Dynamic Load Balancing

by

Christopher W. Clifton

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Master of Science
and
Bachelor of Science

at the

Massachusetts Institute of Technology

June 1986

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
June 17, 1986

Certified by [illegible]
Nancy A. Lynch
Thesis Supervisor

Certified by [illegible]
Flaviu Cristian
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Dynamic Load Balancing

by

Christopher W. Clifton

Submitted to the Department of
Electrical Engineering and Computer Science
on June 17, 1986 in Partial Fulfillment
of the Requirements for the Degrees of
Master of Science and Bachelor of Science

Abstract

Most computer systems are expected to perform a variety of tasks simultaneously. In distributed systems the workload may be spread across multiple processors. To do this, decisions must be made as to where to run each task. Some processors may be better suited to certain tasks than others. This thesis develops a model for load which takes into account different task and processor characteristics. This model is then used to describe the load balancing problem. A distributed system is implemented which uses this model to perform load balancing.

Thesis Supervisors: Dr. Nancy Lynch
Title: Professor of Computer Science
Dr. Flaviu Cristian
Title: Research Staff, IBM Almaden Research Center

Table of Contents

Table of Figures

Table of Tables

Chapter One

Introduction

A large portion of computer science research is today directed toward *distributed systems.* These systems allow a number of individual computers to communicate, cooperate and share resources in performing work. There are many kinds of distributed systems, ranging from geographically distributed communication networks to tightly coupled multiprocessors. Each of these provides certain advantages over a single system. A geographically distributed network is normally used primarily to allow sharing of ideas and data. A tightly coupled system, on the other hand, allows a higher processing throughput than a single processor.

1.1 Loosely Coupled Systems

I am particularly interested in two of the advantages often ascribed to distributed systems: the increased throughput from performing tasks concurrently, and the increased availability of having extra processors available in case some fail.

The second goal requires that the processors be independent enough that failure of one does not preclude operation of the others. However, they must be similar enough that the tasks running at one processor can be run on another if the first fails. As a result, I will be concentrating on what are often referred to as *loosely coupled* distributed systems. These consist of processors which share access to some peripherals such as printers and terminals, as well as mass storage, so that a task is not tied to a particular processor. The majority of communication is via a network. This allows isolation of a failing processor (as opposed to a system with shared memory, where a failing processor might modify memory used by another processor.)

1.2 Need for Load Balancing

Having such a system does not automatically guarantee that we will have achieved these goals. A number of problems remain, such as how to accomplish the necessary communication and monitoring of tasks in order to prevent conflict. The problem which interests me, often referred to as *load balancing*, is how to assign the tasks which need to be accomplished to the processors available. In particular, I would like to do this assignment in a *heterogeneous*[1]environment. I would like to do this without assuming significant a priori knowledge about either the processors or the tasks to be run. As a result, part of the problem will be in determining dynamically what the characteristics of different tasks and processors are, in order to make a good assignment.

1.3 Use of Load Balancing

There are three circumstances in which I feel load balancing is important. The first, *placement*, is deciding where to run a newly created task. Load balancing can also be useful at other times. I break these into two categories, *redistribution* and *changing use.*

1.3.1 Redistribution

Unplanned failure of a processor in effect creates a number of new tasks to be run. The scenario is slightly different from that of a single new task. With a single task, the placement is determined based only on the task and the current state of the processors. With a number of tasks, the placement could depend not only on the task to be placed and the state of the processors. The placement decisions of all of

[1]I am using *heterogeneous* to refer to an environment in which not all of the processors are the same. In particular, this means that some tasks may run better on some processors than on others.

the newly orphaned tasks are interrelated. Simply using the current state of the processors to place all of the orphaned tasks would result in all of the tasks being placed at the least loaded processor. This would immediately overload that processor. This results in a significantly more complex problem, and yet the gains that could result from good load balancing are correspondingly greater. A method used for redistribution would also be useful in the case of a planned shutdown, although the prior notice would allow other options, such as gradually moving tasks from the processor to be shut down. This would make maintenance less of a burden on the users of the system, as they need notice only a slight decrease in the system performance.

A similar situation occurs when a processor is restarted, or a new node is added to the system. Although there is no emergency which must be handled, load balancing is still called for in order to make use of the new processing power available (and ease the load on everything else.)

A third situation, which in a way combines the above two, is what to do in the case of a processor which is overloaded. This is not quite as bad as a failure, as processing still continues, but load balancing could result in tasks completing much sooner.

1.3.2 Changing Use

In some cases, the load on the system may become unbalanced even though all of the tasks were originally placed at the best locations, and no processors have failed. This is a result of changes in the use of tasks. This could be happen as a result of a task completing, leaving a lightly-loaded processor. Some tasks, such as mailer programs or data base managers, run continuously. These tasks act as *servers.* The characteristics of these tasks may change over time, as more (or less) demand is made for their services. Due to these changes, it may be desirable to move already running tasks in order to improve the overall system performance.

This case introduces a problem that was not present in the first two cases. There is normally some expense associated with moving a task. This could be increased use of the network, extra shutdown and startup expense for the tasks moved, or simply the time lost while the task is being moved. As a result, a trade-off must be made between the cost of moving tasks and the eventual gains in system performance from balancing the load.

1.4 Prior Work

There has already been considerable research done in the area of load balancing. Much of this work has been directed toward determining an optimum *static* allocation. The static load balancing problem assumes that there are a number of tasks to be placed in an empty system. I believe that this may be a mathematically interesting problem, but most computer systems do not have the property that all tasks start simultaneously and run to completion before new tasks appear. This is actually a special case of redistribution, and as such serves as interesting background.

There has been work done in *dynamic* load balancing. Dynamic load balancing is used to refer to two problems: allocation of new tasks in a system which is already in use, and moving tasks which are already running in order to redistribute load. There has been some good algorithmic work, but it relies on assumptions about system load which I feel to be unrealistic. This results in difficulty in actually applying this work to real systems.

1.4.1 Static Balancing

Much of the early work in static load balancing concentrated on two-processor or homogeneous multiprocessor systems [Bokhari 79] [Stone 78]. In addition, the tasks to be distributed were assumed to have certain predetermined and unchanging requirements for communication and processing. Usually, each task would have some known communication with each other task, and the object of load balancing was to minimize these communication costs. This resulted in several methods for reducing load balancing to a linear programming [Hofri 78] or a network flow [Stone 77] problem. Solutions to optimally partitioning networks have been known for some time [Ford-Fulkerson 62]. This work, while mathematically interesting, assumes an idealized view of computer systems. Today's distributed systems are likely to be composed of many processors of varying types, e.g. MIT's Common System project [Clark 85]. The problem becomes much more complex, and these solutions lose much of their usefulness.

Later research has filled in some of these gaps. Such special cases as tasks only running on certain processors [Rao 79], costs of file storage as well as CPU use, [Morgan 77], and non-homogeneous systems [Bokhari 81] [Chow 79] have been explored. These solutions use a number of different techniques, making them difficult to compare and combine. All of this work shares a common assumption, which is that all tasks are started simultaneously. This may be useful in a batch-oriented system, but is difficult to apply to on-line systems where tasks may start at any time. In addition, this static work does not take into account changes in system configuration. I feel that a significant contribution of load balancing should be in reasonable handling of failures and other downtime.

1.4.2 Dynamic Balancing

More recently, work has been done on the question of dynamic reassignment [Kar 84]. This does help us in dealing with on-line systems. The problem of task reassignment in the event of processor failure has also been investigated [Chou 83]. However, little work has been done which discusses tasks with characteristics which change over time. There has been some work done which does not assume this *a priori* knowledge of task requirements [Stankovic 85]. This provides some useful heuristics for load balancing. However, this work does not discuss the problem of how to determine the requirements of tasks, and it ignores the redistribution problem.

1.5 Overview

This thesis concentrates on modeling system load rather than on algorithms to perform load balancing. In addition, a system was built that monitors load and uses the information gathered in load balancing decisions. Although the model for load is applicable to a variety of systems, all of the work has been influenced by the environment in which I built the test system. Therefore I will describe this environment in Chapter 2.

Chapter 3 gives a formal definition of load balancing, and also identifies certain useful subproblems (such as the static balancing problem mentioned above.) Chapters 4 and 5 formally introduce the model for system load, and discuss how to characterize real systems using this model. This lays the groundwork for developing load balancing algorithms. Methods of evaluating load balancing algorithms are discussed in chapter 6. This chapter also introduces a particular variant of the load balancing problem which is shown to be NP-hard. Chapter 7 presents the algorithm which I use for load balancing.

The remaining chapters describe the design and implementation of a load balancer. Chapter 8 discusses problems of distributed control of the load balancer. Existing solutions to similar problems are compared, and the one actually used is described in detail. Chapter 9 describes the techniques used to determine load and the actual implementation of the load balancing. Chapter 10 presents some actual results of the load balancer.

Chapter Two

Background and Environment

The direction of my work has been heavily influenced by the environment in which it was done. As a result, I will first discuss this environment, in order to give a better understanding of what I have accomplished. This thesis was started while I was an intern in the Computer Science department at the I.B.M. San Jose Research Lab. A more complete description of this project is available in some of the papers published by this group [Aghili 83], but I will summarize here.

2.1 Highly Available Systems Project

The Highly Available Systems (H.A.S.) Project at the IBM San Jose Research Lab was started to investigate the use of a loosely coupled network of medium to large computers in order to provide a highly available database system. The project has since expanded its goals to providing high availability of all computing resources using distributed systems. The basic premise of this research is that given a number of processors, a system can be built which will recover from the failure of one or more (although not all) processors without human intervention. In addition, the system should be able to make productive use of all running processors at all times (as opposed to having idle backup processors.) This leads to many interesting areas of study, such as providing a communications system which has predictable behavior even when parts of the system fail, detecting failure of tasks and processors, moving tasks from processor to processor, and load balancing.

