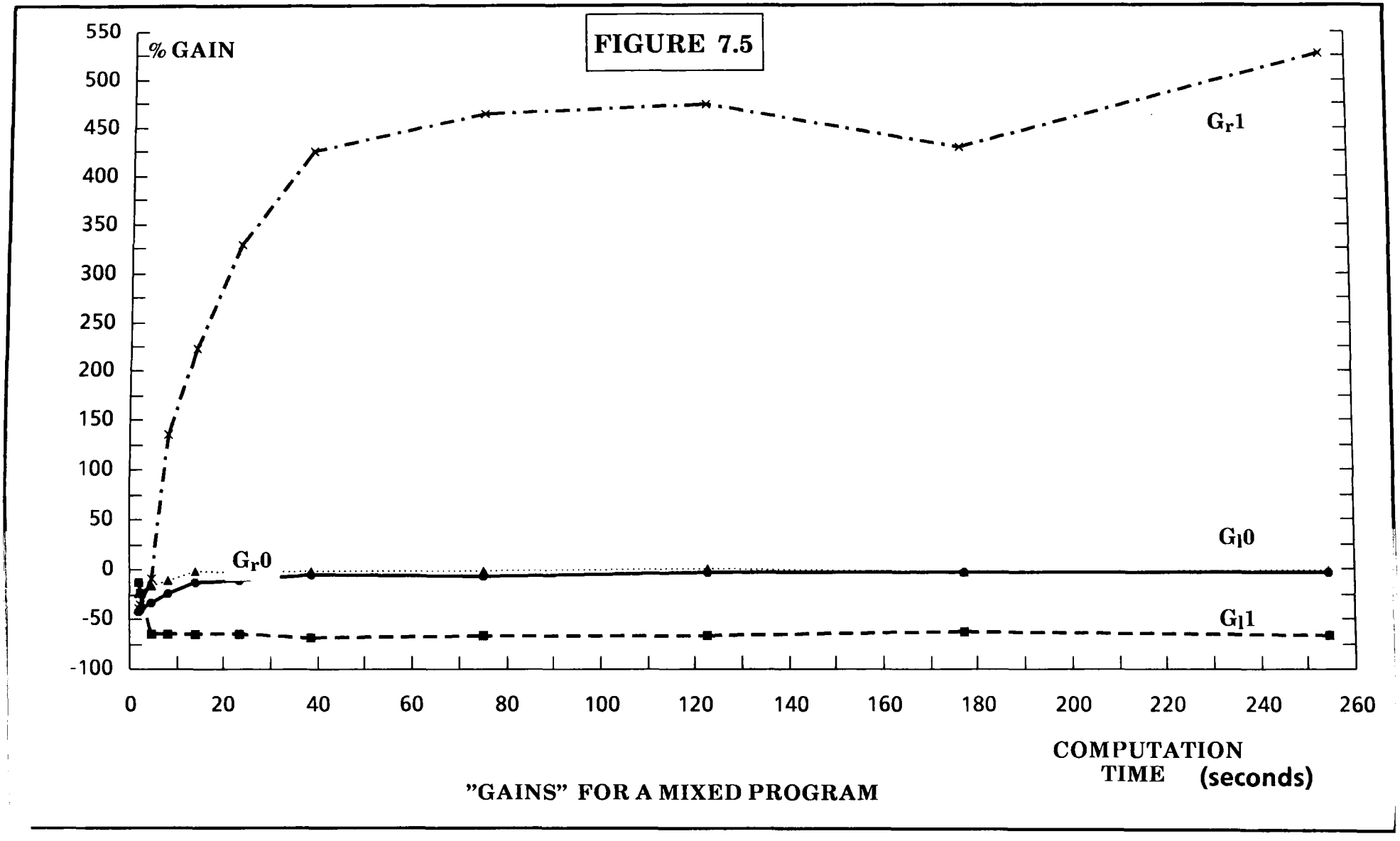
LIMIT	FACTOR	COMPUTATION (SECONDS)
10	1	2.04
25	4	2.42
40	8	4.57
50	32	8.25
60	64	14.00
70	128	23.50
80	256	38.59
100	512	75.26
120	800	122.62
140	1024	177.19
150	1800	254.41

