DYNAMIC SCHEDULING ALGORITHMS FOR DISTRIBUTED SOFT REAL-TIME SYSTEMS

by

Hung-Yang Chang

Computer Sciences Technical Report #728

November 1987

Dynamic Scheduling Algorithms for Distributed Soft Real-Time Systems

by

HUNG-YANG CHANG

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1987

ABSTRACT

This dissertation is a study of **dynamic scheduling** algorithms for **distributed soft real-time** systems with non-periodic workloads. Soft real-time systems permit deadline misses with the performance goal of minimizing the *deadline miss ratio*, the percentage of jobs missing their deadlines, and meeting the *global priority* requirement. In a distributed system with non-periodic workloads, dynamic assignment of tasks to processors can significantly improve system performance. However, job transfers and negotiations between processors exact a cost from the system, limiting performance improvement. Therefore, these algorithms must meet the real-time goals while minimizing the scheduling overhead.

In order to minimize the global deadline miss ratio, an algorithm must employ a negotiation protocol to coordinate schedulers. We devise a *triage* protocol that attempts to transfer a job when the job cannot meet its deadline locally but can meet it at a remote processor. Based on this protocol, three initiation approaches are compared. The receiver-initiated approach is found to be most robust, whereas the sender-initiated approach may cause thrashing when the system load is high.

Global priority scheduling attempts to ensure that the N jobs being served at any moment in the N-processor system are those having the highest priorities. If a job that is waiting has higher priority than the one that is running at a remote processor, the waiting job should be transferred to receive immediate service. The tradeoffs between reducing overhead and relaxing the global priority requirement are explored by comparing four algorithms, each exhibiting a different degree of compromise.

For jobs consisting of a group of cooperating tasks, a global priority scheduling algorithm should take job structure into account. We devise algorithms that give compatible priority service to tasks belonging to a single job, so that cooperating tasks may progress at the same rate, avoiding excessive blocking due to synchronization. Under the assumption that there are two priority levels, our algorithms ensure that tasks are assigned to processors with the lowest priority loads. In total, we present six placement algorithms, evaluated under a workload consisting of jobs having a pipeline structure.

ACKNOWLEDGEMENTS

First, Prof. Miron Livny, my advisor for the last three years, has provided persistent and generous support for this research: he convinced me to focus on the scheduling issues of distributed real-time systems, taught me the trade in performance evaluation, and most importantly helped me organize my often wild ideas.

Prof. Marvin Solomon was a constant source of advise and encouragement throughout my graduate study. He made perceptive criticisms that clarified and strengthened the thesis. He and prof. Raphael Finkel were successful in instilling in me an appreciation of distributed operating systems during the early years of my graduate study. Special thanks to them for the invaluable experience of working on the Charlotte project. Prof. Mary Vernon squeezed in time to read the thesis and made many insightful suggestions. Thanks are due to Profs. Barton Miller and Arne Thesen for serving as my committee members and providing their comments.

Wei-Chung Hsu, Rong-Jay Chen, Yeshayahu Artsy and CuiQing Yang were longterm passengers of the same voyage. They deserve special thanks for offering companionship, fruitful conversations and pleasant distractions. Matt Mutka, Phillip Krueger and Greg Brewster helped in discussing with me issues arising in the course of my research. Phil read the early draft of the thesis; without him I would still be hunting and pecking in Chapter One. Thanks are due to IBM for providing me with a summer internship and graduate fellowship for the last two years of my study.

On a more personal level, I wish to thank my parents for the many years of support they have given me. They have labored a lot to get me to this point. Finally, to my wife Po-Hui goes my deepest appreciation for companionship during my years in Madison. Her encouragement and understanding during the last few months of my graduate study are especially appreciated. It is to her that I dedicate this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

A real-time system is a computer system whose jobs have timing constraints. A distributed real-time system can be exemplified by a large distributed control system, consisting of controlled elements scattered in geographically different locations. For instance, the control systems in the space shuttle [Spector84] and the hexa-leg moving vehicle [Shwan87] contain numerous, scattered devices to be monitored and controlled. There are several reasons to distribute the control in such systems. Distributed control systems have fewer cables, are more reliable, cost less, and provide an easier path for incremental growth. Finally, such systems permit parallel computation.

A distributed real-time system designer is faced with a stringent set of performance requirements. The design must endeavor to meet timing requirements and resource constraints as well as recover from timing violations. Depending on how much a system tolerates violations of timing constraints, real-time systems can be classified into two types: *hard* and *soft*. A hard real-time system cannot tolerate a single violation of a timing constraint. In contrast, a soft real-time system tolerates timing violations and allow jobs to be prioritized to give priority to jobs with greater importance while minimizing the total number of deadline misses.

Although a soft real-time system is designed to tolerate deadline misses, the extent to which jobs miss deadlines is an important concern for the designer. A high deadline miss ratio requires sophisticated compensating mechanisms to cope with functions that are not accomplished in time. Furthermore, it is crucial to give priority to those jobs essential to the distributed soft real-time system so that the system may remain functional even if the system is overloaded. Therefore, how to serve priority jobs while minimizing deadline miss ratio is an important resource management problem for distributed soft real-time systems.

Numerous centralized scheduling algorithms have been developed to meet the deadline demands and priority requirements of real-time systems. Moreover, most of the proposed algorithms are dealing with periodic workloads. These deadline and priority scheduling algorithms have enabled real-time systems to meet the external timing requirement [Baker85] [Jaisal78] in a single processor environment.

Studies of distributed deadline and priority algorithms, by contrast, have been limited. Distributed deadline algorithms have been studied only for a *hard* real-time environment. And distributed priority scheduling algorithms have never been addressed. This thesis presents distributed scheduling algorithms that meet the objectives of deadline and priority for dynamic workloads in soft real-time environments. The challenge is not only to devise new scheduling algorithms, but also to classify and evaluate them.