2.1.1 Communication subsystem

An important part of any distributed system is its means of communication. The project has investigated a number of ways of achieving fault-tolerant communications. One concept that has proven useful is that of *atomic broadcast* [Cristian 85a]. An atomic broadcast provides three guarantees:

1. Either all or none of the intended recipients will receive the broadcast.

2. All messages will be received in the same order at all sites.

3. Messages will be delivered within a known time bound δ (or will not be delivered at all.)

The difficulty is in guaranteeing this in the presence of failure. It has been shown that this cannot be done in general [Fischer 83], but it is possible given some restrictions on the allowable failures [Strong 85].

This capability in a communication system simplifies the problem of distributed control, as communication failures can almost be ignored. I make use of this communication primitive in order to simplify distributed decision handling in my load balancing system. This will be discussed in more detail later, in Section 8.3.

2.1.2 Resource management

Another part of the project involves monitoring the status of tasks in order to insure that all required services are available. This is done through the *Auditor* subsystem [Aghili 83]. The auditor is responsible for restarting tasks in the event of failure. This is an obvious place to make use of load balancing. As a result, my design and implementation was oriented towards the same type of tasks as the auditor.

2.2 Base Operating System

All of the prototyping of this research has taken place on top of IBM's VM/SP operating system. This operating system is based on the concept of a *virtual machine*, a simulated single-user processor. Each task runs in a separate virtual machine. This gives a clear level of granularity in regards to load balancing decisions. A task is not tied to a particular virtual machine. This allows the task to be moved to virtual machines on different physical processors. Load balancing decisions consist of choosing which physical machine a task should run on.

In some cases, tasks may only be able to run on certain physical machines. This could happen if, for example, only one or two of the available physical machines had access to mass storage required by the task. I do take this into account, and allow for restrictions on the freedom of my load balancing decisions. Without some freedom, however, load balancing is no longer an interesting question. Multiple paths to disks or system-wide file servers can be used to enable the processor independence necessary in order to use load balancing to best advantage.

2.3 Influence of H.A.S. Environment on this Work.

The tasks of most interest in this system are continuously running services, such as database managers, mail servers, or file servers. As a result, long-term usage patterns for these tasks can be developed and used in load balancing decisions. I take advantage of this assumption, and integrate monitoring of both tasks and processors into my load balancing. This is not to say, however, that this work is irrelevant to other types of tasks. Such tasks as compilers and text formatters may not run continuously, but their load characteristics should be similar from run to run. This does vary with input, but monitoring and averaging over individual runs can give figures which can be used in much the same way as those from continuously running tasks.

The result is that this research assumes that tasks are stable enough that some characterization of them can be developed. This can be either through continuous observation, or through monitoring of a number of individual runs. This partly defines the granularity of what is considered a task, as it must be possible to independently monitor tasks.

Chapter Three

Definition of Load Balancing

Load balancing involves choosing an assignment of tasks to processors. We start with a set T of tasks and a set P of processors, and determine a configuration of the system where some subset of the tasks in T are running on some subset of the processors in P. Sometimes the system may already have certain tasks running at certain processors. In some cases these may be moved, in others they may be fixed. Note that we do not necessarily have to run all of the tasks, or use all of the processors. In some situations we may want to get all of the tasks done as quickly as possible; in others we may want to get certain tasks done within a certain time, and the rest of the tasks can wait.

Load balancing, then, is used to determine the *configuration* of the system:

> **Definition 3-1:** A **configuration** C is a set of pairs (T,P) where each T is a task, P is a processor, and there is at most one pair for each task $T \in C$. $(T,P) \in C$ means that T is running on processor P in the given configuration.

Load balancing can be defined as a *function* which maps a set of tasks to be run onto this pairing of tasks to processors. The following definitions give several *classes* of load balancing functions. The first is the general case.

> **Definition 3-2:** A **load balancing** function takes a configuration C and a set S of tasks (where any task $T \in S$ is not in C), and returns a new configuration C' containing at most one pair corresponding to each task in S and each task in C.

Note that this function is not limited to choosing where to place the tasks in S. New tasks can be assigned to processors, tasks already running in the system can be reassigned to new processors, and some tasks may not be assigned to any processor.

This definition tells us what qualifies as a load balancing function, but does not specify what the function should accomplish. The goal of load balancing is to improve the performance of the system. This is done by distributing the load imposed by the tasks across the processors available. This load is different from system to system, and can be quite complex. A model for load is given in chapters 4 and 5. The goal of load balancing can likewise vary from system to system. This can make load balancing a computationally difficult problem, as shown in section 6.1.

In many cases, the power of the general load balancing function is not necessary. There are special cases of load balancing problems which are often useful. It may be easier to develop efficient algorithms which give an assignment meeting the desired goal for these cases than for the general load balancing problem. I refer to these special cases as **initial distribution**, **task placement**, and **redistribution**.

Definition 3-3: An **initial distribution** function takes a set S of tasks and a set $\mathcal{P}$ of processors, and returns a configuration C, where for each pair $(T, P) \in C$, $T \in S$ and $P \in \mathcal{P}$.

This performs an initial assignment of tasks to processors in an empty system, otherwise known as *static* load balancing.

Dynamic load balancing is used to refer to two problems. These are described in the following definitions.

Definition 3-4: A **task placement** function takes a task and a configuration, and returns a processor on which to place the task (or null, specifying that the task is not put on any processor.)

A placement function is useful when adding a single task to an already running system.

Definition 3-5: A **redistribution** function takes a configuration C and returns a new configuration C' where each task $T \in C'$ is in C.

Such a function can be used to even the load in an already running system. This is useful when a task changes its running characteristics, due, for example, to an increase in the number of people making requests of that task.

Chapter Four

Model for System Load

In order to decide which tasks to assign to which processors, we must be able to evaluate potential choices. One way to do this is to compare the processing resources needed by a task with the resources provided by a processor. To do this, we need a general method to:

1. Characterize the capacities of a processor. These are the total resources that the processor can provide.

2. Characterize the requirements of a task. These are the resources used by the task, and may vary depending on the state of the system.

3. Relate these two.

This would give the information necessary to make decisions, except that we do not yet have a *goal.* I see the end goal of load balancing as minimizing the **response time** of the system. This is the time from when a request is submitted until the result is returned. Not only is this what a user really means when asking how fast a system is, but it would seem easy to measure: simply start a task, and measure the time it takes to complete. However, general-purpose computer systems are capable of handling a variety of tasks, which may have different running characteristics. Which do we choose in order to measure the response time?

It would seem reasonable to choose some task as a "standard" with which to measure response time. For example, some commonly used system command could be timed each time it was called, and this would be used as the system response time. Alternatively, a special program could be written which would be run each time the response time of the system was needed. The time required for this special program to run would be the response time for the system.

If the response times of all tasks were directly proportional to this "standard task", this would be a useful measure of system performance. However, tasks may vary in the kind of processing required. One task could need a lot of CPU time while another requires the use of a communication channel. This difference in the kind of *processing resources* required by a task can result in the "standard task" not actually reflecting the performance of other tasks on the processor. For example, the "standard task" could be CPU dependent, and run very slowly in a system with an overloaded CPU, but an I/O intensive task would still run quickly. There has been work done which takes these differences into account [Kuck 78], but with an emphasis on performance analysis, not load balancing.

Many time-sharing operating systems will allow tasks using different kinds of processing to run concurrently. For example, while one task is waiting on I/O, another can use the CPU. In this way, running two tasks simultaneously will result in better response times than running them sequentially. However, two tasks which compete for the same resource will not gain by running concurrently. They will have to share the resource, resulting in slower response. Tasks which do not need to use that resource, however, will still run as quickly as before. Load balancing algorithms can exploit this concurrency if they are able to consider different processing resources separately.

Considerations like these make characterizing the load on a system more difficult. A "standard task" will actually only reflect the response times for tasks requiring a similar set of resources. Minimizing response time is still a good goal, but the load on a system can not be measured using a single task, or for that matter, any single value. To adequately characterize load it is necessary to look at a variety of factors.

4.1 Load as a Resource Vector

In order to capture this variety of processing types, I will represent processing resources using a *vector*. The components of this vector correspond to amounts of particular processing resources, e.g., CPU cycles/second, memory, communication channels, etc. The processing resources actually used as the components of the vector are based on the system under consideration. For example, in a system with very high speed communication channels, communication costs may be insignificant enough to be ignored in load balancing. These *resource vectors* will be used to describe both the capacity of individual processors, and the resources used by individual tasks.

Definition 4-1: A **Resource vector** $R = (r_1, \ldots, r_n)$ is a length n vector of non-negative reals. $R[i]$ is used to refer to resource r_i.

This relates to processors in the following manner. Each processor has an associated resource vector R where each component $R[i]$ of the vector characterizes the processor's capacity for resource r_i. Typical values for these components would be the number of instructions per second of the CPU, or the total amount of memory attached to the processor.

Definition 4-2: The **capacity** of a processor P is a resource vector $capacity(P)$, where the the i^{th} entry in the vector denotes P's capacity for resource r_i.

This leads to a method of describing of a task using a vector of *requirements*, where each component of the vector refers to the amount of a particular resource used by the task. For example, one component of the vector for a processor may refer to how many instructions can be processed per second. For a task, this component of the vector would give how many instructions the task executes per second. This is not sufficient, however, as the running characteristics of a task are dependent on its environment. For example, a task which accesses a network may spend more time waiting for data from the network at a processor with a slow communication link than at one with a fast link. Since it is spending more time waiting, it will use less of the CPU. The requirements of a task may change depending on the state of the rest of the system.

The requirements of a task can be characterized using a resource vector when the system is stable. In fact, for each possible configuration of the system there may be a different resource vector which characterizes the task. This leads to characterizing a task using a *set* of resource vectors. In any given configuration, one of these resource vectors will actually correspond to the resources used by the task. This vector is referred to as the *current requirements vector.*

Definition 4-3: The **requirements** of a task T is a set of resource vectors $requirements(T)$. Each vector $R \in requirements(T)$ represents a possible pattern of resource usage by T, where the component $R[i]$ of the vector corresponds to the amount of resource r_i used when R is the *current requirements vector* for the task.