COMPUTATION TIMES FOR A MIXED PROGRAM

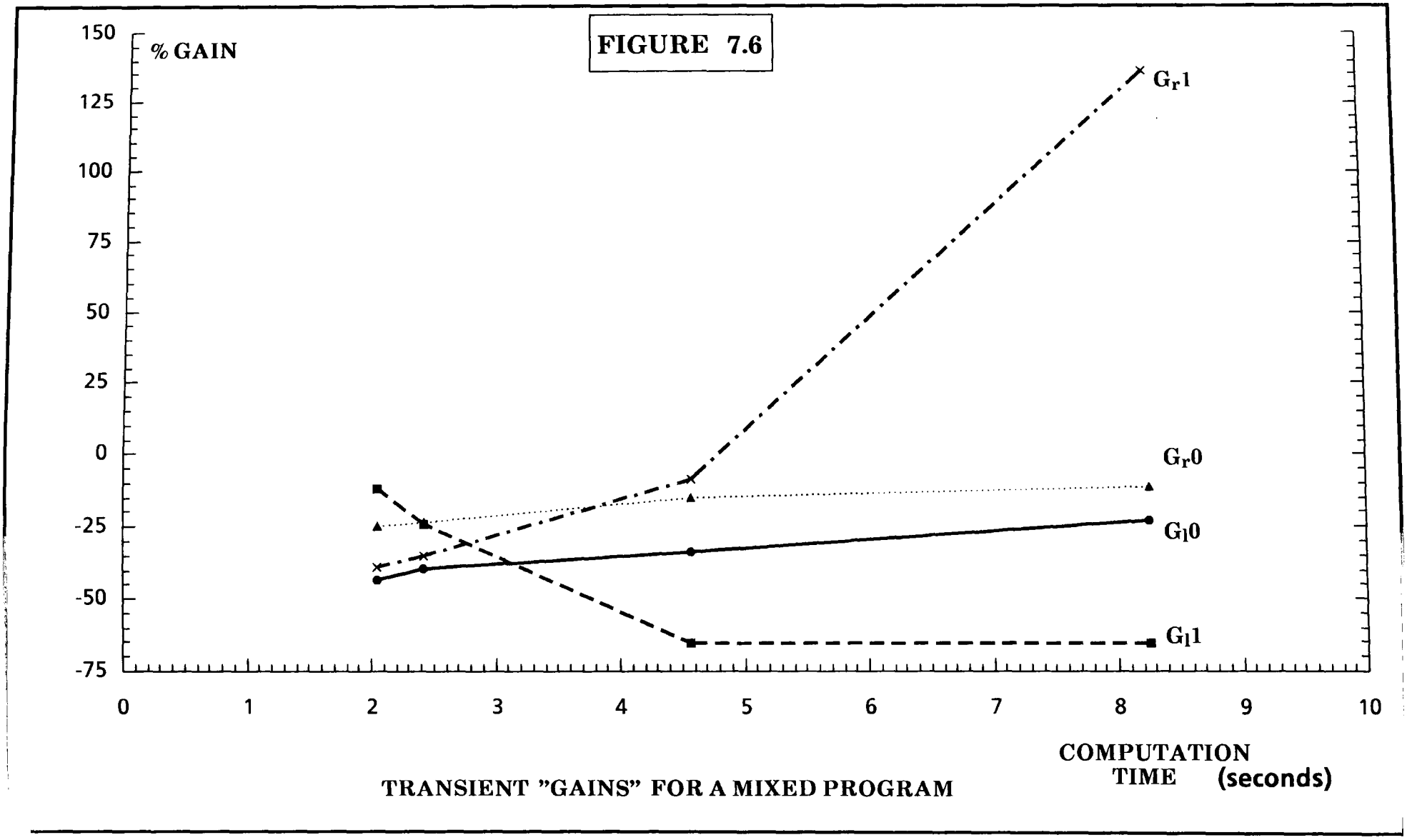
TABLE 7.4

The values of the controllable factors of the environment used during the mixed program experiments were same as used for the I/O intensive program. The difference was that the control program running on the occupied computer forked two processes that executed the mixed program (rather than the I/O intensive program). Therefore the control program had to be supplied with three parameters: one to indicate whether to take measurements with or without the use of load sharing; and the other two indicating the values of LIMIT and FACTOR.

The gains obtained for the mixed program are plotted against the computation (given in Table 7.4) in Figure 7.5. The transient gains are shown in



"GAINS" FOR A MIXED PROGRAM



TRANSIENT "GAINS" FOR A MIXED PROGRAM

Figure 7.6. We note that these gains are very similar to the gains of the I/O intensive program. The main difference is that in this case the steady state gain $G_{|}1$ is at about -65% (compared to -90% for an I/O intensive program). This is because of the CPU component of the program which increases the gain, but not sufficient enough to make up for the loss due to the I/O component.

Although the CPU and I/O components contributed the same amount of time when only one mixed program was run, the results indicate that once there is a competition for the disk access, the I/O component of the program dominates the outcome of the gain of load sharing.

7.5 UNIX UTILITY PROGRAMS

We investigated our load sharing scheme using two commonly used Unix utility programs called **nroff** and **cc**. Although the precise behaviour of these programs was not known, thus making it difficult to interpret the results, we wanted to observe the gains with some *real* programs.

7.5.1 nroff:

Nroff is a text formatting program that can be run with several different options. We used the following form:

```
nroff -ms infile > outfile
```

Five different sized input files were used to provide different computations. Table 7.5 shows the file sizes and the corresponding computation times.

The environment used for the nroff program was same as for the I/O intensive program. However, in this case the control program received only one

parameter which indicated whether to take measurements with or without the use of load sharing scheme. The name of the input file was *built into* the control program, and its contents were changed to that of different sized input files.

FILE SIZE (BYTES)	COMPUTATION (SECONDS)
74	33.33
340	37.22
4590	65.42
11377	101.64
22449	300.47

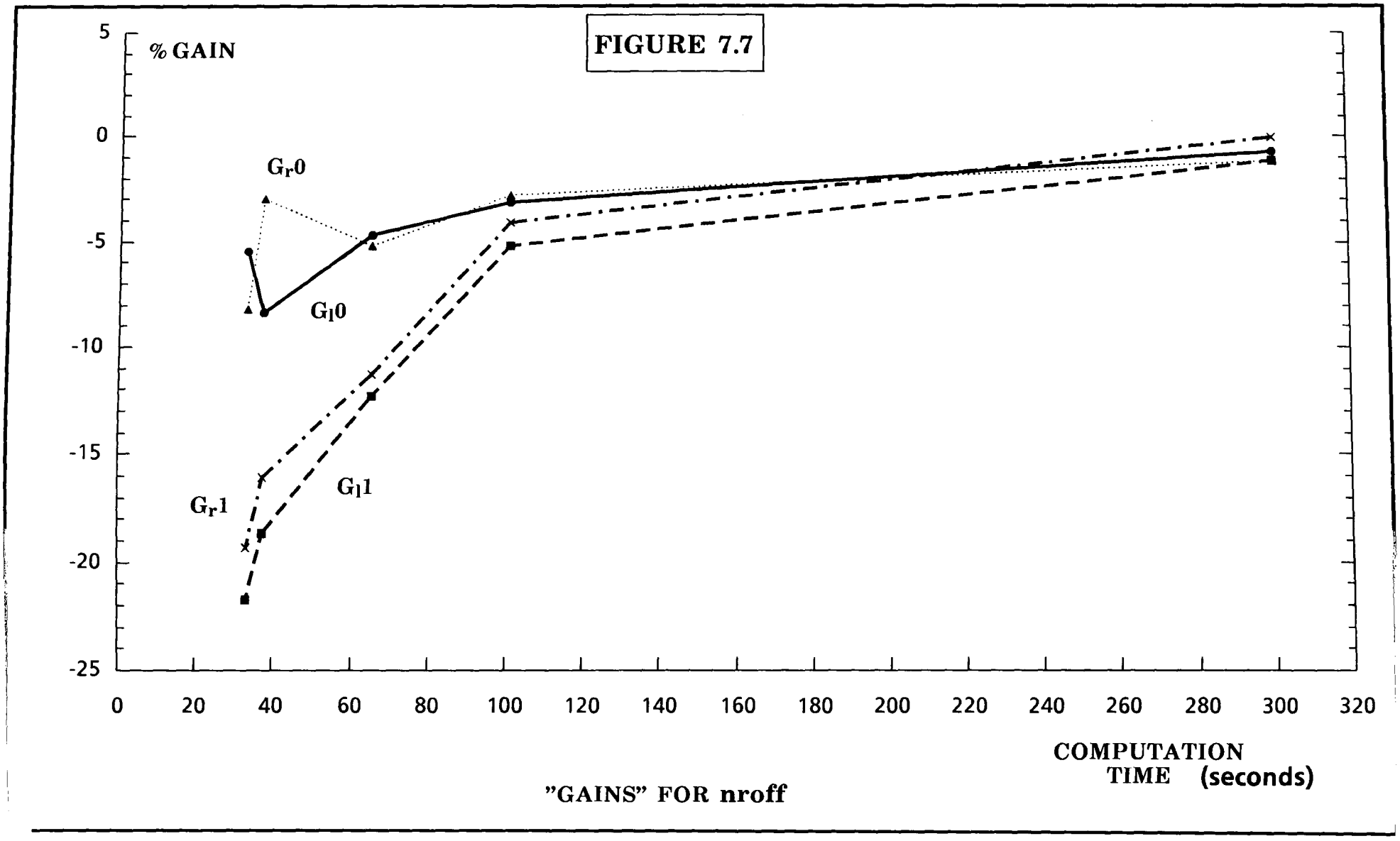
COMPUTATION TIMES FOR nroff

TABLE 7.5

In order to find a *typical* size of an nroff source file we carried out a survey of all nroff source files resident under the */user* directory on a multiuser general purpose machine extensively used by the research staff of the Computing Laboratory. The results of this survey are summarised in Table 7.6. We found that the average size of an nroff source file was 10367 bytes, and more than 77% files were larger than 1000 bytes.