Furthermore, computation structures may be divided into two types — sequential and parallel [Andrew83]. A sequential computation consists of one thread of computation, whereas a parallel computation contains multiple threads that communicate and synchronize with each other. Since a distributed real-time system permits both types of computations, this thesis examines the distributed scheduling problem of both sequential and parallel computations.

The study of this thesis can be divided into three parts:

(1) **distributed deadline** scheduling,

(2) **distributed priority** scheduling for **sequential computations**, and

(3) **distributed priority** scheduling for **distributed computations.**

The major goal of this thesis is therefore to devise and identify feasible scheduling algorithms to provide deadline and priority service in global resource management. Before we proceed to discuss the three distributed scheduling problems, we will examine distributed soft real-time systems and their distributed resource management problems.

1.1 Distributed soft real-time systems

Timing constraints are of particular importance to real-time systems. A timing constraint is an assertion that imposes a deadline on the execution of an operation. An operation may be periodic or sporadic [Dorathy82]. A real-time system violates a timing constraint if any job misses deadline. Because the system cannot serve all jobs in time, deadline miss will occur when a system is overloaded. And because of the likelyhood of transient overloads when job arrivals are sporadic, timing violations are almost inevitable in distributed real-time systems that control large and complex physical systems under various physical constraints[1].

A hard real-time system cannot tolerate a single violation of a timing constraint. Consequently, such systems require a large reserve capacity to ensure all jobs meet their time constraints under the worst possible load. When there is no way to establish an upper bound on the offered load, a rejection mechanism must be provided in a hard real-time environment. If there is a possibility that the job may not meet its time constraint, the system must reject a job upon its arrival. Ramamrithm *et al.* [Ramam84] and Zhao *et al.* [Zhao85] were the first to study this approach and to examine ways of reducing the rejection ratio of submitted jobs. However, since this approach must serve jobs in a first-come-first-guarantee basis, it is possible that an incoming job that is rejected due to the possibility of transient overload may be the most important job, the one which is invoked to remedy the cause of a transient overload.

In contrast, a soft real-time system tolerates timing violations and allows jobs to be prioritized according to their contribution to the continuing function of the system. Therefore, when confronted with an overwhelming load, a soft real-time system can choose to retain jobs that are essential to the system, allowing the system to remain functional under an overloaded condition. Since the increasing complexity in modern real-time systems entails high probability of transient overloads, this unique capability of a soft real-time system to remain functional under timing violations is crucial for a system to be fault-tolerant.

System Type	**Hard Real-Time**		**Soft Real-Time**		
Job Rejection	No	Yes	Yes	Yes	No
Decision Point	--	Job Arrival	System Rescheduling	Job becomes late	--
Performance Metrics	Pass/Fail	Rejection Ratio			Miss Ratio and lateness

Table 1.1: Real-time Performance Requirements

Table 1.1 summarizes the alternative methods of real-time systems to cope with timing violations and to measure their performance. Soft real-time systems can be divided into three types depending on how a timing violation is handled. In a system of the first type, a job is rejected when the system predicts that the job will be late. A job can be rejected upon arrival or when the system schedules a new job, since the new job may cause previously admitted jobs to miss deadlines. In the second type of system, a job is rejected when the job actually misses its deadline. Since previously admitted jobs may terminate early, this approach maximizes the duration of the scheduling period for the job, increasing the probability that the job meets its deadline. In the third type of system, a job is continued until completion even after it becomes late. Although a job is late, its result may still be valuable. For example, the late result of an

[1] Physical limitations include constraints of heat dissipation, weight and size of physical machines.

image analysis in a robot-eye system can be used to retrofit the current arm position. For a general model of time-varying value of soft real-time jobs with deadline constraints, see [Jensen85].

With the increasing complexity of distributed real-time systems, we believe that the extra flexibility of permitting deadline misses will be needed to cope with transient overloads. Although there has been much research on distributed resource management for non-real-time systems, there has been relatively few studies on managing resources for distributed soft real-time systems. The next section provides a general view of distributed resource management problems.

1.2 Distributed Resource Management

A resource manager is designed based on the type of resource to be managed, the workload to be served, and the underlying control mechanism provided by the system. Since a distributed system can be characterized by resource multiplicity, concurrent workload and decentralized control, the design of a distributed resource manager is more complex than its centralized counterpart.

(1) *Resource* multiplicity: Multiple instances of the same resource type (e.g. processor) coexist within the system. The load of one instance can be transferred to others.

(2) Concurrent *workload*: Besides the general workload pattern arising from the environment (e.g. a data processing system may be I/O bound), the pattern of communication and synchronization between processes contributes to the dynamics of the workload. The management of resources must take into account the relationship between processes to avoid deadlock and to improve response time.

(3) Decentralized *control*: Owing to the overhead of remote communication and the locality of processes' resource demands, it is often natural to decentralize the control mechanism of resources in a distributed system. With decentralized management, the cooperation between replicated managers and the efficient execution of a global policy become critical. Techniques are needed to make decentralized control as efficient as possible.

These characteristics make distributed resource management a very challenging problem. There are many kinds of resources in a distributed system including files, processors, communication devices, and printers. Each of these resources has a unique set of management requirements. Our focus in this study is on processor management.

Replicated processor managers are assumed to be resident on all machines. These managers may consult with each other in deciding *where* and *when* to run a process. The question of *where* involves two basic features: process placement and process migration. The question of *when* has to do with local process scheduling. These three features—placement, migration, and local scheduling— are the basic mechanisms that can be controlled by the replicated processor managers.