In order to make use of this set we must be able to determine which of the resource vectors is the *current requirements vector* in a given configuration. In general, this is done by taking a profile of the entire system.

Definition 4-4: Profile(C) is a function which takes a configuration C and returns a set of (T,R) pairs, where for each task $T \in C$, $R \in requirements(T)$ is the corresponding *current requirements vector.*

These functions give us the necessary information to completely characterize a system.

Definition 4-5: A system is composed of

P = $\{P\}$, a fixed set of *processors*,
T = $\{T\}$, a fixed set of *tasks*,
R, a fixed, totally ordered set of n *resources*,
where r_i denotes the ith resource in the set, and
capacity, requirements, and *profile* functions.

These functions are quite general. As stated, any change in the configuration could affect every task. In some cases it would be nice to determine the status of the tasks on a particular processor without knowing the entire system configuration. This can be done if the status of a task only depends on the local part of the configuration.

Definition 4-6: Status(P,S) is a function which takes a processor and a set S of tasks, and returns a set of (T,R) pairs corresponding to each task $T \in S$, where $R \in requirements(T)$ corresponds to the *current requirements* vector for T when every task $T \in S$ is running on P.

Note: This function is only valid if it can be defined such that **status**(P,S)$\subseteq$**profile**(C), where $(T,P) \in C \Leftrightarrow T \in S$.

There is an invariant on the relation of tasks and processors which must be satisfied by **profile** (and therefore by **status**.) This is that the tasks running on a processor will never use more than that processor's capacity of any resource.

For any processor P and resource i, (4-1)

Let $\mathfrak{R} = \{R\}$ such that $(T,P) \in C$ and $(T,R) \in profile(C)$. Then

$$\sum_{R \in \mathfrak{R}} R[i] \leq capacity(P)[i]$$

Another way to describe this invariant is to use an **availability** vector. This vector corresponds to the amount of resources a processor has which are not in use. The above equation states that the availability of any resource cannot be negative.

Definition 4-7: Availability(P,C) is a resource vector corresponding to the unused capacity of P running in configuration C. It is defined such that:

Let $\mathfrak{R} = \{R\}$ such that $(T,P) \in C$ and $(T,R) \in profile(C)$. Then

$$Availability(P,C)[i] = capacity(P)[i] - \sum_{R \in \mathfrak{R}} R[i]$$

This gives the information necessary to make load balancing decisions. The next step is to define the *goal* of load balancing.

4.2 Quality

In order to state the goal of load balancing, I use a concept of *task quality*. This is a number which gives a measure of how well a task is running, based on which vector is the *current requirements vector*. This will normally be related to the response time of the task, although other goals could be chosen. Desirable current requirements vectors will have high qualities, and undesirable current requirements vectors will have low qualities.

> **Definition 4-8:** A **task quality** function takes a task T and a resource vector R (where $R \in \mathit{requirements}(T)$), and returns a non-negative number.

Such a function is effectively a ranking of the resource vectors in $\mathit{requirements}(T)$. The resource vector which corresponds to a task having all of the resources it can possibly use will have the highest quality, and one which corresponds to the task not being able to run at all will have quality 0.

The real goal of load balancing is to improve the *overall* response time of the system. The task quality function gives us a measure by which to judge individual tasks. We now need a **configuration quality** function which uses all of the individual task qualities to give a measure of how well the entire system is running.

> **Definition 4-9:** A **configuration quality** function takes a set of (*task, task quality*) pairs and returns a real number.

A configuration quality function is what is actually used to evaluate load balancing decisions. A simple configuration quality function would be to just average the task qualities; however, it may be desirable to assign priorities to certain tasks. Thus a typical configuration quality function would be a weighted average of the task qualities.

Note that these functions take no notice of *where* tasks are running, only *how* they are running. This is a desirable feature, as the goal of load balancing is to place tasks so as to increase the speed of the system. The location of tasks is only interesting in that it affects the speed of the individual tasks, but this is reflected in the current requirements vector. The task quality, and thus the configuration quality, are based on the current requirements vector. Therefore the quality can only reflect the location of tasks as it relates to the speed of the system, which is exactly what is desired for load balancing.

Chapter Five

Relation of Model to Actual Systems

The main goal of any model is to help us to understand the real world. In particular, this model is intended to be useful in performing load balancing in real computer systems. It provides a means of dividing load balancing into two problems: characterizing the system, and developing algorithms for load balancing. I will discuss the former problem in this chapter, and the latter in chapter 7.

I will try to describe general techniques for using this model to characterize systems. Section 9.1 discusses an actual implementation based on this model.

There are a number of factors which contribute to the load on a processor. Some of these have been mentioned (CPU, I/O) as justification for the model. Here is a more comprehensive list

- Processor use (there may be a variety of these, such as specialized numeric processors.)
- Memory (again, this is not necessarily a single item. Large systems often have caches or various speeds of memory.)
- Storage devices (Disk, Tape)
- Interprocess communications
- Peripherals (i.e., printers)

It is important to remember that this list is by no means complete; as computer systems grow and develop, new factors may appear.

It would be impossible to list all of the factors that contribute to load, and describe here exactly how to characterize each of them. This is a task which must be performed separately for each system. It is possible, however, to develop some general techniques to use in describing a system. To do this, I divide these factors into two classes, **shared** and **dedicated** resources. A **shared** resource is one which can service multiple tasks simultaneously. An example would be a processor in a time-sharing system. **Dedicated** resources are those that are tied to a single task, such as a tape drive.

In the above examples, the difference between shared and dedicated resources is in how long the resource is tied to a particular task. A time-shared CPU will normally spend less than a second on each task. If the time required to move the task to a different processor is shorter than the wait for this one, such as in a shared-memory system, processors would be considered a dedicated resource. However, in a loosely-coupled system the time required to move a task will be longer. In these systems, a CPU would be shared. The decision as to which resources are dedicated and which are shared may vary considerably between systems, but the handling of each of the two separate cases stays the same.

5.1 Dedicated Resources

To characterize a dedicated resource, the processor vector simply has a number corresponding to the amount of that resource available on the processor. The task *requirements* vectors are similar. These contain a number corresponding to the amount of the resource in use. One of the requirements vectors corresponds to the case in which the resource is not available. In this case, the component of the vector corresponding to the resource indicates that none is in use. In addition, the other components will indicate little or no use of any resources, indicating that the task is waiting for the resource. The **task quality** will be 0, reflecting that the task is not accomplishing anything. This indicates that this is a poor choice of location for the task. However, in some situations it may be desirable to place the task on a processor even though it won't be able to run. For example, if some resource is located at only one processor, the tasks needing that resource could be placed on that processor to wait for the resource to become available.

An example of a dedicated resource would be memory (assuming a single level memory system, systems using paging and caching are discussed in the next section.) The component of the vector $capacity(P)[memory]$ would indicate the amount of memory available. A task T requiring a fixed amount m of memory would have one vector $R_0 \in requirements(T)$ with $R_0[memory]=0$. The rest of the components and the task quality of R_0 would also be 0. All of the other vectors $R_i \in requirements(T), i \neq 0$ would have $R_i[memory]=m$. The meaning of the invariant given in equation 4-1 is clear; the tasks on a processor can not use more memory than the processor has.

In some cases, the task may be able to run without a dedicated resource, perhaps by using an alternative resource. In this case, instead of having a single requirements vector for the case in which the dedicated resource is not available, the set of requirements vectors is partitioned into two subsets. One subset of the potential vectors handles the case where the resource is not available. The vectors in this subset reflect the running characteristics of the task without the resource, and show that the task is not making use of the resource. The second subset states that the resource is in use, and each vector corresponds to a potential operating condition while using the resource. This is the general case of dedicated resources.

5.2 Shared Resources

Shared resources are those which can theoretically be divided up infinitely. A prime example is a CPU, which can be time-shared between any number of tasks. Another task can always be added, but the tasks already on the processor will suffer. Note that this is in theory; in practice there is normally an upper limit on the amount of sharing that can be done. This can be handled by having an extra component in the vector which keeps track of the maximum amount of sharing which can be done, in the manner of a dedicated resource.

The interesting part of a characterizing a shared resource is representing the sharing. For the capacity vector of the processor, each shared resource is represented by a value in the vector corresponding to the amount of that resource which the processor is capable of providing. Task vectors contain a corresponding value denoting the amount of the resource that that task is using. This seems no different from the method for handling a dedicated resource. The difference shows up when a particular resource becomes a *bottleneck*. In this case, the resource is fully utilized. In the dedicated resource case, this meant that no other task could make use of the resource. The value for a dedicated resource in the requirements vectors for a task had only two possible values, corresponding to the task either having or not having use of the resource.

With a shared resource, addition of another task results in some or all of the tasks receiving a smaller share of the resource. How this division of the resource is done depends on the scheduler in the operating system. With a fair scheduler, the loss of resources will be spread across all of the tasks. More complex schedulers may divide the resource up differently. However, in any case, the task requirements vectors for a task will have many possible values for the resource, depending on what share of the resource the task is given. If the capacity of the processor for a shared resource is not as great as the amount of that resource the tasks on the processor would like to use, then the current requirements vector for some of the tasks will reflect a lower than desired use of the resource, so as to satisfy equation 4-1. This will lower the quality of the tasks, and thus the quality of the system.

Using the example of a CPU, the capacity vector of a processor contains a component with the number of instructions per second that the processor is capable of. The requirements vectors for each task will have a corresponding component giving the number of instructions per second used by the task. This number will vary between 0 and some maximum, depending on which resource vector is the *current requirements vector.* The current requirements vectors for all of the tasks are chosen so as to satisfy equation 4-1. As a result, the number of instructions per second used by all of the task on a processor is less than or equal to the number of instructions per second that the processor is capable of.