The results for the nroff program are shown in Figure 7.7. Here we note that G_r1 and G_l1 start being negative and become steady at around 0%. This is different from all the previous results. The fact that G_l1 moves towards 0% indicates that as the computation increases nroff becomes more of a CPU intensive program, but because of remote accesses to I/O files the gain does not become higher than 0%. Note that for an average sized source file the gain is in the region of -5% to 0%.

The behaviour of G_r1 can be explained if we remember that nroff not only accesses I/O files supplied by the user, but also needs macro files that contain



"GAINS" FOR nroff

GREATER THAN (bytes)	PERCENTAGE
1000	77.67 %
2000	60.33 %
3000	50.48 %
4000	43.69 %
5000	39.32 %
11000	25.59 %
20000	17.41 %

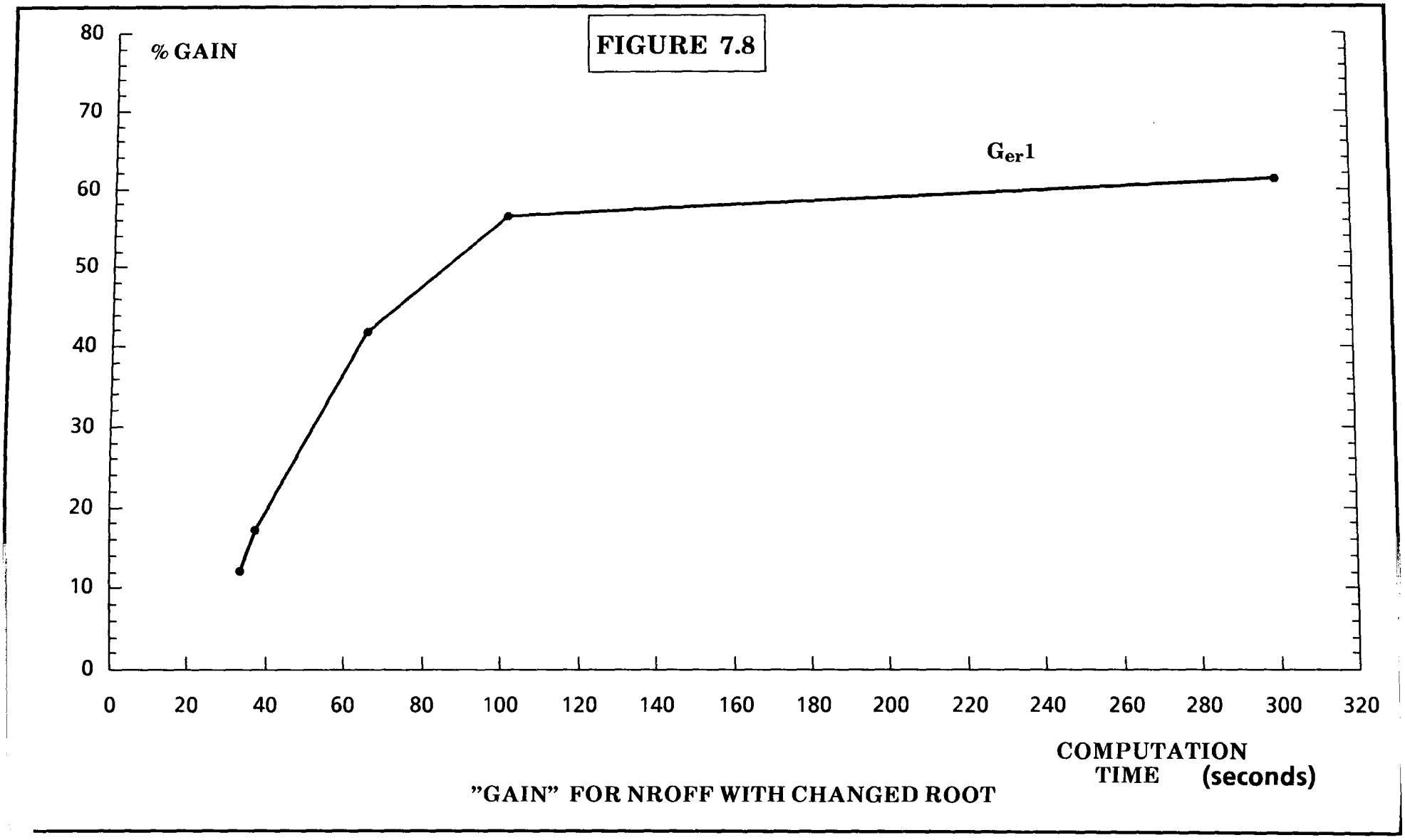
nroff SOURCE FILE SIZES

TABLE 7.6

details of how to interpret the formatting commands given by the user in the input text. Normally the macro files are accessed from the machine which forked the process that executes the nroff program. Thus, in the case of G_r1 even though the I/O files are located on the program where the program is executed, the macro files are still accessed over the Ethernet from the occupied computer. Hence a low value for gain is obtained.

The Newcastle Connection provides a system call to execute a program with root (/) changed to the computer that executes the program. If the remotely executing program now needs a file, then it would be searched for on the remote computer. If this *changed root* method of remote execution is used to execute nroff on the remote free computer (during the measurements for G_r1) then the macro files would be accessed from the previously free computer. Hence, the overhead of remote access of macro files will also be avoided.

We obtained another set of values for G_r1 using the changed root execution. The result is plotted in Figure 7.8. We note that the gain in this case is positive, and for an average sized source file it is over 55%.



"GAIN" FOR NROFF WITH CHANGED ROOT

7.5.2 cc :

cc is a compiler for the programming language C. We investigated our load sharing scheme using this program in same environment as described for the I/O intensive program. CC program was run using different sized input files to provide different amount of computation. Table 7.7 shows the file sizes with the corresponding computations.

FILE SIZE (BYTES)	COMPUTATION (SECONDS)
45	24.73
391	34.16
900	40.37
2016	47.08
4386	60.14
11277	109.89

COMPUTATION TIMES FOR cc

TABLE 7.7

In order to find a *typical* size of a cc source file we carried out a survey of all cc source files resident under the */user* directory on a multiuser general purpose machine extensively used by the research staff of the Computing Laboratory. The results of this survey are summarised in Table 7.8. We found that the average size of a C source file was 3944 bytes, and more than 58% files were larger than 1000 bytes.

The results obtained for the cc program are shown in Figure 7.9. These results show that both G_l1 and G_r1 remain negative, and for an average sized file the gain is in the region of -20% to 0%.