A. *process placement*

Process placement addresses the question of where to place a newly invoked job. Depending on the system, various constraints can be added to the placement feature. However, the problem of finding an optimal placement of processes is in general NP-complete assuming *a priori* information [Coffman79]. Without such information, there is no way to prove the optimality of a scheduling algorithm.

B. *process migration*

By process migration we mean the transfer of a process from one processor to another after it has begun execution. The migration of a process is not trivial. Besides having to copy the context and core image of a process, all I/O and interprocess communication connections must be reconnected to the new host. Examples of such facilities can be found in distributed operating systems such as Demos/MP [Powell83] and Charlotte [Artsy86].

C. *local scheduling*

Local scheduling selects the next process to be served by the local processor. The scheduling policies can be either preemptive such as Round-robin, or non-preemptive such as First-come-first-served. The performance objective of a system determines the discipline to be selected. For example, to provide fair service to users, a time-sharing system often uses Round-robin discipline for preemptable resources and First-come-first-served discipline for nonpreemptable resources [Kleinrock76].

These features of distributed processor management should be integrated together according to the real-time objective of the system.

1.3 Distributed Deadline scheduling

The goal of distributed deadline scheduling is to minimize the deadline miss ratio in a distributed system. Although an optimal scheduling algorithm to minimize deadline miss ratio is available for a single queue [Moor69], it cannot be extended to distributed queues directly. Our algorithms attempts to complement the local deadline scheduling by ensuring that when a job cannot complete on a processor in time but can meet its deadline on another processor, the job is transferred to the latter processor for execution.

Our algorithms employ a "triage" selection rule, in which jobs are classified into three categories: *blessed, hopeful, and late.* A blessed job is able to meet its deadline locally, a hopeful job cannot satisfy the time constraints locally but is not yet late, and a late job has passed the possible point of meeting its deadline. A hopeful job is a candidate for migration to another computer so that it can be saved from missing its deadline. The impact of sender-initiated, receiver-initiated, and symmetrically-initiated approaches were evaluated using this selection rule. Our results show significant improvement of deadline miss ratio over a system that does not dynamically share loads among computers. The receiver-initiated approach is found to be more suitable for real-time systems because of its stable response to dynamic load change.

1.4 Distributed Priority Scheduling

Distributed priority scheduling aims to provide prioritized services to the users of a distributed system. Owing to the multiplicity of queues, the meaning of priority in a distributed system is not well defined. Therefore, a job with a higher priority cannot be guaranteed a better treatment over a lower priority job. To address this issue, a notion of global priority is introduced in this thesis.

Since the notion of *global priority* implies a total ordering of jobs in the system, this ordering dictates that the N highest priority jobs should be served in an N computer system. However, a straightforward implementation using a central agent to dispatch jobs does not work well. Although such approach guarantees the enforcement of the total ordering of jobs, its scheduling may entail significant overheads. The overheads arise from both the serialization of control and the excessive job transfer needed to preserve the total priority ordering.

We devised a family of algorithms to support the global Preemptive Resume discipline for two priority levels. Relaxation of the ordering requirement of global priority was found to improve the average response time while reducing the performance gap between priority classes. The priority scheduling algorithm that reduce scheduling overhead while guaranteeing better service for high priority jobs has been identified.

1.5 Priority Scheduling for Distributed Computations

Jobs in real-time systems can be divided into two categories: control and computation. A control job interacts with devices to dispatch commands and to receive status reports. A computation job performs time-consuming numeric operations, such as transforming signals from the amplitude domain to the frequency domain (Fourier transformation). Advances in distributed real-time systems have raised the possibility of improving the performance of both types of jobs through the use of distributed computations.

Cheng *et al.* [Cheng87] has studied the problem of scheduling a special type of task graph that solely contains precedence constraints for *hard* real-time environments. In contrast, this thesis addresses the problem of scheduling cyclic distributed computations for a soft real-time environment with priority requirements.

To support priority scheduling for distributed computations, a notion of *Composite Priority* is introduced to distinguish between job priority and task priority, in which job priority implies a global ordering of distributed jobs and task priority implies an ordering of tasks relative to other tasks belonging to the same job. Owing to the limited scope of this study, this thesis touches upon the issue of assigning task priority and focuses on priority scheduling problems for jobs with a pipeline structure.

This thesis partially concentrates on the scheduling problem of initial task placement with an aim to giving high-priority jobs the best possible service without starving low priority jobs. In stead of dedicate processors for each job, job mixing is used as the basis for priority scheduling. Priority-based mixing improves the response time of both low and high priority jobs because it reduces the interference between jobs. Furthermore, using knowledge of the pipeline structure, we are able to devise algorithms that improve over random placement under various system conditions.

1.6 Thesis structure

The thesis is structured as follows: The next chapter surveys the related work in three areas: central priority scheduling, dynamic scheduling for hard real-time systems, and load distributing algorithms for general-purpose distributed systems. Chapter Three presents distributed deadline scheduling algorithms and their performance. In Chapter Four, we examine distributed priority scheduling algorithms and compare their performance under a variety of system configurations. Chapter Five demonstrates the scheduling problem and the solutions of providing priority services to parallel computations. A discussion of simulation environments with a focus on a new language that permits the specification of parallel computations is put forth in Chapter Six. Finally, our conclusions and future research direction are presented in Chapter Seven.

Chapter 2
Related Work

This thesis draws upon results in the following three areas: scheduling under hard real-time constraints, centralized priority scheduling, and distributed scheduling for time-sharing systems. Since this study considers scheduling problems in soft real-time systems, we examine scheduling problems in hard real-time systems for background information. Moreover, our study on distributed priority scheduling is based on centralized priority scheduling. Therefore, the scheduling results for centralized priority are surveyed. Furthermore, the research in distributed scheduling for time-sharing systems provides us with a framework of distributed resource management.