A more complex example, involving both shared and dedicated resources, is a multilevel memory system (caching, or paged main memory). Overall, the total amount of virtual memory on the processor is a dedicated resource (although enough is normally available that this will not be a major factor in load balancing.) But main memory (or a cache) is shared. When all main memory is in use, some of the information in main memory is paged out to a disk, and other information is brought in. This allows processing to continue (although at a slower rate.) This sort of a memory can be represented using a separate component of the resource vector for each type of memory. Tasks attempt to run entirely in main memory, but when this is not possible, a vector is chosen which represents the *average* amount of main memory in use by the task, as well as the total virtual memory in use. As the average amount of main memory in use by a task goes down, so would the task's use of other resources. This would cause the task quality to go down.

In terms of the model, the difference between shared and dedicated resources is that whereas a dedicated resource partitions the requirements vectors into two subsets, a shared resource partitions the requirements vectors into a possibly infinite number of subsets. In effect, then, dedicated resources can be thought of as a subset of shared resources. However, I see reasons to actually think of the two as different.

1. Load balancing decisions for dedicated resources involve qualities with only two possible values corresponding to the availability or lack of the resource. In many cases, this is a simple run/no run decision.

2. Use of a dedicated resource does not change the current requirements vector of other tasks. This makes predicting the results of a placement decision much easier.

A load balancer will have to ask different questions for the two types of resources. With a dedicated resource, the question will normally be "Which tasks should be run?" With a shared resource, the question is closer to "In this configuration, are there any tasks which are not running well enough?" The first is a much simpler question to answer, perhaps by prioritizing the tasks. The second problem involves many trade-offs. Except in real-time control systems, it is rare to have an absolute cutoff for adequate performance.

5.3 Representing Communications

The cost of communicating over a particular channel is easily represented by either a shared or dedicated resource (depending on what sort of a protocol is being used.) However, deciding which communications channel a task is going to use is a different matter. In fact, two tasks which communicate will not need to use such a channel at all if they are on the same processor. How can this be represented?

One way to do this is to have a component in the resource vector which contains a different value for each link which may be used to communicate between the two tasks (including one for the case when they are on the same processor.) This component is used only for purposes of determining which link is used for communication between the tasks, and does not correspond to a processing resource. The **profile** function will know that for this component, resource vectors must be chosen such that the value of this resource in the current requirements vectors for the two tasks are compatible.

This component partitions the set of requirements vectors for the two tasks into a different subset for each possible communication link between the tasks. The vectors in each subset will reflect the processing resources used when that link is in use for the intertask communication. For example, if one of the processing resources was a communication link between processors *a* and *b*, then the resource vectors for the two tasks would only use this resource (the communication link) if one task was on *a* and the other was on *b*. The "extra" component would make sure that the current requirements vector for each of the tasks would show use of the communications link if and only if one task was on *a* and the other was on *b*.

In this case, the **status** function is no longer valid, since the requirements vector currently in use may be dependent on the location of a task which is not at the current processor. This appears to make load balancing more formidable, although it is difficult to actually prove that the problem is more difficult.

Chapter Six

Evaluation of Load Balancing Algorithms

The previous chapters have specified what load balancing is, what information is used in load balancing, the goal of load balancing, and how to characterize a particular system. Before introducing an algorithm for load balancing, I would like to discuss methods of judging load balancing algorithms. One method is to compare the quality of the configurations produced by an algorithm with the quality of an *optimal* configuration. This optimal assignment is the one which actually has the highest configuration quality. Finding the optimal configuration may be computationally infeasible, however (see section 6.1.) As a result, there should be other means by which to measure load balancing algorithms.

One criterion is the *efficiency* of the load balancing algorithm. A load balancing algorithm will take processing resources away from other tasks. A trade-off must be made between the gains of load balancing, and the cost involved in load balancing. An algorithm which determines the optimal configuration is not helpful if the time required to run the algorithm is longer than the time available to complete the tasks which are waiting to be assigned.

Another criterion based on preventing the load balancing algorithm from overusing system resources is *stability*. If task requirements and processor capacities do not change, and new tasks are not introduced, then multiple runs of a *stable* load balancing algorithm (in particular, the **redistribution** function) will eventually stop making changes in the configuration. A load balancing algorithm which does not meet this criterion will constantly be moving tasks. The load imposed by moving tasks will be detrimental to the system. As such, a load balancing algorithm should be able to reach some point at which it is satisfied, and seeks no further improvements.

This is not enough, however, as no load balancing at all (assigning tasks to processors on some random basis) will satisfy the two previous criteria. Testing if the algorithm *improves* the configuration quality will ensure that a load balancing algorithm gives some gains, but is a less demanding criterion than comparing with an optimal configuration. This, like *stability*, is most pertinent to **redistribution** algorithms.

This is not intended to be a complete list. It is a sampling of considerations which are important in the design of a load balancing algorithm. The following section justifies the use of criteria other than optimality in judging load balancing algorithms.

6.1 Computational Difficulty of Load Balancing.

Finding an optimal solution to load balancing (an algorithm which maximizes the configuration quality) can be a computationally expensive problem. A particular load balancing problem is given by specifying a *system* (definition 4-5), and *task quality* and *configuration quality* functions. An optimal solution for a particular load balancing problem is a load balancing function (definition 3-2) which gives the configuration with the highest configuration quality for any particular set of tasks to be run. For some load balancing problems, finding an optimal solution can be shown to be an NP-hard problem. This section describes such a problem, and shows that it is NP-hard.

I will actually be working with a problem which I call *minimum acceptable configuration.* This problem is to determine if given a particular minimum configuration quality M and a set of tasks to be run, there is a configuration with $\mathit{configuration\,quality} \geq M$. This is no harder than the optimal solution problem, as if we can find an optimal solution to a given load balancing problem, we can easily check to see if the optimal configuration quality is greater than the minimum M. Showing that the minimum acceptable configuration problem for a particular load balancing problem is NP-hard will thus show that optimal load balancing for that problem is also NP-hard.

In addition, I restrict minimum acceptable configuration to finding an *initial distribution*. This is a special case of the general load balancing problem (definition 3-2), and thus showing that initial distribution is NP-hard for a particular problem shows that general load balancing for that problem is also NP-hard.

The minimum acceptable configuration problem, then, is as follows:

> **Definition 6-1:** For a particular load balancing problem (i.e. a system plus task and configuration quality functions), the **minimum acceptable configuration** problem asks:
>
> > Given a set of processors $\mathcal{P}$, a set of tasks S, and a *minimum acceptable configuration quality* M, does there exist a configuration $C = \mathit{initial\,distribution}(S, \mathcal{P})$ such that
> >
> > 1. $\forall\, T \in S, \exists\, P \in \mathcal{P}$ such that $(T, P) \in C$
> >
> > 2. $\mathit{configuration\,quality}(\mathit{profile}(C)) \geq M$?

Note that this problem requires that all of the tasks be run.

There are scheduling problems which are similar to load balancing which have been shown to be NP-complete. These use a single value, the time required to complete a task, as the sole information needed to describe a task in order to make scheduling decisions. This is comparable to load balancing using a single processing resource. The following class of load balancing problems is similar to these scheduling problems.

Definition 6-2: The **single resource** class of load balancing problems is defined as follows:

Let R be a single-component rational-valued vector.
Note: Since there is only a single processing resource in this system, I will actually use resource vectors as a value instead of a one-component vector. This is done for ease of notation, and is no different from using a single component resource vector.

Let P be a fixed set of processors.

Let T be a fixed, infinite set of tasks, where each task $T \in \mathrm{T}$ has an associated weight $w(T) \in \mathbb{Q}, 0 < w(T) \leq 1$.
This weight is used to rank the tasks in terms of the amount of resources required. This would normally be incorporated into the *requirements* of the task, but is separated to simplify the NP-hardness proof.

$\forall P \in \mathrm{P}$, let $capacity(P) = 1$.
Note that all processors are identical.

$\forall T \in \mathrm{T}$, let $requirements(T) = \{x \mid x \in \mathbb{Q}, 0 \leq x \leq 1\}$.
The *current requirements* of a task can be anything from 0 (using none of the resource) to 1 (using all of the resource available on a single processor.)

$task\ quality(T, R) = w(T) \cdot R.$

$$configuration\ quality(\{(T, R)\}) = \min_{\{(T,R)\}} (task\ quality(T, R)).$$

The configuration quality is the quality of the lowest quality task in a given configuration.

$profile(C) = \{(T, R)\}$ such that for each $P \in C$,

1. $\sum_{(T, R) \in profile(C) \mid (T, P) \in C} R = capacity(P)$
2. If $(T_i, P), (T_j, P) \in C$, and $(T_i, R_i), (T_j, R_j) \in profile(C)$, then $task\ quality(T_i, R_i) = task\ quality(T_j, R_j)$

The first item is equation 4-1, and the second states that the current requirements vector for all of the tasks on a given processor are chosen so that the task qualities of the tasks on that processor will be the same.

We can now state the theorem which is the focus of this section.

Theorem 6-3: The *minimum acceptable configuration* problem for a *single resource* load balancing problem is NP-hard.

Proof: Reduction from the *multiprocessor scheduling* problem given on page 65 of [Garey 79]. The scheduling problem as defined by Garey and Johnson is as follows (paraphrased and some notation changed for compatibility):

Theorem 6-4:

Let $\mathcal{T}$ be a finite set of tasks,
$l(T) \in \mathbb{Z}^+$ the "length" of $T \in \mathcal{T}$,
$m \in \mathbb{Z}^+$ a number of processors,
$D \in \mathbb{Z}^+$ a deadline.

The **multiprocessor scheduling problem:**

Does there exist a partition $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \ldots \cup \mathcal{T}_m$ of $\mathcal{T}$ into m disjoint sets such that

$$\max_{\mathcal{T}_i \subseteq \mathcal{T}} \Big(\sum_{T \in \mathcal{T}_i} l(T) \Big) \leq D \quad ?$$

is NP-Complete.