GREATER THAN (bytes)	PERCENTAGE
1000	58.10 %
2000	38.99 %
3000	30.18 %
4000	25.07 %
5000	21.12 %
11000	09.77 %
20000	04.08 %

cc SOURCE FILE SIZES

TABLE 7.8

The attempts to investigate the gain G_r1 when executed with changed root failed initially because the cc program forks other programs, and the *execute with changed root* facility could not cope with that. Further investigation revealed that the problem was occuring in the cc program where it invoked shell to start the linker by using '<' and '>' symbols to assign the input and output files respectively. We could not readily extend the functionality of *excr*, but by changing the cc program so that it opened the I/O files before using the exec system call to invoke the linker, we managed to get readings for G_r1 with changed root. The cc program became faster because of our changes, therefore TC_w and TC_s (completion time without and with load sharing) were taken again. The result for G_r1 with changed root is shown in Figure 7.10. We note that the gain is now increasingly positive for computation times larger than 60 seconds. However the gain for an average sized input file is still in the region of 0%.

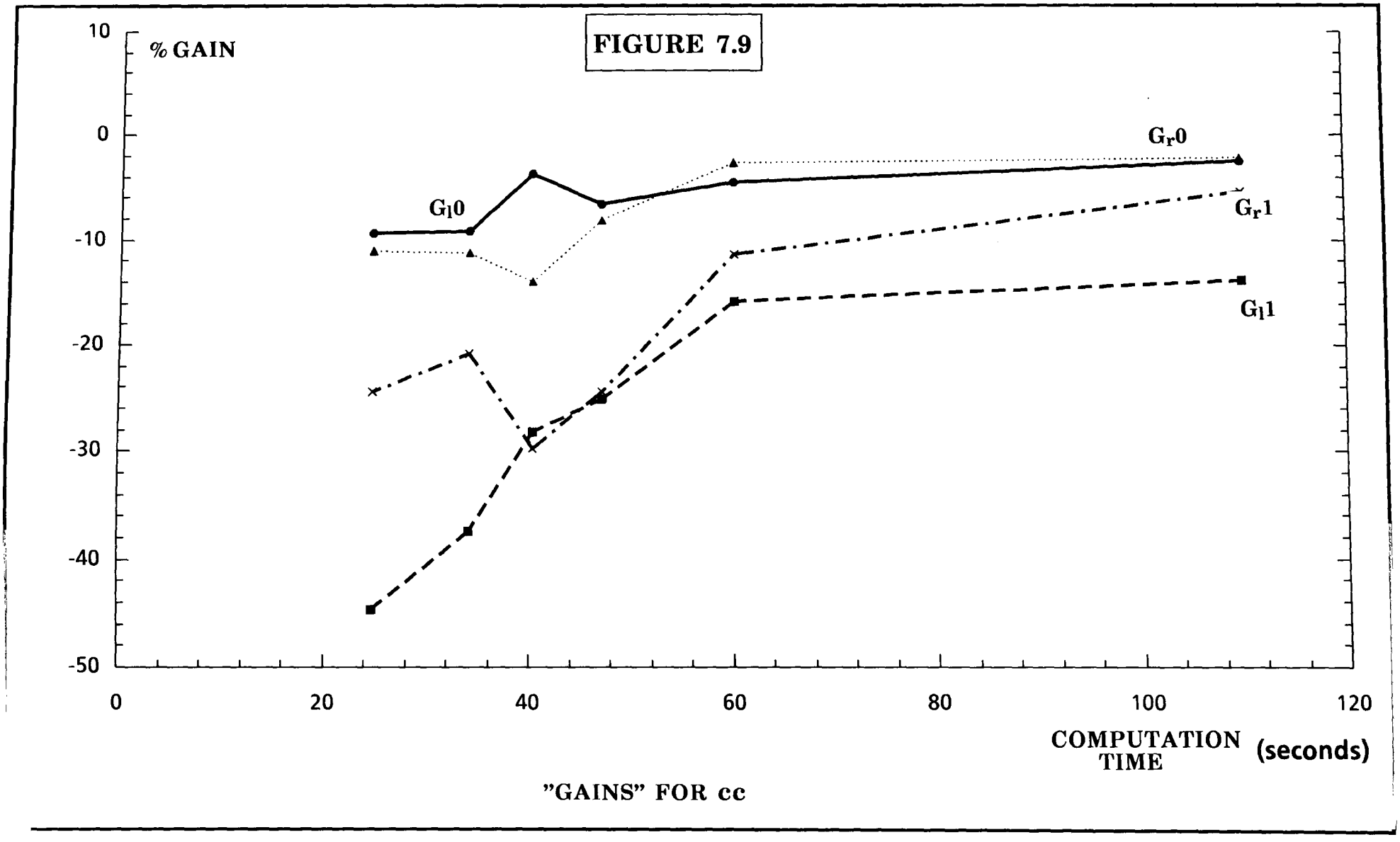


FIGURE 7.9
% GAIN
10
0
-10
-20
-30
-40
-50
0
20
40
60
80
100
120
COMPUTATION TIME (seconds)
G_r0
G_l0
G_r1
G_l1