2.1 Scheduling under hard real-time constraints

With hard real-time constraints, tasks *must* meet deadlines. To meet this challenge, optimal algorithms have been identified for a range of systems. But owing to the assumptions made in theoretical studies (e.g. no I/O requirements), they are not used in practice. Instead, existing practices use ad hoc methods to manually craft the scheduling of real-time operations. A typical method [Spector85] divides time into slots of different sizes. These slots are often called major cycles and minor cycles, where a major cycle is the length of longest periodic operation. A major cycle consists of many minor cycles. To schedule real-time operations is to hand-place operations into slots so that all the deadline requirements can be met. It is often necessary to manually slice an operation into small pieces of execution sequence to fit into minor cycles. Although these ad hoc methods are difficult to use and generate codes which can not be maintained easily, they represent a workable solution method to schedule limited resources under severe constraints.

The existing gap between the theoretical studies and the daily practice of real-time scheduling can only be shortened by more research in the real-time scheduling area. We survey previous results in three types of system configurations: single processor, multiprocessor, and distributed computer network.

2.1.1 Single processor systems

For a fixed set of jobs with the same release time, two scheduling algorithms are found to be optimal in the sense that if any job schedule allows the system to satisfy deadline constraints then these two algorithms must also allow it [Mok78]. These algorithms are earliest deadline scheduling (scheduling the task with earliest deadline, ED) and least laxity scheduling (scheduling the task with the lest difference between its deadline and computation time, LL). However, for jobs with arbitrary release time (allowing dynamic job arrival), the deadline scheduling problem is NP-complete [Garey79].

Liu and Layland [Liu73] derived necessary and sufficient conditions for scheduling periodic tasks with deadlines. The result was then extended to multiprocessors by the same authors. Later, Sha [Sha86] further extends Liu's results to include aperiodic tasks under restricted load conditions.

Teixeira [Teixeira78] developed a model for static priority assignment that addresses external interrupt, scheduling overhead, and preemption. The static priority scheduling is sub-optimal in terms of the resource needed to allow a set of tasks to meet their deadlines, but since the static priority can be associated with the hardware interrupt facility, it takes considerably less overhead than the optimal scheduler.

When conditions do not permit all jobs to meet deadlines, many algorithms are available to minimize the loss of missing deadlines. Moor [Moor69] developed an optimal scheduling algorithm to minimize the number of jobs that miss deadlines. Lawler [Lawler77] devised an optimal algorithm to minimize the average lateness of the system.

2.1.2 Multiprocessor systems

Mok and Dertouzos [Mok78] described multiprocessor scheduling in a hard real-time environment by a game in which tokens are moved on a coordinate system defined by laxity and computation time. They found that no scheduling algorithms can be optimal in multiprocessor systems without a-priori knowledge of task deadlines, computation times, and start times.

Muntz and Coffman [Muntz70] developed an efficient algorithm to determine minimal-length preemptive schedules for tree-structured computations in multiprocessor systems. To plan for the capacity of a real-time system, Leinbaugh developed an analysis method to determine whether systems with a given set of resources can guarantee the response time of real-time tasks. This approach makes conservative estimates on how tasks interfere with each other [Leinbaugh80]. Therefore, the capacity found to be necessary to guarantee the performance of a real-time system is an estimate.

2.1.3 Distributed computer systems

Ramamritham *et al.* [Ramam84] have proposed a bidding algorithms for distributed hard real-time systems. The central notion of this algorithm is to *guarantee* at arrival time that a job meets its deadline. The job is *rejected* when such a guarantee can not be given. An algorithm based on the bidding scheme is used to search for one computer that can guarantee the desired deadline.

The guarantee mechanism was extended by Zhao [Zhao86] to include a heuristic searching procedure that schedules jobs having multiple resource demands. Zhao *et al.* [Zhao85] also proposed a focused addressing mechanism to select bidders according to the past bidding experience. The method significantly cuts down bidding overhead and increase the success rate of bidding when the load is highly imbalanced and the communication overhead is very high. Cheng *et al.* [Cheng86] extended the bidding paradigm to include clusters of tasks having precedence constraints. The preallocation scheme that partitions tasks into groups of tasks having adjacent deadlines is interesting.

2.2 Centralized priority scheduling

Priority scheduling consists of mechanisms and techniques to provide preferential services for customers. Priority scheduling can be used to either maximize system profit or minimize system cost. For example, Kleinrock [Kleinrock76] solved an optimum bribing problem, in which a process is allowed to "buy" his relative priority by means of a bribe. By assigning priorities according to the unit of bribes, the system is able to optimize the total bribes received. Alternatively, when the cost of waiting for a service varies for each customer, a scheduler can reduce the overall system cost by assigning priority according to the cost per unit of waiting time. Therefore, a priority index, which is a quantified *measure of importance* attached to a job, is a useful token for scheduling a resource. The policies that use such priority indices to schedule the next customer for service are priority scheduling policies.

Any priority scheduling policy should offer specific rules so that the scheduler knows:

- Which task to select next.
- Whether to preempt or to continue the current service when a new task arrives.

These rules can either be exogenous, i.e., depending solely on the extrinsic priority class of a task; or they can be endogenous, i.e., based upon both current state of the system and the priority of a task. In the exogenous case, the priority of a job strictly determines its scheduling order, while in the endogenous case, priority is considered to be a hint rather than absolute.

Under an exogenous priority scheduling, tasks of higher priority receive service prior to those of lower priority. Within a priority class, First-come-first-serve (FCFS) is often used as a supplemental rule to break ties. Two alternatives are available to determine when a high priority job receives service if it arrives when a low priority job is in progress: Head-of-line(HOL) and Preemptive-resume(PR). Under the HOL discipline, a newly arrived job is inserted into the priority queue but does not receive service until the server completes its current job. In contrast, under the PR discipline, current job is immediately interrupted to let the server start the new job. Both disciplines maintain a priority queue with newly arrived jobs being inserted into the queue according to their priority ordering. Jaiswal [Jaiswal68] surveys the analytical results of HOL and PR disciplines comprehensively. The results are based on a model of single server systems with Poisson arrival and arbitrary service-time distributions.