Proof: See [Garey 79].

This corresponds closely to the problem of theorem 6-3. We have a set of tasks, a set of processors, a cutoff (in this case, a deadline, not a minimum quality), and a weighting function for tasks.

The multiprocessor scheduling problem states that if the total length of all the tasks assigned to any processor is too great, they will not be completed by the deadline. This is analogous to saying that if the tasks on a processor desire too much of a resource, the *task quality* of these tasks will be unacceptably low.

Given this, the maximum used in theorem 6-4 corresponds to *configuration quality*. The multiprocessor scheduling problem asks if there exists an assignment such that all of the tasks will complete by a certain deadline. The minimum acceptable configuration problem asks if there is an assignment such that the configuration quality is greater than the minimum. If we take the configuration quality to be the minimum of the task qualities, then the two are analogous.

In order to show the reduction, it must be shown that a solution for the problem of theorem 6-3 can be used to solve the problem of theorem 6-4, with any conversions necessary being performed in polynomial time. The close correspondence between the problems simplifies this. The following lemma shows that a solution for the minimum acceptable quality problem for single resource load balancing can be used to solve the multiprocessor scheduling problem.

First, we must convert an instance of the multiprocessor scheduling problem to an instance of the minimum acceptable configuration load balancing problem. This can be done using the following function f:

> Given an instance x of the multiprocessor scheduling problem (a finite set $\mathcal{T}$ of tasks, a number of processors m, a deadline D, and a length function $l(T)$), define an instance $f(x)$ of the minimum acceptable configuration problem for single resource load balancing as follows:
>
> $S=\{T \in \mathrm{T} \mid$ each $t \in \mathcal{T}$ has a corresponding T such that $w(T)=\dfrac{1}{l(T)}\}$
>
> $\mathcal{P}=\{P_1 \ldots P_m\}$
>
> $M=\dfrac{1}{D}$
>
> Note that the set S must be of the same size as the set $\mathcal{T}$.
>
> **Lemma 6-5:** Instance x of the multiprocessor scheduling problem is satisfiable if and only if instance $f(x)$ of the minimum acceptable quality problem for single resource load balancing is satisfiable.

Proof: Assume we can solve the minimum acceptable configuration problem for single resource load balancing. Then we know that the following is true if and only if an instance satisfies the minimum acceptable configuration problem:

$$\mathit{configuration\ quality}(\mathit{profile}(C)) \geq M$$

$$\min_{(T,R)\in \mathit{profile}(C)} (\mathit{task\ quality}(T,R)) \geq M$$

Using the definition of profile given in definition 6-2, we know that the task quality on any given processor P is constant, and is given by

$$\forall (T,R)\in \mathit{profile}(C), (T,P)\in C,$$
$$\mathit{task\ quality}(T,R) = w(T)\cdot R = \frac{R}{l(T)}$$

We also know from the profile function that

$$\sum_{(T,R),\, T\, on\, P} R = \mathit{capacity}(P)$$

Combining these (and noting that $\mathit{capacity}(P)=1$), we get

$$\frac{l(T)}{R}\sum_{(T,R),\, T\, on\, P} R = \frac{l(T)}{R}$$

Since the task quality, and thus its inverse, is constant on any given processor,

$$\sum_{(T,R),\, T\, on\, P} \frac{l(T)}{R}R = \frac{l(T)}{R}$$

$$\sum_{(T,R),\, T\, on\, P} l(T) = \frac{l(T)}{R} = \frac{1}{\mathit{task\ quality}(T,R)}$$

This can be carried back into the configuration quality to give

$$\min_{P\in \mathrm{P}} \left(\frac{1}{\sum_{(T,R),\, T\, on\, P} l(T)} \right) \geq M$$

(iff the problem is satisfied.)

Since $D\in \mathbb{Z}^{+}$, $D=\frac{1}{M}\geq 1$, so the previous line can be transformed to

$$\max_{P\in \mathrm{P}} \left(\sum_{(T,R),\, T\, on\, P} l(T) \right) \leq D$$

(iff the problem is satisfied.)

Since the partitions of $\mathcal{T}$ in the multiprocessor scheduling problem correspond to the assignment of tasks to processors above, this last statement answers the multiprocessor scheduling question. ▮

All that remains is to show that the conversion function f can be performed in polynomial time. The construction of the set of processors $\mathcal{P}$ and the computation of the minimum acceptable configuration M are trivial. The only difficult problem is the proper choice of tasks in S. For each task $t \in \mathcal{T}$, the set of tasks to be scheduled, a corresponding task $T \in \mathrm{T}$ must be chosen such that its weighting function $w(T) = 1/l(t)$. However, since the tasks requirements sets are all identical, the only distinguishing characteristic is the weighting function. This allows us to construct the set S by constructing a weighting function for each task. This is trivial, and thus the construction of S can be done in polynomial time.

This shows the polynomial time reduction of the multiprocessor scheduling problem to the minimum acceptable configuration problem for single resource load balancing. ▮

This proves that finding an initial distribution which meets some minimum configuration quality for a single resource load balancing problem is NP-hard, and thus the problem of optimal load balancing for the problem is NP-hard.

The class of load balancing problems defined above is actually quite simple, with all processor capacities identical and a single component in the resource vector. It can be easily reduced to load balancing problems specified in other ways by a simple restriction. I believe that many of the load balancing problems encountered in actual systems can be shown to be NP-hard in this manner. This justifies the search for non-optimal, but efficient, algorithms for load balancing.

Chapter Seven

Heuristics for Load Balancing

The emphasis of this thesis is not on developing optimal or efficient algorithms for load balancing. It instead concentrates on working out a better way to describe the problem, and showing that this description is useful in relation to real-world systems. There has been a good deal of work done in the area of load balancing algorithms (see section 1.4.) Some of this work could be extended to use this model for load without great difficulty, using the technique discussed in the next section for reducing resource vector load to a single value. Any serious discussion of load balancing algorithms for this model should go beyond that, however. Such a discussion would be beyond the scope of this thesis. As such, the following simple algorithm is presented with little discussion as to optimality of decisions or comparison with other algorithms.

7.1 Placement Algorithm

The system implemented uses a simple **task placement** algorithm. The idea behind the algorithm is that a single processing resource is chosen as the *critical* resource, and this resource is used to find the best location for the task. This would seem to have all of the problems of using a single value for load. However, this algorithm differs from algorithms based on a single measure of load in that the decision of what to use as a load balancing criterion is based on the current system configuration. For example, a database manager will be heavily dependent on access to disk, and thus will be placed on a processor with plenty of unused I/O capacity. However, if I/O is not heavily utilized in the system, some other resource may be used as the criterion for load balancing

Actually, this single criterion is chosen from among shared resources. Dedicated resources are checked first, and if placing the task at a given processor would violate the invariant of equation 4-1, it is removed from the set of possible locations for the task. Once this is done, the critical resource is chosen.

The manner used to select the critical resource is to choose one of the task's resource vectors as a **baseline** vector. This vector should be one which gives a measure of the relative use of various resources under a wide variety of configurations. For example, a mail system would primarily make use of communications channels. Although some CPU and other resources would be needed, these would be small in relation to the communication cost. If the communication channel was heavily used by other tasks, the amount used by the mail server would probably be small, but the amount of CPU use would go down as well. The communication used by the mail server would still be relatively large compared to the use of other resources.

There are a number of possible ways to choose this baseline vector. One possibility would be to use the vector which is most often chosen as the current requirements vector. Another would be to decide on some standard configuration, and use the current requirements vector in that standard configuration as the baseline vector. This latter method is the one used in the system described in chapter 9. The choice of the baseline vector is dependent on the system, and should reflect the relative use of resources in a variety of configurations.

Once the baseline vector is chosen, the resource in this vector with the highest value could be used as the critical resource. However, this strategy does not take into account the status of the rest of the system. The method used is to compare the task's baseline vector with a vector which is built from information about the current system state. This is done by finding a component-by-component average of the current requirements vectors of all the tasks in the system. This average gives a means for determining which resources are heavily used in the system. In more formal terms:

Definition 7-1: Average(P,C) is a resource vector defined as follows:

Let $\mathfrak{R} = \{R\}$ such that $(R,T) \in profile(C)$ and $(T,P) \in C$.

$$\forall \text{ resources } i,\ average(P,C)[i] = \frac{\sum_{\mathfrak{R}} R[i]}{|\mathfrak{R}|}$$

The baseline vector of the task to be placed is compared with this average vector, and the component which is the highest in relation to the average is used as the critical resource. If the system is short of some shared resource, each task will get less of the resource, and the average will be lower. This will cause the difference to be greater, increasing the chance that the resource will be chosen as the critical resource. For example, if the processors in the system do not have enough CPU cycles/second to keep up with the demand, the use of the CPU by each task will drop. If the task to be placed is a mail server, it would normally be considered communication intensive. However, comparing the low values for CPU use in the average vector with the baseline vector of the mail server may show that the mail server uses a relatively high amount of CPU. The placement will then be done based on the availability of CPU cycles, as this is of greater effect on the configuration quality than the relatively lightly used communications channels. The algorithm for choosing the critical resource is shown in figure 7-1.

Once the critical resource is chosen, the algorithm looks for the processor with the highest **availability** of that resource. If all of the processors have 0 availability of the resource, the processor whose tasks have the highest **average** for the resource is chosen. This assumes that the tasks with lower averages are doing poorly, and could ill afford to give up more of the critical resource. The algorithm for selecting a processor is shown in figure 7-2.

Critical$(R, \mathcal{P}, C)$ is a function which takes a resource vector R, a set of processors $\mathcal{P}$, and a configuration C, and returns the critical resource c determined as follows:

Let A be the resource vector such that:

$$\forall \text{ resources } i,\ A[i] = \frac{\sum_{P \in \mathcal{P}} average(P, C)[i]}{|\mathcal{P}|}$$

Return the shared resource c such that

$$\forall \text{ shared resources } i,\ \frac{R[c]}{A[c]} \geq \frac{R[i]}{A[i]}$$

Figure 7-1: Choosing the Critical Resource.