"GAINS" FOR cc

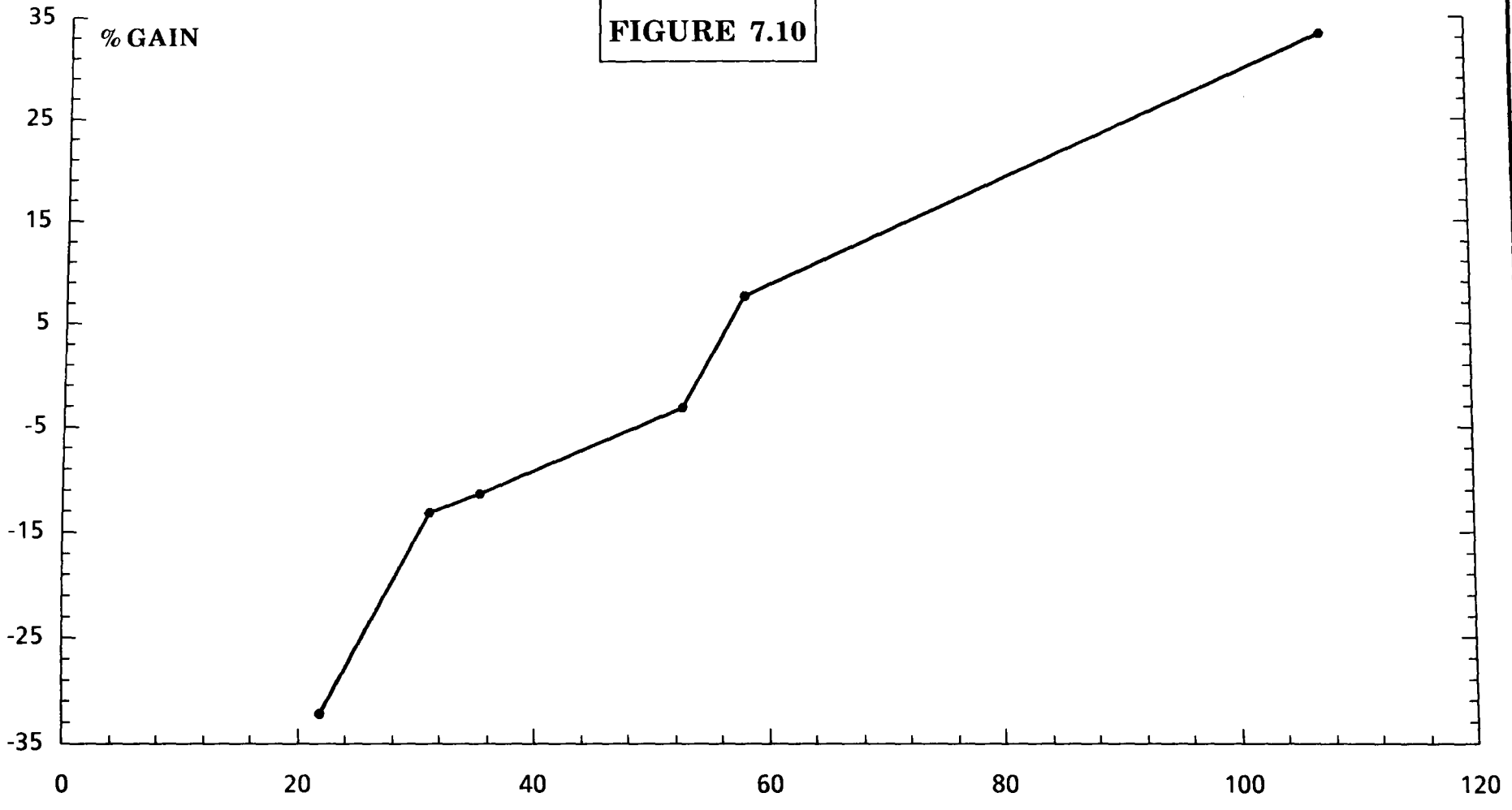


FIGURE 7.10
% GAIN
35
25
15
5
-5
-15
-25
-35
0
20
40
60
80
100
120
COMPUTATION (seconds)
TIME
"GAINS" FOR cc WITH CHANGED ROOT

7.6 CONCLUSIONS

In this chapter we have presented the results of the experiments carried out to evaluate our load sharing scheme. We used different types of programs, and discovered that the CPU intensive programs are suited for load sharing as long as their computational requirements are more than about 10 seconds. The I/O intensive and mixed programs are suited for load sharing only when their I/O files are located on the computer where the program is to be executed, in which case very high gains can be obtained. For standard Unix utilities we found that it is necessary to know the general behaviour of the program before it can be decided whether they are suitable for remote execution as a result of load sharing. If the I/O files of a Unix utility programs are residing on the remote free computer then executing the program with changed root can result in positive gain.

CHAPTER EIGHT

CONCLUSIONS

In this thesis the problem of load sharing in distributed computer systems has been studied. After discussing the role of load sharing within the context of distributed systems, we reviewed several studies already carried out in the field of load sharing. The fundamental issues that need to be resolved in order to implement a load sharing scheme in any distributed system were identified. We then considered an existing Unix United system and designed a simple load sharing scheme for it. This load sharing scheme was implemented and experiments were carried out to evaluate it.

In this chapter we look back at our work and discuss what has been learnt from it. In section 8.1 we consider our load sharing scheme for the Perq system to establish the extent to which it satisfies the desirable features described in section 3.2.8. In section 8.2 we give some suggestions for future work involving our load sharing scheme. We conclude this thesis in section 8.3 by discussing what we have learnt from our work about our scheme in particular, and load sharing in general.

8.1 DESIRABLE FEATURES

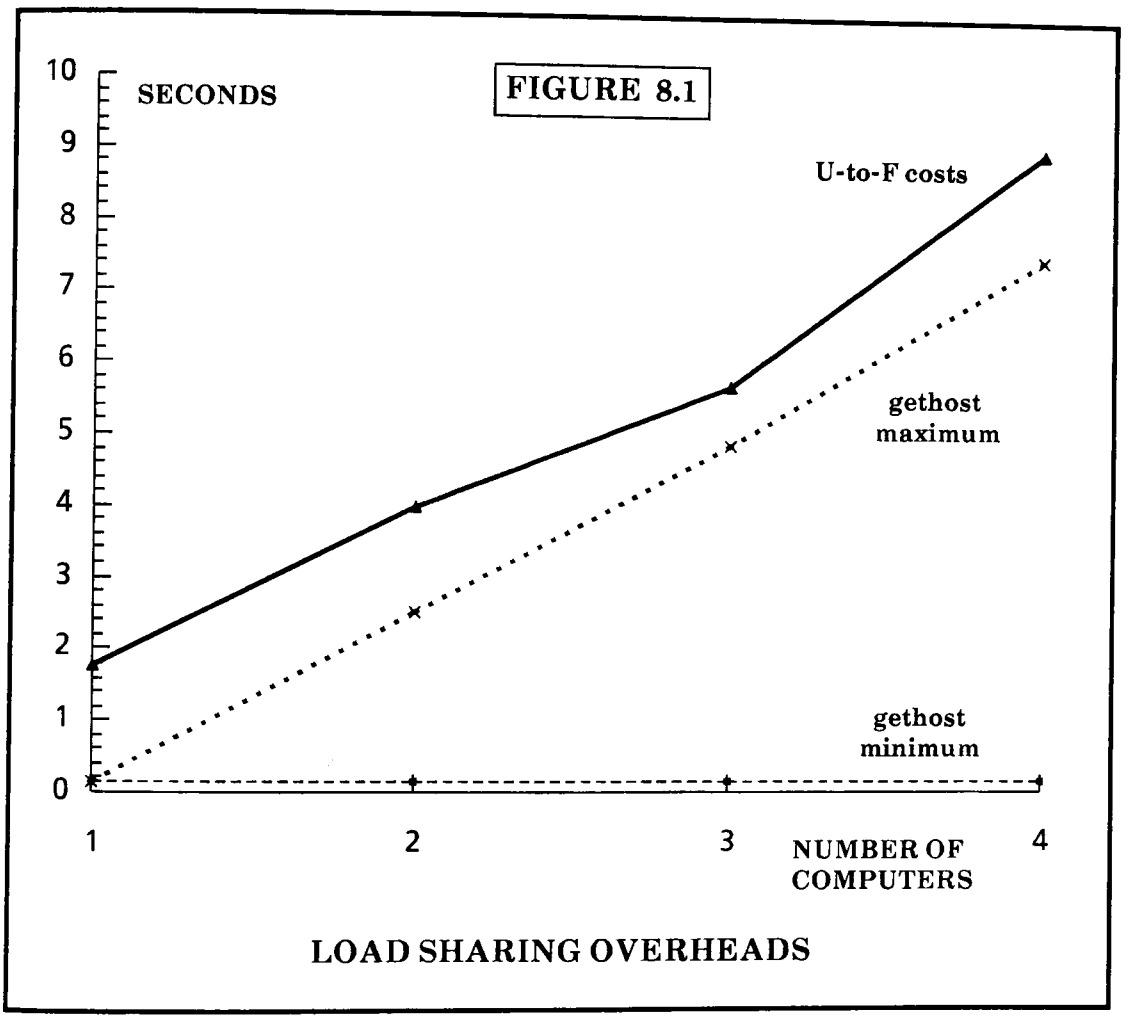
In this subsection we shall assess the load sharing scheme for the Perq system by observing the degree to which it possesses the desirable features of load sharing schemes as outlined in Chapter Three (section 3.2.8). We consider each feature separately:

I - **Stability:** The stability of a system can be measured with respect to several criteria. For example, a stable system would give an acceptable response time to its users in different system states; more relevant to the systems implementing a load sharing scheme is that a free computer should not become inundated with requests from the remote computers. There are some built-in features of our load sharing scheme for the Perq system that encourage stability. For example, confirmation of the current state of the server computer avoids tasks being sent to computers that have changed state since the last information exchange; the different order of storing the state information in the state vectors prevents the same free or hosting computer being selected by all the other computers. Furthermore a request to execute a program remotely (as a result of load sharing) is not further considered for load sharing by the receiving computer; thus preventing a request from moving among the computers without being executed at any computer.

II - **Implementability:** We have demonstrated that it is possible to implement our load sharing scheme without changing the existing system software. This was achieved because we decided to implement our scheme above the Newcastle Connection, and therefore utilised the existing mechanisms for remote program execution and information exchange. Therefore we can safely say that the implementation of our load sharing scheme is fairly simple, and does not require major changes to the existing structure of PNX or the Newcastle Connection.

III - **Cost:** The total overhead of the load sharing scheme for the Perq system can be divided into five distinct components. The first component of the cost is incurred when a program suitable for load sharing is invoked and it is determined whether the local computer is busy. The average time taken to do this, in the absence of any other user-level programs, is 1.94 seconds.

If the local computer is busy then the second component of the cost is contributed by the function (*gethost*) responsible for finding a free computer in the system. The minimum value of this cost is obtained when no free or hosting computer can be found in the local state vector; while the maximum value is obtained when the local state vector shows all the other computers to be free which are in fact all occupied. Figure 8.1 shows the minimum and maximum



values of these costs for different number of computers in the system. Note that the maximum cost of the second component varies almost linearly with the number of computers in the system, while the minimum cost remains constant at about 0.14 seconds.

If a free remote computer can be found then there are some additonal costs involved in updating local and remote state vectors, and setting the count of remotely executing requests. The total value of this third component of the overhead is approximately 11 seconds.

The fourth component of the overhead is incurred when a program is executed remotely. Our results in Chapter Seven have shown that the value of this component depends on the nature of program being executed remotely. The CPU intensive programs have less overhead than the I/O intensive programs.

The last and fifth component of the overhead is incurred when a state transition takes place on a computer, for example when a computer is switched ON all the other computers need to be informed. Figure 8.1 shows the times taken to inform different number of other computers in the system when the transition U to F takes place.

Thus we see that the cost of our of load sharing scheme, excluding the cost of remotely executing a ptrogram, can be as little as 1.94 seconds and as high as 20 seconds. It must be recalled, however, that our load sharing scheme avoids the expensive parts until it is clear that there is a good chance of finding a free computer, and the unnecessary exchange of system information is also avoided.

IV- **Autonomy:** The mechanism used by the Newcastle Connection (and therefore by the load sharing scheme) for invoking remote execution does not force the remote computers to execute tasks. The remote computers are free to either reject or accept the request made by the other computers. If the remote computer rejects the request then the load sharing scheme re-executes the command on the machine originally requested by the user. Note that the load

sharing scheme is not responsible for recovering from the mistakes in the original requests.; the load sharing scheme would only re-execute a command if it made changes to it.

v- **Transparency:** The load sharing scheme is transparent in that a user does not become aware that some of his commands are not executed where he intended.

vi- **Tunability:** The only explicit parameter that can be varied according to the changes in the system is the threshold value used to decide whether the local machine is busy or not. Some control is however implicit in the design of the scheme. For example, as more computers become occupied, there is less chance of a program being executed remotely. Once all the computers are occupied, the load sharing scheme does not find any free computers, and all the programs are executed locally. This is what is desired of a load sharing scheme: it is more active when there are large differences, and less active when there are less differences in the loads on the computers in the system. We can claim (rather optimistically) that our load sharing scheme is *self-tuning*.

8.2 Suggestions for Future work

Our implementation of the load sharing scheme for the Perq system is essentially a prototype that did not require any changes to the existing software. It would be interesting to measure the performance of our load sharing scheme using an implementation that makes changes (discussed in Chapter Six) to Unix and Newcastle Connection software.

The experiments described in Chapter Seven were designed to reveal the **potential** of our load sharing scheme and, therefore, rather extreme environments were used. A better idea of the benefits of our load sharing

scheme will be established if more realistic workloads were used for the experiments. For example, one could observe the usage of single-user machines over a given period to find out the typical command-mix and command-arrival rate in the system. It would then be possible to construct a workload that represents the real load.

In the design of our load sharing scheme we assumed that only the standard programs which already exist on the remote computers are considered for load sharing. It should, however, be interesting to investigate the gains of load sharing when a program is first copied to the remote computer (using the existing Newcastle Connection mechanisms) and then executed remotely.

Our results indicated that for real Unix programs load sharing was more useful when these programs accessed local temporary files. One could change other commonly used programs so that they always access local files. Furthermore, the load sharing scheme can be modified to take account of the location of I/O files before making a decision to execute a program remotely. This new scheme could then be evaluated.

8.3 EPILOGUE

By designing, implementing and experimenting with the load sharing scheme for the Perq system we have discovered several significant points. We confirmed that the benefit of our load sharing scheme depends on the nature of the program that gets executed remotely as a result of load sharing the system. In this respect the CPU intensive programs are well suited for remote execution. The I/O intensive and mixed programs are not suited for remote execution if most of their I/O files are resident on the local machine. However, large gains are to be made if load sharing results in executing a program where

its I/O files are residing. The standard Unix utility programs benefit from load sharing if not only the user-supplied I/O files are located where the program is to be executed but also the I/O files which the program uses itself are at its place of execution.

In this study we have demonstrated that it is possible to implement a load sharing scheme using the mechanisms provided by the Newcastle Connection. However the *connected server* facility had to be added to the Newcastle Connection for the load sharing scheme to work properly. If the load sharing scheme is to be modified to take account of the location of I/O files then the Newcastle Connection should provide system calls that can determine the location of I/O files. Furthermore if we are to load share requests which arrive directly from another computer (and not sent by the load sharing scheme) then the Newcastle Connection should allow a remotely executing program to remotely execute another program.