Endogenous priority disciplines are introduced to meet other system objectives. Problems that may be addressed by such disciplines include how to avoid an excessive preemption overhead and how to prevent starvation. The assumption that preemption requires negligible time is not always valid. For example, the preemption of processes in a large vector machine has higher overhead than in a scalar machine. Coffman [Coffman76] defined a priority discipline that decides whether to preempt a job depending on its remaining service time. If the remaining time is below a threshold the system waits until the current job completes. The threshold is a parameter to control the total cost of preemption. To prevent the starvation of low priority tasks, Jackson [Jackson60] proposed a time-out priority rule that increases the priority of a long-waiting task when its waiting time exceeds a constant. Such endogenous decision rules, though more difficult to analyze than exogenous rules, are more important for our study and more useful in practice. Deadline scheduling is an example of such rules.

Previous results of priority scheduling are restricted to the case of a central priority queue. Some of these results are based on the multiserver model. Buzen and Bondi [Buzen83] have derived the response time of priority classes under PR scheduling in an M/M/m system with a centralized queue. For nonpreemptive multiprocessor systems having two priority classes, to reserve sufficient processing capacity for high priority jobs, new arrivals of low priority jobs are not allowed to begin service if less than a fixed proportion of processors are idle. This fixed proportion is defined as a "cut-off" point for the priority system. Benn [Benn66] has studied the performance of this "cut-off" priority scheduling system.

2.3 Distributed Scheduling for time-sharing systems

The research in distributed scheduling can be classified into several areas: static task placement, dynamic scheduling for sequential computations, and dynamic scheduling for parallel computation. These areas differ in terms of workloads and performance goals.

Early efforts in static task placement were primarily intended to optimize the response time of a single distributed computation [Stone79] [Ma82] [Rao79] [Gylys76] [Bokhari81] [Ousterhout80]. That is, a parallel computation which is composed of two or more concurrent tasks is to be placed in an exclusive multi-computer environment. Since each task requires both cpu and communication services, the placement must maximize parallelism while minimizing communication delays. Several solution methods have been identified, including graph theoretic, mathematical programming, and fast heuristic methods. These methods are applicable only with *a priori* knowledge of the resource demands of the parallel computation.

General-purpose distributed computer systems (e.g. Locus [Popek 81]) contain independent streams of sequential computations arriving at individual computers. These systems do not have *a priori* knowledge of jobs' resource demands and must make scheduling decisions dynamically. Algorithms have been developed for such systems to support resource sharing and to improve job response time. These algorithms were evaluated under a wide spectrum of hypothetical workloads [Wangs84] [Livny82] [Krueger84] [Bryant81] [Eager84] [Eager85] [Xu84]. Practical evaluations of some of the algorithms, using trace-driven simulation of Unix systems, confirm the validity of load sharing in general computing environments [Otta86] [Zhou87].

Another important problem is the dynamic scheduling of parallel computations. The problem assumes a time-sharing environment of parallel computation. The target system has a dynamically-arriving job stream, composed of multiple tasks. The system can be either a distributed multi-computer or a multiprocessor with both shared memory and local memory. Among the many aspects of this problem, those touched by previous work are processor allocation, cooperation between local schedulers, and local priority assignment [Tilborg84] [Ousterhout80] [William85].

2.3.1 Static task placement

Given *a priori* knowledge of a parallel computation, one of three methods may be used to find a task placement in a dedicated computing environment: graph theoretic, integer programming, and heuristic.

For graphic theoretic methods, the relationships of tasks in a parallel computation are represented by a graph. The tasks are represented by the nodes of the graph. If two tasks communicate with each other, then an arc connects the two corresponding nodes. The costs of interprocessor communication are the weights of the arcs. Processors are added to the graph as nodes. The execution costs of a task on processors constitute the weights on arcs between the task and processors. It is assumed that there is no concurrency between tasks. Given this task interconnection graph, a standard min-cut algorithm [Ford62] can be used to obtain the optimal task placement which minimizes both interprocessor communication cost and module execution cost [Stone79]. However, this problem becomes intractable if more than 2 processors are included in the graph.

The integer 0-1 programming method uses a graph to describe the communication relationship between processes. In this graph, processors are represented as nodes and communication channels as arcs. Typical branch and bound methods are then employed to solve the process placement problem. Ma [Ma82] has allocated 23 tasks to an indefinite number of processors using this method. The advantage of this method is that engineering restrictions (e.g. a certain job must reside in a certain machine because of the hardware restriction) can be used to prune the branches early so that the solution space does not grow exponentially.

Heuristic methods provide approximate solutions to the process placement problem [Gylys76] [Bokhari81]. These methods start with an initial task allocation and then iterate through evaluations and modifications until the total cost of interprocessor communication and queuing delay converges. It is easy to incorporate various constraints, such as processor preference, in the cost functions. Gylys proposed a clustering technique which uses a distance function as the searching criterion [Gylys76]. Bokhari proposed a scheme to find an optimal mapping on a point-to-point network where each processor is dedicated to one task [Bokhari81]. Since these heuristic techniques require less computation time than optimal methods, they may be used to solve time-critical or large problems that do not require an optimal solution.