Placement(T, C) takes a configuration C and a task $T \in C$, and returns a processor P determined as follows:

```
Let $\mathcal{P}$ be the set of processors in $C$,
    $R \in requirements(T)$ be the **baseline** vector chosen for $T$.

For each dedicated resource $d$,
    For each processor $P \in \mathcal{P}$
      if $availability(P, C)[d] \leq R[d]$ then $\mathcal{P} = \mathcal{P} - \{P\}$

$c = critical(R, \mathcal{P}, C)$

Let $P \in \mathcal{P}$ be the processor with the highest $availability(P, C)[c]$.
    If $capacity(P)[c] > 0$, return $P$.
      else return $P \in \mathcal{P}$ with the highest $average(P, C)[c]$.
```

Figure 7-2: Placement algorithm for Load Balancing.

Chapter Eight

Distributed Control

One of the problems in this load balancing implementation is communicating load balancing decisions to different processors. The load balancing system monitors tasks and processors in order to determine requirements and capacities. Since the monitoring must be done at each processor, the load balancer is already somewhat distributed. This leads to questions about the load balancing algorithm, should it be distributed, or should a central load balancer send new tasks to the proper locations? This chapter begins with a description of some of the requirements of the system resulting from the environment described in chapter 2. The remainder will discuss possible solutions. The one used in the implementation is then presented in detail.

8.1 Requirements

The primary goal of the Highly Available Systems group is just what the name implies: providing a reliable system. As a result, any load balancing algorithm must be able to handle failures. This immediately gives us one requirement: duplication of information. Necessary information must not be confined to a single site, where it could be lost in case of a failure. It is easy to see that in order to handle n failures, the information must be available at at least $n+1$ sites.

This also leads to a requirement that load balancing be distributed; if the load balancer were at a single site, failure of that site would cause loss of load balancing. This can be handled in many ways: a voting system; leader election in case of failure (or a predefined succession list); or distributing the algorithm such that the loss of a particular node (and its load balancer) will not affect the rest of the system.

8.2 Possible Protocols

One method of providing a reliable load balancing system would be to make all decisions at a central site. This site would be chosen using a leader election protocol. The load balancing algorithm would not have to worry about concurrency in its own operation. The only remaining problems would be collecting the needed data and communicating the results, so as to actually act on the decisions. This would be basically a master-slave arrangement, with the master sending commands to all of the remaining nodes.

Certain nodes would have to be chosen as potential masters, so as to have the information available to take over in case of a failure of the current master. In effect, this means full replication of code and data at all of the chosen sites; in order to avoid having an arbitrary limit on the number of allowed failures, all information must be present at all sites. This would be easy to change, however, should someone wish to set such a limit and gain the corresponding savings in replicating information.

The other possibility would be to develop a distributed load balancing algorithm. This would have certain advantages. It may be possible to make decisions locally, for example, a processor could determine that it is a good place to start a new task without looking at other processors. This would lower communication costs. However, this appeared to be a difficult approach, and is not pursued in this thesis.

8.3 Delta-common storage

The protocol used, suggested by Flaviu Cristian [Cristian 85a]realizes some of the advantages of a totally distributed protocol, while being simple and robust. This is done using *atomic broadcast* (described in section 2.1.1.)

The idea behind δ-common storage is that all of the processors *in effect* share common memory, which happens to take up to δ time units to update. In practice, all changes to the δ-common storage are made by broadcasting the update using an *atomic broadcast.* The atomic broadcast guarantees that a message will arrive at all sites within δ time units, or will not arrive at any sites.. Each processor maintains a local copy of δ-common storage, which is updated only when a broadcast is received. Since the broadcast messages are received at all sites (or at none), this guarantees that all copies of the storage will be identical.

Once we have the δ-common storage, it is simple to write a fully distributed protocol for load balancing. All of the information on task and processor characteristics is kept in δ-common storage. Any changes to this (for example, a processor failing or a task changing its characteristics) are made using an atomic broadcast. In addition, any request to place a task, or otherwise redistribute the load in the system, is made using an atomic broadcast. When a request to place a task or redistribute the load arrives, each site runs an identical load balancing algorithm. This algorithm uses only information contained in the δ-common storage. Since all of the copies of δ-common storage are identical, the algorithms will all give identical results. The load balancer at each processor only acts on results which involve starting or stopping a task *at that processor.* Since all of the decisions are identical, there is no need to communicate the results of the algorithm.

The decisions made are the same as running the same algorithm in a master-slave arrangement, except that no communication of results is necessary, and all leader-election problems are avoided. The disadvantage is in duplicating processing, but the placement algorithm that I am using is efficient enough that this is not a problem.

Chapter Nine

System Design and Implementation

The model for load and definitions of load balancing given in chapters 4 and 3 are useful for describing load balancing, but difficult to apply directly to an on-line load balancing system. For example, the model describes tasks in terms *requirements*, a set of resource vectors which potentially characterize the task. This could be a very large set. Given a different resource vector for each configuration, if we have p processors and t tasks, there are n^t resource vectors for each task. Storing such a set on the computer may be infeasible. In addition, it may be impossible to actually determine in advance all of the possible vectors which could characterize a task. As a result, it is necessary to *approximate* this set, providing the vectors necessary for the load balancing algorithm. For example, the load balancing algorithm of chapter 7 needs only the current requirements vector, and a baseline vector.

Determining this approximation is one of the more difficult parts of designing a load balancing system. The critical factors in the load must be determined, and their interactions studied. For example, using all of the main memory of a computer will result in paging, which will slow down each task and cause each to demand less of a percentage of the CPU. Actual or simulated use of the system should be studied in order to determine what conditions actually result in a fast or slow response time.

This chapter is devoted to a description of the design of the load balancer which I developed for the system described in chapter 2. I will try to avoid implementation details and instead give an overview of the reasons behind design decisions.

9.1 Monitoring

One of the first decisions was that the **requirements** vectors should be determined dynamically. This meant that I would have to monitor the tasks to determine the use of each resource. Another possibility would be to ask the programmer to describe a task, or otherwise statically determine the characteristics. I have already given some reasons why I believe this is not a good method. One of the most important was the desire to have an **automatic** system which did not require user intervention. I could also have chosen to have some initial *set-up* time during which the characteristics of all of the tasks would be determined. This would neglect the possibility that tasks change with time, and could impose difficulties on adding new and different tasks.

Continuously monitoring the system imposes certain constraints and allows me to take some liberties. Non-optimal placement decisions are less critical, as mistakes will show up in the monitoring and redistributions made. As long as the algorithm results in significant improvements in the configuration quality, a good (although not necessarily optimal) configuration will eventually be reached. This allows for a simpler and more efficient load balancing algorithm. The disadvantages are from the extra cost imposed by monitoring the system, and the difficulty of monitoring some resources. Intertask communication, for example, is difficult and expensive to determine by monitoring the system. Tracking each message and communicating the results could significantly increase communication costs.

9.1.1 Choice of Processing Resources to Consider

One of the first steps in the design was to choose which processing resources to consider. In chapter 5 I mentioned a number of possibilities: CPU, memory, communications, etc. Which of these are important in this environment? In order to make this decision, I looked at three factors for each resource:

1. Ease of monitoring: It would not make sense to make load balancing decisions on information which was not available.

2. Likelihood of becoming a bottleneck: There is no need to use a processing resource in determining load if it is so plentiful that it will never be critical.

3. Ease of computation: Some resources may have complex interactions which make them difficult or expensive to use in figuring load.

I chose to look at CPU use, memory requirements, and total I/O. CPU use is easy to obtain. The operating system maintains values on both the total utilization of the CPU and the total amount of CPU time used by each task. These can be used to get values for the amount of CPU time used per second. Memory use is slightly more difficult; the operating system maintains values for the paging rate and the amount of memory actually used by each task (as an average per timeslice.) Calculating from these gives a reasonable set of values to use to determine how heavily utilized the memory is.

Total I/O is also easy to obtain. The operating system maintains values for total number of **Start I/O** instructions (which call operating system primitives to perform the actual I/O.) I would have liked to break I/O down into separate categories for each processor-processor path, and each path to disk. Use of the disks can proceed in parallel, and encouraging this parallelism is a goal of the load balancer. However, although the information necessary to make these decisions is available from the operating system, it would be computationally expensive to obtain without incorporating the monitoring directly into the operating system, which was beyond the scope of my project.

For similar reasons I was not able to take into account interprocess communications. Using the actual expense of communicating on each particular processor-processor path would not have been necessary. The major cost of interprocess communication appeared to be in the drivers on either end, due to high-speed communication channels. Therefore the communication cost was not as much a factor of which pair of processors were communicating as a factor of whether there was any interprocess communication required. This would have made for a simple decision process; there would be two task vectors corresponding to whether communicating processes were on the same or different processors. I would have liked to incorporate this feature into the system, however, I felt the expense and difficulty of tracing message traffic would have been unreasonable.

9.1.2 Load averaging methods

The next step was to determine how to find the *current requirements* and *baseline* vectors for a task. Determining the current requirements vector is an easy task for the monitoring system, as the information available characterizes the task in the current configuration. The baseline vector is more difficult. To determine this we must be able to *predict* how the task will run under a different configuration. This is done by comparing current requirements vector with the current vectors of other tasks, and the availability vector of the processor. The exact manner in which this is done is described in section 9.1.3.2.

Determining these vectors is primarily a problem of studying the operating system and performing experiments. It is difficult to give general methods for doing this. I will instead describe the methods used in my implementation for VM.

9.1.3 Data Gathering Implementation

Once the choice as to important processing resources had been made, it was necessary to actually determine how to gather and store this information. The VM operating system maintains information about processor use in control blocks which may be accessed by privileged users. I periodically gather and process this information. The actual details of how the raw data is turned into useful information follows.