Looking back at our work, we have learnt a great deal about the issues involved in load sharing distributed computer systems, but have not been able to completely resolve these issues for the load sharing scheme to be beneficial in every situation. Our view is that there are no absolute solutions to the problems in load sharing distributed systems. The potential usage and the structure of the system dictate the solutions, and the resulting load sharing scheme is beneficial in certain situations. Whether load sharing is worthwhile will depend on how often these situations exist in the system.

REFERENCES

[ALONSO 83]

R. Alonso,
The Design of Load Balancing Systems For Local Area Network Based Distributions,
U.C. Berkeley Publications, Fall 1983.

[BACH 86]

Maurice J Bach,
The Design of the Unix Operating system,
Prentice Hall Software Series, 1986.
ISBN NO: 0132017997

[BANAHAN 82]

M. F. Banahan, A. Rutter,
Unix: The Book,
Sigma Technical Press, 1982.
ISBN No: 0905104218

[BARAK 85a]

Amnon Barak, Ami Litman,
MOS: A Multicomputer Distributed Operating System,
Software Practice and Experience, Volume 15(8), August, 1985.
Pages 725-737

[BARAK 85b]

Amnon Barak, Amnon Shiloh,
A Distributed Load-balancing Policy for a Multicomputer,
Software Practice and Experience, Volume 15(9), September, 1985.
Pages 901-913

[BERSHAD 86]

Brian Bershad,
Load Balancing With Maitre d',
;login: Volume 11 (1), January/February, 1986
pages 32-45

[BLACK 86]

J. P. Black,
The Architecture of UNIX United,
Technical Report Series No 220, August, 1986.
Computing Laboratory,
University of Newcastle upon Tyne.

[BLAIR 83]

Gordon S. Blair,
Distributed Operating System Structures for Local Area Network Based Systems,
PhD Thesis, 1983, University of Strathclyde.

[BOKHARI 79]

S. H. Bokhari,
Dual Processor Scheduling with Dynamic Reassignment,
IEEE Transactions of Software Engineering
Volume SE-5 No 4, July, 1979.
Pages 341-349

[BROWN 84]

P. J. Brown,
Starting with Unix
Addison-Wesley, 1984.
ISBN No: 0201132338

[BROWNBRIDGE 82]

D. R. Brownbridge, L. F. Marshall, B. Randell,
The Newcastle Connection - or UNIXes of the world unite
Software Practice and Experience
Volume 12(12), December, 1982.
Pages 1147-1162

[CAREY 86]

Michael J. Carey, Hongjun Lu,
Load Balancing in a Locally Distributed System
ACM Proceedings of SIGMOD '86 International Conference on management of Data, Washington D C, May 28-30, 1986.
Pages 108-119.

[CHOU 82]

Timothy C. K. Chou, Jacob A. Abraham,
Load Balancing in Distributed Systems
IEEE Transactions of Software Engineering, Volume SE-8, 1982.

[CHOW 79]

Yuan-cheih Chow, Walter H. Kohler,
Models for Dynamic Load balancing in a Heterogeneous Multiple Processor System
IEEE Transactions on Computers, Volume C-28 No: 5, May, 1979.
Pages 354-361

[CHU 80]

W. W. Chu, L. J. Holloway, M. T. Lan, Kemal Efe,
Task allocation in Distributed Data Processing,
IEEE Computer, November, 1980.
Pages 57-69

[DUNSMUIR 85]

M. R. M. Dunsmuir, G. J. Davies,
Programming the UNIX System,
Macmillan Computer Science Series, 1985.
ISBN No: 0333371569

[EAGER 84]

Derek L. Eager, Edward D. Lazowska, John Zahorjan,
Dynamic Load Sharing in Homogenous Distributed Systems
Technical Report 84-10-01, October, 1984,
Department of Computer Science,
University of Washington

[EAGER 85]

Derek L. Eager, Edward D. Lazowska, John Zahorjan,
A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing,
Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems, August, 1985.
Performance Evaluation Review, Volume 13 No: 2.
Pages 1-3

[ENSLOW 78]

Philip H. Enslow,
What is a Distributed Processing System?
Computer, January, 1978.

[FERRARI 83]

D. Ferrari, G. Serazzi, A. Zeigner,
Measurement and Tuning of Computer Systems
Prentice Hall, 1983.

[FERRARI 86]

Domenico Ferrari, Songnian Zhou,
A study of Load Indices for Load Balancing Schemes
1986 Proceedings of Fall Joint Conference, Dallas, Texas.
November 2-6, 1986. IEEE Computer Society Press.
Pages 684-690

[FOXLEY 85]

Eric Foxley,
Unix for super Users
International Computer Science Series, 1985.
ISBN No: 0201142287

[HAC 85]

Anna Hac, Theodore Johnson
A Study of Dynamic Load Balancing in a Distributed System
Report No: JHU/EECS-85/15, 1985.
Department of Electrical Engineering & Computer science
The Johns Hopkins University.

[HARARY 69]

F. Harary,
Graph Theory,
Addison Wesley, New York, 1969.

[ICL 84]

ICL PERQ: Guide to PNX
R10139/00
International Computers Limited
January, 1984

[JENSEN 78]

Douglas E. Jensen
The Honeywell Distrbuted System
Computer, January, 1978.

[JONES 79]

A. K. Jones et al,
StarOS, a Multiprocessor Operating system for the support of Task Forces
Proceedings of the Seventh ACM Symposium on Operating System Principles, Pacific Grove, Calfornia, December, 1979.
Pages 117-127

[KARSHMER 83]

A. Karshmer, D.DePree, J. Phelan,
The New Mexico State University Ring Star System: A Distributed UNIX Environment
Software Practice and Experience,
Volume 13(12), December, 1983.
Pages 1157-1168.

[KAUFELD 82]

J. C. Kaufeld, D. L. Russel
Distributed Unix System,
Workshop on Fundamental Issues in Distributed Computing,
ACM SIGOPS and SIGPLAN,
15-17, December, 1982.

[KEEFE 85]

D. Keefe, G. M. Tomlinson, I. C. Wand, A. J. Wellings,
PULSE - An ADA-based Distributed Operating system,
Academic Press, 1985.

[KELLEY 84]

A. L. Kelley, Ira Pohl,
A Book on C,
1984.
ISBN No: 0805368604

[KERNIGHAN 78]

Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language
Prentice Hall Software Series, 1978.

[LEFFLER 83]

Samuel J. Leffler, Robert S. Fabry, William N. Joy,
A 4.2bsd Interprocess Communication Primer (Draft of July 27, 1983)
Computer Systems Research Group, Department of Electrical
Engineering & Computer Science
University of California, Berkeley.

[LELAND 86]

Will E. Leland, Teunis J. Ott,
Load-balancing Heuristics and Process Behavior
ACM Performance Evaluation Review
Special Issue Vol: 14 No: 1, May, 1986
Pages 54-69

[LELANN 81]

Gerard Lelann,
Motivations, Objectives, and Charescterisation of Distributed Systems,
Lecture Notes in Computer Science (105),
Edited by B. W. Lampson, M. Paul, & H. J. Siegert,
Springer Verlag, 1981.