2.3.2 Dynamic scheduling for sequential computations

Without a-priori knowledge of the behavior of a process, a resource manager depends solely on the current system state to make a management decision. Since the system state may change dynamically, a resource manager must be prepared to adapt to changes. Moreover, management decisions must be made quickly enough to cope with the changes. For this reason, many potential optimal solutions based on classic mathematical techniques such as integer programming are not applicable. Worse yet, network delay prevents resource managers from obtaining up-to-date information of the system. Therefore, it is necessary to develop algorithms that are adaptive and fast to meet the requirements of dynamic process management.

The objective of dynamic process management is to share loads among computers to meet a performance objective. The objective may be response time or response time ratio. Most previous studies assumed a workload of continuously arriving, independent processes. This workload can be abstracted as an m*(M/M/1) model.

m*(M/M/1) systems

An m*(M/M/1) model consists of *m* single-server stations with each station having exponentially distributed interarrived times and service demands. A network connecting these stations is assumed so that job transfer/migration is possible. This model is suitable for commercially available multicomputer systems (e.g. Lotus [Popek82]). Livny has shown, over a wide range of system sizes and utilizations, that in a m*(M/M/1) system there is a high probability that at least one computer is idle while tasks are queued at other computers [Livny82] [Livny84]. This fundamental observation was supplemented with a variety of load sharing algorithms such as status-broadcast (BST) and poll-when-idle (PWI).

Information and location policy

Since dynamic scheduling relies on the state information gathered across the system, it is crucial to find efficient ways to exchange this information. Two approaches have been proposed to disseminate state information. Each approach is closely related to the location policies of dynamic scheduling algorithms.

The first approach is to periodically exchange state information to estimate average system load. This average plays a key role in the BST algorithm to determine the sender and recipient of a job transfer [Livny82]. A sender is a computer with its load above the system average, while a recipient is a computer with its load below the system average. Since broadcasting in a point-to-point network leads to $O(N^2)$ messages for every state information exchange, improvement is needed to reduce such overhead. Barak [Barak85] developed two information scattering algorithms that require O(N) messages for every exchange. In his first algorithm, every computer sends messages to a fixed set of neighboring computers. In the second algorithm, messages are sent to random computers. It is shown that after D steps, any change in the load of one computer will propagate to all other computers, where D is in the order of $\log_2 N$. Manber *et al.* [Manber87] later improved the first algorithm by reducing the size of the receiver set to one, and yet guaranteeing a dissemination of status information within O(N) steps. However, the necessary condition is to have senders select the next receivers according to an N×N Latin square, in which each column represents a sender, each row represents a step in the sending cycle, and the value in each position is the receiver to be selected by the sender represented by the column.

The second approach is to probe neighboring computers at the last minute before making a load transfer decision. This approach allows the transfer decision to be made within a small part of the system such as the PWI algorithm [Livny84]. Although this type of algorithm does not balance the load among computers, it was shown to perform very well in terms of average response time. Eager *et al.* employed an analytical model to show that a simple location selection policy, searching for a computer with a load higher than a threshold, yields performance close to that of a more complex policy, searching among K computers for the computer with the lowest load (K is a tunable system parameter) [Eager84].

Producer-initiated vs. consumer-initiated

Two fundamental tactics of synchronization can be found in a producer-consumer relationship. The producer can push a commodity to the consumer, or the consumer can pull a commodity from the producer. These two tactics can also be combined so that both sides take initiative in achieving a transfer of commodity. This fundamental dichotomy of initiation methods has been discovered in many disguises such as source-initiated vs. server-initiated, sender-initiated vs. receiver-initiated, and bidding vs. drafting. Although these algorithms deal with different system configurations, they all share the same characteristics.

Wang and Morris [Wang84] classified two approaches of service sharing: source-initiated and server-initiated. In their network service model, a source machine is the source of new requests and a server machine is the machine that provides service to requests. Server and source machines represent physically different machines. Server-initiated was defined as letting the server machine ask for work from the source when the server lacks work to do, while source-initiated was defined as letting the source machine send work to the server machine when the source generates new work. Various types of workloads were examined. A major conclusion from their analytical study is that the server-initiated approach is in general superior.

In an analytical study, Eager *et al.* [Eager85] compared receiver-initiated and sender-initiated load sharing policies using the m*(M/M/1) model. A receiver was defined to be a computer receiving a load transfer. They found that a sender-initiated policy is better than one that is receiver-initiated, except when the system load is high.

Ni *et al.* [Ni83] presented a drafting algorithm that allow a computer to ask for extra work to do from peers when the current load is low. The drafting algorithm monitors the local load condition and labels the computer as having a low, medium or high load. A transition from middle to low loads triggers the negotiation action. Since this algorithm sets off the negotiation for a transfer of load at the consumer (receiver) side, this algorithm can be classified as a receiver-initiated algorithm. In contrast, Stankovic and Sidhu studied an adaptive bidding algorithm for distributed groups [Stankovic81] [Stankovic84]. In the algorithm, a computer sends out bids to other computers to search for surplus capacity to serve a newly arrived job. The computer returning the highest bid gets the job. This approach is sender-initiated because the sender of the job takes the initiative to find a foster computer for the job it can not handle.

Sender-initiated approaches are unstable when the system load is high. Stankovic addressed this issue with examples of several distributed scheduling algorithms [Stankovic85].

Workload considerations

With few exceptions, distributed scheduling algorithms have been evaluated under the assumption of a Possion job arrival process and exponentially distributed service demands. It is important, therefore, to examine these algorithms under practical workloads. Based on the observations of BSD Unix systems, two major discrepancies from previous workload assumptions have been reported [Cabrera86] [Leland86] [Zhou86]. The first discrepancy is that job arrivals tend to be bursty. The second one is that the service demands of jobs have a high coefficient of variation, ranging from 10 to 25. With few very long-running jobs, the system contains a large number of small jobs (~99%) having a lifetime of less than 6 seconds. However, trace-driven simulations of known distributed scheduling algorithms using such a workload were found to confirm the the previous results of these algorithms.