9.1.3.1 Processor Data

I deviated from this model in my handling of information about processors. The *capacity(P)* vector is supposed to give the total amount of each resource on the processor. I instead used data on the available capacity (or rather, the capacity already in use.) Using the model directly, this information would be obtained from looking at the current requirements vectors of the tasks. However, the information as to the current load on the processor was easier to obtain by monitoring directly. In effect, what I have is an *availability* vector. This does not require many changes in the algorithm of chapter 7. In this system, it was computationally easier to use availability.

The VM operating system keeps most basic information in the Prefix Storage Area. It is out of this area that I gather information as to the machine state. I am really only interested in measures of CPU utilization, memory use, and I/O use; but these values are not readily available. The information available amounted to total time spent in each of a variety of wait states (see table 9-1). Periodically obtaining these values (and noting the time passed between measurements) we can note the ratio of time spent in each state as follows: as follows:

$$\mathit{ratio\ of\ time\ spent\ in\ wait} = \frac{\mathit{New\ wait} - \mathit{Old\ wait}}{\mathit{Current\ time} - \mathit{Previous\ time}} \qquad (9-1)$$

Supervisor state CPU time (Operating system CPU use) can be obtained by subtracting the sum of the resulting ratios from one.

This gives a good measure of what the current system bottlenecks are. If a significant portion of the time is spent in page wait, the memory is probably

Name	*Description*
Idlewait	Total system idle wait time.
Pagewait	Total system page wait time.
IOntwait	Total system I/O wait time.
Probtime	Total system problem state time (User CPU use.)

Table 9-1: Data areas containing information on CPU use [IBM 82].

overtaxed. But I am after the *total* utilization. Knowing the bottleneck on a particular processor does not help determine if it is a better choice for a task than another processor with a similar bottleneck.

In order to find the total utilization, I look at *queue length*; the number of tasks actually waiting for some type of processing resource. By multiplying this value by the ratio of the wait time for a resource to the total time passed (equation 9-1), I obtain a good value for the desired amount of each resource. This is actually a deviation from the model, for this does not correspond to the *availability* vector. However, figuring the resources in this way includes information about how much of the resource is desired with a fully utilized shared resource into the availability vector. This necessitates only a few changes in the algorithm, and results in a computational saving in this system. The value thus obtained is then averaged in with old data, so as not to overreact to temporary changes in system use. This is a choice made due to the degree of coupling of the system, and the expense of moving tasks.

The actual implementation also has a number of constants which are used to weight this information. All of the raw values used to compute the vectors are multiplied by their corresponding constant before being used. This is needed in order to have the vectors meet certain constraints. For example, a vector for a processor should show that the demand for a particular resource has doubled is the need for that resource by the tasks running on the processor doubles. Task vectors satisfy a different constraint. They are supposed to reflect a baseline vector, which reflects the resource requirements of the task running in some hypothetical standard configuration. A change in the makeup of the task should be reflected in this vector, but a change in the way it runs due to a change in the demand on the processor should not. The proper values for these constants were determined by experimentation. Tests were run with the system running under a variety of configurations, and the constants were modified until the desired results were achieved. Some samples of the actual vectors of systems running under different configurations are shown in appendix A. This experimentation turned up certain other situations which the monitoring must correct for. For example, each of the processors runs a background task that causes the system to spend all idle time in I/O wait, thus giving the impression that an idle processor is overloaded with I/O.

9.1.3.2 Task Data

Obtaining the task data was similar. Each task (corresponding to a *virtual machine*) has a corresponding VMBLOK. Each VMBLOK contains considerable information about that task (see table 9-2.) The CPU values are added and divided by the time between monitoring updates to obtain a ratio of CPU use. The same is done with the SIO count to obtain I/O rate. The Drum, disk, and core memory values are added to obtain the total amount of memory allocated to the task. The working set size is taken directly as a measure of how much actual primary memory is needed.

This gives a value corresponding to the tasks *current requirements vector*, but the

Name	*Description*
VMVtime	Virtual Problem-state CPU time used.
VMTTime	Virtual Supervisor-state CPU time used.
VMIOcnt	Virtual SIO count for non-spooled I/O. (I chose to ignore spooled I/O, as the tasks of interest in this system used little of this type of I/O.)
VMPDrum	Count of user pages on drum.
VMPDisk	Count of user pages on disk.
VMPages	Number of currently resident real pages.
VMWSProj	Projected working set size. (Number of pages needed to run.)

Table 9-2: Data areas containing information on tasks [IBM 82].

algorithm also needs the **baseline** vector. To do this, I multiply the CPU and I/O values by their corresponding values for the processor. This (after adjusting by some constant factors as described in the previous section) gives a relatively stable value, regardless of system load.

Each of these values are averaged in with old data, so as not to overreact to temporary anomalies in the running characteristics of the task. I chose to average this so as to accomplish a 90% replacement of data every hour; this value should be chosen for each system based on the expense of moving tasks.

9.1.4 Communicating Information

I have already discussed my desire to fully replicate information on the status of the system. The primary drawback to this is communication cost; each update to the information must be broadcast throughout the system. Therefore I keep the monitoring results in separate local storage until *significant* changes occur. After each update this local information is compared with the (local) copy of δ-common storage. If there is a large change in the status of the processor or one of the tasks then the new vector is used to update δ-common storage. This is done by broadcasting a message stating that the vector has changed, and giving the new values.

The load balancing algorithm uses only the information in δ-common storage. As a result these decisions may not be made on the most current information available. All decisions will be consistent, however, and can be based on arbitrarily current information (within the limit imposed by δ) at the expense of increased communication costs.

9.2 Design of the Load Balancer

The monitoring subsystem puts the resource vectors into δ-common storage. The load balancing subsystem is started on each receipt of an update to this storage (by the task which receives the broadcast update.) It then goes and checks to see if the new information is significant enough to demand action. Since all processors have a load balancer executing the same algorithm on the same data, the same decision will be reached at each. The action thus decided upon is carried out, with each of the load balancers performing the part of the action relevant to its processor. A more concrete version of this is shown in figure 9-1.

Since new_task requests often come in frequent batches (for example, when a

The *load balancing program* is a continuous loop which waits for a message, and then acts on it.

Note: **Placement** is the placement algorithm given in figure 7-2.

Storage is the local copy of δ-common storage.

Local_storage is the copy of storage maintained by the monitor containing current information about the local processor and tasks. This is used only to broadcast information when a new load balancer is started.

```
Broadcast a configuration_request message.
Wait for 2*δ, to give time for all of the information to arrive.

repeat
  message := The next message received, taking update messages first
             if there is more than one in the buffer.

  if message.type = new_task then
    The message contains a request to place a new task in the system.
    processor := placement(message.task, storage)
    if processor := local_processor then
      Start the new task on this processor.
    storage := Predict the way the vectors for all of the tasks
               and processors will look after the task has started.

  elseif message.type = update_to_storage then
    The message is an update to δ-common storage
    storage[message.location] := message.new_vector

  elseif message.type = configuration_request then
    Broadcast update_to_storage messages with the vectors for
    all of the tasks and the processor in local_storage.

forever;
```

Figure 9-1: Load Balancing Program

processor fails) it is necessary to have some feel for the results of previous decisions before new ones are made. Otherwise all of the tasks will be dumped on the least loaded processor, causing it to be overloaded. Since monitoring takes time, I have chosen to attempt to ***predict*** the status of the system after each decision. This is done by performing a vector addition on the task and processor resource vectors. The function which actually relates each of the components is slightly more complex than simple addition, but only as a result of constant terms which are used to handle idiosyncrasies in the system. These must be determined for each operating system by study and experimentation.

Actually starting tasks is not a part of the load balancer. In the H.A.S. project (chapter 2) this is the responsibility of the **Auditor** subsystem. How these are done is very dependent on the system: A shared memory system may just transfer a pointer between processors; a loosely-coupled system may have to send code over the network. There has been research in this area, for more information see [Theimer 85].

Chapter Ten

Results

Although I have completed an operating prototype, it has not been integrated into a system in day to day use. As such, my results are based on experiments in a three-processor test system using "artificial" tasks. These tasks were tightly controlled. I am not sure how these would compare with the characteristics of actual tasks such as a compiler or database manager, but they do give a good feel for the quality of the load balancing decisions made.

10.1 Description of Tests

I created three sample tasks: a heavy CPU user, a memory-intensive process, and an I/O intensive process. These were each designed to use a given amount of the target resource, while using as little as possible of the other resources. A portion of each task was timed, in order to give a value for response time. Some results of these tests are given in table 10-1. Using this, I was able to obtain results for the change in response time of each type of job under various load conditions.

Once I had created the jobs and run some performance tests under a variety of conditions, I started the load balancing and monitoring system. I first tried monitoring tasks under a variety of conditions, to make sure that the monitoring system adequately reflected their **baseline** vector regardless of the general load characteristics. These results are summarized in appendix A.

After a waiting for a period of time to allow the monitoring to characterize the tasks, I began to move tasks. While otherwise leaving the system stable, I added a given

Results from monitoring each task in an unloaded system.
Note: Units given are intended for use only as relative measures.

	CPU use	*Memory used*	*Memory desired*	*SIO rate*	*Response time*
CPU task	2738	139	139	0	52
Memory task	0	1517	1827	0	48
I/O task	2	64	64	5561	36

Table 10-1: Characteristics of tasks used in testing.

task to each of the processors, noting the resulting change in response times. I also took note of the recommendations of the load balancer, in order to compare its decision with the optimal decision based on measuring response times in trials of all possible placements.

After trying this with multiple load situations, I tested the **predict** function. To do this, I introduced tasks slowly (following the load balancer's decisions), allowing time for the monitoring to catch up. I then performed the same set of introductions rapidly, testing to make sure that the resulting decisions were the same.

10.2 Evaluation of Decisions

The load balancing decision process performed well. It placed tasks away from others of the same type. This compared well with test data that showed that the response time of the system deteriorated when multiple tasks of the same type were placed on the same processor. Part of this is probably due to the simplistic nature of the tasks I was using. Appendix B contains some information on the actual test results.