[LIVNY 82]

Miron Livny, Myron Melman
Load Balancing in Homogeneous Broadcast Distributed Systems
Department of Applied Mathematics
The Weizmann Institute of Science, Rehovot, Israel.

[LO 83]

Virginia Mary Lo,
Task Assignment in Distributed Systems,
PhD Thesis, October, 1983,
Univesity of Illinois.

[LO 84]

Virginia Mary Lo,
Heuristic Algorithms for Task Assignment in Distributed Systems,
Proceedings of 4th International Conference on Distributed Computing, 1984.
Pages 30-39

[MARSHALL 87]

Lindsay F. Marshall
Robert J. Stroud
Personal Communication
Computing Laboratory, The University of Newcastle upon Tyne
March, 1987

[METCALFE 85]

R. M. Metcalfe, D. R. Boggs,
Ethernet: Distributed Packet Switching for Local Computer Networks,
Comms. A. C. M., Volume 19(7), July 1976.
Pages 395- 404

[NARSINGH 74]

Deo Narsingh,
Graph Theory with applications to Engineering and Computer Science
Prentice Hall, 1974.

[NEEDHAM 80]

R. M. Needham, A. J. Herbert,
The Cambridge Distributed Computing System
Addison-Wesley, 1980.

[NI 81]

Lionel M Ni, Kai Hwang,
Optimal Load Balancing strategies for a Multiple Processor System
Proceedings of the 1981 International Conference on Parallel Processing.
Pages 352-357

[NI 85]

Lionel M. Ni, Chong-Wei Xu, Thomas B. Gendreau,
A Disributed Drafting Algorithm for Load Balancing,
IEEE Transactions of Software Engineering, Volume SE-11 (10), October, 1985.
Pages 1153-1161

[NOWITZ 80]

D. A. Nowitz, M. E. Lesk,
Implementation of a Dial Up Network of Unix Systems
Proceedings of the COMPCON 80, Washington D C, September, 1980.
Pages 483-486.

[POPEK 83]

G. Popek, B. walker, et al.,
LOCUS A Network transparent, High Reliability Distributed system
Proceedings of Eigth ACM Symposium on Operating Systems Principles, Pacific Grove, California, December, 1983.
Pages 169-177

[PRICE 84]

Camille C. Price, S. Krishanaprasad,
Software Allocation Models for Distributed Computing Systems
Proceedings of 4th International Conference on distributed Computing, 1984.
Pages 40-48

[RAMAMRITHAM 84]

Krithivasan Ramamritham, John A. Stankovic
Dynamic Load Scheduling in Distributed Hard Real Time Systems
Proceedings of 4th International Conference on Distributed computing, 1984.
Pages 96-107

[RANDELL 84]

B. Randell,
The Newcastle Connection: A Software subsystem for Constructing Distributed UNIX Systems,
Technical Report Series 194, September, 1984,
Computing Laboratory,
University of Newcastle upon Tyne.

[RANDELL 85]

B. Randell
System Design and Structuring,
Technical Report Series No: 198, March 1985,
Computing Laboratory,
University of Newcastle upon Tyne.

[RAO 79]

G. S. Rao, et al,
Assignment of Tasks in a Distributed Processing System with Limited Memory
IEEE Transactions on Computers, Volume C-28 No: 4, April 1979
Pages 291-299

[RITCHIE 78]

D. M. Ritchie, et al,
Unix Time-Sharing System
Bell Systems Technical Journal, Part 2, Vol 57, No: 6,
July-August 1978.

[RITCHIE 84]

D. M. Ritchie, et al,
The Unix System,
AT & T Bell Laborotories Technical Journal, part 2, Vol 63, No: 8
October, 1984.

[ROWE 82]

L. Rowe, K. Birman,
A Local Network Based on the UNIX Operating System,
IEEE Transactions of Software Engineering,
Volume SE-8(2), March, 1987.
Pages 137-146.

[SHILOH 83]

A. Shiloh
Load sharing in a Distributed Operating System,
M. S Thesis, July, 1983,
Department of Computer Science, The hebrew University of
Jerusalem, 91904, Israel.

[SHRIVASTAVA 82]

S. K. Shrivastava, F. Panzieri,
The Design of a Reliable Remote Procedure Call Mechanism,
IEEE Transactions on computers, July, 1982.

[STANKOVIC 85]

John A. Stankovic,
Stability and Distributed Scheduling Algorithms,
IEEE Transactions of Software Engineering, Vol SE-11, No: 10,
October, 1985.
Pages 1141-1152

[STONE 77]

H. S. Stone,
Multiprocessor Scheduling with the aid of Network Flow Algorithms
IEEE Transactions of Software Engineering, Vol SE-3 No 1, Jan, 1977.
Pages 85-93

[STROUD 83]

Robert J. Stroud,
Installing the Newcastle Connection on a Perq (a study in paranoia) or How the Butterflies finally came home to roost
SRM 350, July, 1983,
Computing Laboratory,
The University of Newcastle upon Tyne.

[STROUD 86]

Robert J. Stroud,
Recursive Transparency OR Just how transparent is the Connection anyway?
SRM 431, June, 1986,
Computing Laboratory,
The University of Newcastle upon Tyne.

[STROUD 87]

Robert J. Stroud,
Naming Issues in the Design of Transparently Distributed Operating Systems.
PhD Thesis, July 1987.
Computing Laboratory,
The University of Newcastle upon Tyne.

[TRIPATHI 80]

Anand R. Tripathi, Edwin T. Upchurch, Jones C. Browne
An Overview of Research Directions in Distributed Processing
Proceedings of Distributed Computing, Washington D C, Sept 23-25, 1980, COMPCON 80.
Pages 333-340

[WANG 85]

Yung-terng Wang, Robert J. T. Morris,
Load sharing in Distributed Systems
IEEE transactions on Computers, Vol C-34, March, 1985.
Pages 204-217

[WU 80]

S. B. Wu, M. T. Liu,
Assignment of Tasks and Resources for Distributed Processing
IEEE COMPCON Proceedings on Distributed Processing, Fall 1980.
Pages 655-662

[YU 86]

Philip S. Yu, Simonetta Balsamo, Yan-Hang Lee,
Dynamic Load Sharing in Distributed Database Systems
Proceedings of Fall Joint Computer Conference, Dallas, Texas,
November 2-6, 1986,
IEEE Computer Society Press.
Pages 675-683

[ZATTI 85]

Stefanno Zatti,
A Multivariable Information Scheme to Balance the Load in a Distributed System
Report No: UCB/CSD 85/234, May 1985.
Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720

[ZHAO 85]

W. Zhao, K. Ramamritham,
Distributed Scheduling Using Bidding and Focussed Addressing
IEEE Proceedings of Real-Time Systems Symposium, December, 3-6 1985.
Pages 103-111