Load sharing vs. load balancing

Distributed scheduling algorithms should be evaluated under different performance objectives. Krueger [Krueger87] studied the performance of a load *balancing* algorithm with an objective of sharing resources equally. This algorithm differs from a load *sharing* algorithm, which has an objective of fully utilizing the resource (reducing the wait-and-idle probability). Actually, the difference between sharing and balancing in terms of mean wait ratio or standard deviation in wait ratio was found to depend on workload. When initiation rates were inhomogeneous or service demands were hyperexponential, balancing was found to be significantly better in terms of both performance indices.

Distributed problem solver

The notion of distributed problem solving in AI is closely related to that of distributed scheduling. Distributed problem solving uses sophisticated local information and meta-level control to infer a global solution state. Distributed scheduling may be viewed as a special example of distributed problem solving [Lesser83]. The results in both areas can cross-fertilize each other. For example, the idea of comprehensive bidding and focused addressing were first proposed as part of a high level protocol for distributed problem solving network [Smith80]. These ideas were then applied in the context of distributed real-time system [Zhao85].

2.3.3 Dynamic scheduling of parallel computations

Ousterhout [Ousterhout80] has studied the scheduling problem of "task force" (or parallel computation as we call it) in a tightly coupled multiprocessor system. The algorithm is based on the principle that the process working set of a computation has to be co-scheduled to minimize process thrashing. Ousterhout develops a matrix structure where all the processes of a task force are mapped to the same row. Scheduling on each processor is synchronized, so that all processors schedule the same row for the duration of a time slot. The performance of this co-scheduling algorithm is evaluated via experimentation. It is unclear how well the notion of co-scheduling is applicable to loosely coupled distributed systems.

Tilborg [Tilborg84] have studied a hierarchical task force scheduling algorithm called wave scheduling. The algorithm assumes that each processor is capable of serving only one process at a time, and a task force is scheduled to run only when all members of the task force can be placed. A scheduling request is passed up and down a hierarchy of schedulers until a scheduler that is able to schedule the whole task force is found. Although the hierarchical algorithm provides a framework for task force scheduling, the question of how to use the framework to satisfy performance objectives such as minimizing response time has not been answered.

Chapter 3
Distributed Deadline Scheduling Algorithms

3.1 Introduction

Owing to the potential for high reliability and high performance, distributed systems provide a promising means for real-time applications [Duffie82] [Scho85] [Bourn84]. In order to realize these potentials, difficult dynamic resource allocation problems have to be solved. Since modern distributed real-time systems are capable of dynamically scheduling jobs in remote processors, such capability must improve the probability that jobs meet their deadlines.

Real-time systems are confronted with different workloads in scheduling jobs for meeting deadlines. Previous research has addressed the problem of real-time scheduling by assuming a cyclic, static workload [Munt70] [Mart82]. In contrast, the study presented in this chapter focuses on a dynamic workload, in which the magnitude of service demands and job arrival times are random variables. Under such a workload, if the dynamic system does not have a finite worst case load, meeting all the deadlines becomes unobtainable. Even if it is obtainable, the system may not have a sufficient capacity to cover the worst case load. Therefore, it is crucial to examine overload situations to come up with systems that are overload-tolerant.

While previous studies have looked into the problem of real-time scheduling under overload situations, they all assume a *hard* real-time environment [Ramam84] [Zhou85]. In such an environment, jobs arrive at an overloaded machine are either rejected or relocated to other machines. Since there are reasons to believe that a soft real-time system permitting deadline misses has advantages over a hard system in dealing with an overload situation, this study examines the distributed real-time scheduling problems for soft real-time systems.

In this chapter we present a dynamic deadline scheduling algorithm for soft real time systems. In a soft real-time system, jobs are allowed to miss their deadlines. The decision of terminating a job or not depends on its lateness and on the time-varying value of the result [Jensen85]. The tradeoffs between different rejection strategies are system dependent. In this study, we do not present a total evaluation of different rejection strategies. We assume that all late jobs are continued until completion. The performance of a soft real-time system therefore depends on its deadline miss ratio and average lateness. In this study, we have used the deadline miss ratio as the primary metric and lateness as a secondary metric. Since no requests are rejected or disposed of, the rejection rate is zero and the actual load is the the offered load of the system.

Our dynamic deadline scheduling algorithm has a two-phase polling strategy and is based on the Shortest-processing-time-first local scheduling policy. Two approachs to load sharing in distributed systems have been identified [Eager85] – Sender-initiated and Receiver-initiated. The exporter of a service demand is viewed as the sender, while the receiver is the importer of the demand. For example, the contract net protocol [Smith80], bidding [Stank84] and the above-average [Krueg84] algorithms are sender-initiated, while the drafting [Ni85] and the Poll-When-Idle [Livny82] algorithms are receiver-initiated. Using the average response time as the performance measure, Eager *et al.* [Eager85] have evaluated the performance of sender-initiated and receiver-initiated load sharing algorithms. They have observed that the receiver-initiated approach is inferior at low loads and superior at high loads. While a sender-initiated bidding algorithm has been evaluated for a distributed real-time environment [Ramam84], the relative performance of the two approaches in a real-time environment has not been profiled.

The real-time scheduling algorithm presented in this chapter is receiver-initiated. Using discrete event simulation, we have analyzed the performance of this algorithm and have compared it with that of a sender-initiated algorithm. The latter algorithm is also based on a polling scheme and is similar to the "threshold algorithm" of Eager *et al.* [Eager84]. The results obtained from this study have demonstrated the stability of the receiver-initiated algorithm, where stability is defined as bounded performance at different loads and overheads [Stank85]. In addition, the sensitivity of the sender-initiated algorithm to variations in load and overhead has been shown. Moreover, we have compared the performance of the proposed algorithms with a simple load sharing algorithm, Poll-when-idle [Livny82], that does not use any deadline information. We find that when load is low, the real-time oriented algorithms have little performance advantage over this simple algorithm.