Predict did not fare as well. For a few iterations, it performed successfully. The differences between the predicted and actual system status were not large enough to change the decisions. But the errors tended to build with the number of predictions made. Whether this is a problem is very much a factor of the normal influx of jobs in the system. If jobs come in infrequently, this is not too major a difficulty. If jobs come in in batches, it may be desirable to look at an alternative form of load balancing which will take all new tasks into account simultaneously.

10.3 Expense of Load Balancer

The load balancing algorithm itself is quite simple and inexpensive. Since the algorithm is run on demand, this expense will be paid back over time even if load balancing only results in small gains in the configuration quality. Monitoring is more of a problem. Since monitoring is continuous, it will result in a permanent decrease in configuration quality. This must be offset by a greater increase in quality as the result of load balancing.

Fortunately, the monitoring is not that expensive. I found that the monitoring used less than 1% of the available CPU. This was while monitoring 20 tasks at five second intervals. I would actually use a significantly longer interval in practice, but this savings would probably be negated by the larger number of jobs I would expect in a system this size. Table 10-2 contains a breakdown of the cost of the monitoring. The potential gains of this monitoring/load balancing system more than make up for the extra cost of running it.

	CPU use per check (seconds)	*Memory required (bytes)*
Per CPU monitored	0.0041	92
Per task monitored	0.0012	76

Table 10-2: Cost of running the monitor.

Chapter Eleven

Conclusion

I have presented a new model for load which separates different types of processing resources. Load is given as a **resource vector**, as opposed to a single value. This measure can be applied to both processors and tasks. Processors are characterized by a **capacity** resource vector, which describes the availability of each type of processing power on that processor. Tasks are described by a set of **requirements** vectors, where each vector in the set corresponds to the tasks usage of resources in a particular system configuration (assignment of tasks to processors.)

This load model has been applied to the problem of load balancing in distributed systems. In distributed systems where each node is capable of processing multiple tasks simultaneously (for example, a time-shared computer), this load model exploits concurrency at each node. A simple algorithm has been given to choose the best location for a single task. This may be extended to multiple task placement by repetitive application. It can also be used to handle overloaded nodes by allowing processors to "shed" tasks.

This load balancing system has been applied to a loosely-coupled distributed system using IBM mainframe computers running the VM operating system. A monitoring subsystem was designed to measure the capacities of each processor and the requirements of each task. The load balancing system was implemented, and test results obtained which verified the advantages of this method of load balancing.

11.1 Results of this Study

Load balancing is not a new topic. There has been considerable research in the area, but only casual mention has been made of balancing based on different types of processing power. This work provides a framework for load balancing based on multiple types of processing power.

In addition, little work has been done on determining the load imposed by tasks. This thesis describes a monitoring system, which dynamically determines the characteristics of tasks and processors relevant to load balancing. This enables an automatic load balancing system to exploit differences in tasks and processors.

11.2 Further Work

The load balancing algorithm presented in this thesis is quite naive. There is room for considerable research into better algorithms for load balancing based on multiple criteria for load. There is also room for research in determining task characteristics. While monitoring is a useful technique, it would be more efficient to determine the running characteristics of a task statically. This could be done through analysis of the object code, or as part of the compilation process. The latter could also be useful in optimization techniques; for example, optimizing a task for a processor with limited memory.

Appendix A

Sample Monitor Results

Note that the units are only intended as relative measures. Figures shown for tasks correspond to data used as a *baseline* vector.

Processor idle

	CPU time	*Page wait*	*I/O wait*
Processor	118	0	829

One CPU intensive task

	CPU time	*Page wait*	*I/O wait*
Processor	2076	0	78

	CPU use	*Memory used*	*Memory desired*	*SIO .rate*	*Response time*
CPU task	2738	139	139	0	52

Three CPU intensive tasks

	CPU time	*Page wait*	*I/O wait*
Processor	3677	0	0

	CPU use	*Memory used*	*Memory desired*	*SIO rate*	*Response time*
CPU task	2090	139	139	0	120

Two memory intensive tasks

	CPU time	*Page wait*	*I/O wait*
Processor	460	980	509

	CPU use	*Memory used*	*Memory desired*	*SIO rate*	*Response time*
Memory task	1	1569	1874	0	48

Four memory intensive tasks

	CPU time	*Page wait*	*I/O wait*
Processor	676	4132	34

	CPU use	*Memory used*	*Memory desired*	*SIO rate*	*Response time*
Memory task	0	587	1711	0	480

One I/O intensive task

	CPU time	*Page wait*	*I/O wait*
Processor	317	0	1554

	CPU use	*Memory used*	*Memory desired*	*SIO rate*	*Response time*
I/O task	2	64	64	5561	36

Three I/O intensive tasks

	CPU time	*Page wait*	*I/O wait*
Processor	545	0	2371

	CPU use	*Memory used*	*Memory desired*	*SIO rate*	*Response time*
I/O task	0	173	173	3028	73

Appendix B

Sample Load Balancing Decisions

Note that the units are only intended as relative measures.

Moving a memory intensive task.

Processor 1: Two CPU tasks, one memory task.
Processor 2: Three memory tasks.
Processor 3: Two I/O tasks, one memory task.

	CPU time	*Page wait*	*I/O wait*
Processor 1	1043	0	0
Processor 2	87	318	472
Processor 3	58	0	848

Recommends moving to **Processor 1.**

Moving a CPU intensive task.

Processor 1: Two CPU tasks, two memory tasks.
Processor 2: One memory task.
Processor 3: Two I/O tasks, two memory tasks.

	CPU time	*Page wait*	*I/O wait*
Processor 1	988	0	0
Processor 2	20	2	451
Processor 3	69	0	843

Recommends moving to **Processor 2.**

Moving two I/O intensive tasks and one memory intensive task.

Processor 1: One CPU task, two memory tasks.
Processor 2: One CPU task, one memory task.
Processor 3: Two I/O tasks, two memory tasks.

	CPU time	*Page wait*	*I/O wait*
Processor 1	801	0	0
Processor 2	791	0	174
Processor 3	66	0	727

Recommends moving one I/O intensive task to **Processor 2** *and the other tasks to* **Processor 1.**

References

[Aghili 83] Houtan Aghili, et al.
A Highly Available Database System.
In *Digest of Papers: COMPCON*, pages 9-11. March, 1983.

[Bokhari 79] Shahid H. Bokhari.
Dual Processor Scheduling with Dynamic Reassignment.
Transactions on Software Engineering SE-5(4):341-349, July, 1979.

[Bokhari 81] Shahid H. Bokhari.
Shortest-tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System.
Transactions on Software Engineering SE-7(6):583-589, November, 1981.

[Chou 83] Timothy C. K. Chou.
Load Redistribution Under Failure in Distributed Systems.
Transactions on Computing C-32(9):799-808, September, 1983.

[Chow 79] Yuan-chieh Chow and Walter H. Kohler.
Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System.
Transactions on Computing C-28(5):354-361, May, 1979.

[Clark 85] David D. Clark, et al.
The LCS Common System.
, Massachusetts Institute Technology Laboratory for Computer Science, 1985.
Proposal to Defense Advanced Research Projects Agency.

[Cristian 85a] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev.
Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement.
In *Proceedings of the 15th annual International Symposium on Fault-Tolerant Computing*, pages 200-206. IEEE Computer Society Press, June, 1985.

[Cristian 85b] Flaviu Cristian.
On Exceptions, Failures, and Errors.
Technique et Science de l'Informatique 4(4), Dunod, Paris, 1985.

[Fischer 83] M. Fisher, N. Lynch, and M. Paterson.
Impossibility of Distributed consensus with one Faulty Process.
In *Proceedings of the 2nd Symposium on Principles of Database Systems.* 1983.

[Ford-Fulkerson 62]
L. R. Ford, Jr. and D. R. Fulkerson.
Flows in Networks.
Princeton University Press, 1962.

[Garey 79] Michael R. Garey and David S. Johnson.
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W. H. Freeman and Company, 1979.

[Hofri 78] M. Hofri and C. F. Jenny.
On the Allocation of Processes in Distributed Computing Systems.
Research Report RZ 905, IBM, May, 1978.

[IBM 82] *VM/SP Data Areas and Control Block Logic Vol. I (CP)*
1982.

[Kar 84] Gautam Kar, Christos N. Nikolaou, and John Reif.
Assigning Processes to Processors: a Fault Tolerant Approach.
In *Fourteenth International Conference on Fault Tolerant Computing*, pages 306-309. IEEE Computer Society Press, June, 1984.

[Kuck 78] David J. Kuck.
The Structure of Computers and Computations.
John Wiley & Sons, New York, 1978.

[Morgan 77] Howard L. Morgan and K. Dan Levin.
Optimal Program and Data Locations in Computer Networks.
Communications 20(5):315-322, 1977.

[Rao 79] Gururaj S. Rao, Harold S. Stone, and T. C. Hu.
Assignment of tasks in a Distributed Processor System with Limited Memory.
Transactions on Computing C-28(4):291-298, April, 1979.

[Stankovic 85] John A. Stankovic.
An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling.
Transactions on Computers C-34(2):117-130, February, 1985.

[Stone 77] Harold S. Stone.
Multiprocessor Scheduling with the Aid of Network Flow Algorithms.
Transactions on Software Engineering SE-3(1):85-94, January, 1977.

[Stone 78] Harold S. Stone.
Critical Load Factors in Two-processor Distributed Systems.
Transactions on Software Engineering SE-4(3):254-258, May, 1978.

[Strong 85] Ray Strong.
Problems in Fault Tolerant Distributed Systems.
In *COMPCON*, pages 300-306. IEEE Computer Society, February, 1985.

[Theimer 85] Marvin Theimer, Keith Lantz, and David Cheriton.
Preemptable Remote Execution Facilities for the V-system.
Technical Report, Stanford University, March, 1985.