In Section 2 we present the system model employed in our study. The two deadline-oriented distributed scheduling algorithms are described in Section 3, and the performance results are discussed in detail in Section 4. Conclusions are drawn in Section 5.

3.2 System model

A model of a distributed soft real-time system with a dynamic workload is assumed in this study. It is natural to associate a dynamic workload with a soft real-time system because temporary overload is inevitable when the workload is acyclic. Our system model consists of N homogeneous processors that are interconnected by a local area network. Each processor is subjected to an independent stream of jobs. All jobs are of the same type, and can thus be served by any of the N processors. A job has a *size*, a *processing demand*, and a *deadline tightness* associated with it. The size of a job, which is the size of the image to be transferred for remote execution, and processing demand is independently exponentially distributed. Moreover, the means of job size and processing demand are chosen to be 1. The deadline tightness, which is the maximum amount of time a process can be delayed before completion (not including process execution time), is obtained by multiplying D (the tightness parameter) with the service demand of a job.

The processors of the distributed system are interconnected by a single broadcast communication medium. No specific communication protocol is assumed and the transmission rate of the network is C data units per time unit. A First-come-first-serve queueing system is used to model the communication network. Since the network is unlikely to become a bottleneck, this simplified model of communication channel meets the needs of our study.

The processing overhead of a placement or a migration consists of two elements: the negotiation element and the job transfer element. We assume that the processing time of a negotiation message (note that a message must be processed by its transmitter and receiver) requires a fixed *MsgP* units of time. The size of each negotiation message is fixed at *MsgS* units of data. Since this model does not permit interprocess communication, the overhead of a job transfer mainly comes from moving the core image of a job [Powell83] [Artsy85]. Therefore, it is assumed that the processing time associated with a job transfer is proportional to the size of the job. A job transfer requires *TransP* units of processing time per unit of data transferred.

3.3 Dynamic scheduling algorithms

In a soft real-time distributed system, a job can be in one of the following three states: *blessed*, *to-be-late*, and *late*. A job is *blessed* if it will finish in time. If it is known that the job will not meet its deadline unless a job transfer takes place, the job is tagged as a *to-be-late* job. A job enters the *late* state once it is clear that it will miss its deadline. The state of a job can be determined by considering the processing requirements and deadlines of all the jobs currently placed at the processor.

Each of our algorithms is composed of five policies that represent the five key components of distributed scheduling. These five separable and yet integrated components are the following :

(1) *Local scheduling* determines the sequencing of local jobs.

(2) *Information policy* dictates the method of exchanging load status among processors.

(3) *Initiation policy* decides when to initiate a migration request.

(4) *Candidate selection policy* determines how to choose a job for migration.

(5) *Location policy* defines the terms under which two processors may transfer a job.

Since our algorithms share the same local scheduling and information policies, we will begin by describing these policies. Having done that, we will proceed to describe both the receiver-initiated algorithm and the sender-initiated algorithm with primary emphasis on the other three policies.

Finding an optimum local scheduling policy for a given set of jobs that minimizes mean lateness has been shown to be "NP-complete" [Garey79]. The problem of finding an optimum for a stochastic system with dynamic job arrivals is even harder. Although a pseudo polynomial time algorithm can solve the lateness minimization problem [Lawler77] in the static case, this method is unlikely to be computationally feasible in an open system where decision time is limited. Therefore, we must rely on simple heuristics that find "good" execution sequences quickly. It is known that it is generally best to base priority schedulings on processing and deadline information [Baker84]. Since the purpose of the thesis is to evaluate the performance of the non-local components of a distributed scheduling system, we use the Shortest-processing-time-first (SPT) discipline as the local scheduling policy for all algorithms.

The algorithms proposed in this study collect information about the instantaneous load of remote processors via a polling mechanism. Polling is part of the negotiation process. Each poll requires the exchange of two messages - a polling message and a reply message. No attempt is made to maintain an up-to-date picture of the current load distribution of the entire system. The subset of system processors polled is limited to *K* processors per migration attempt. This polling limit is a tunable parameter of the algorithm.

3.3.1 Receiver-initiated deadline scheduling

Upon the completion of a job, the load of the processor is examined to determine whether the processor is underloaded. When the number of jobs remaining in the queue is smaller than the threshold *T1* (a tunable system constant), the processor is tagged as underloaded. When an underloadeded state is entered, the negotiation module begins polling other processors to offer help. This "poll-when-underloaded" initiation policy is similar to that of the Poll-When-Idle receiver-initiated algorithm [Livny82].

Two-phase polling is used to give higher priority to the most urgent jobs. The degree of urgency of a job is reflected in its state, which shows the likelihood of completing the job in time. A "to-be-late" job is most urgent, because the job may still finish in time if it can be transferred to a processor that can "bless" it. A "late" job is urgent, so that its lateness can be reduced. A "blessed" job is the least urgent, for it can complete in time at the processor. In the first phase of polling, the poller seeks a job that is in "to-be-late" or "late" state. If the poller fails to locate any such job, a "blessed" job is sought in the second phase of polling. This second phase can be viewed as doing anticipatory work [Livny84] to reduce the probability of missing deadline in the future. However, the "blessed" job selected in the second phase must not become "late" due to the transfer delay.

Since an idle or an underloaded computer has excess capacity, an attempt is made to continue the polling effort when the search for a possible migration fails. To prevent the continued polling from interrupting other processors excessively and from flooding the communication network with polling messages, a timer is set to re-invoke the polling after *I* time units. The polling will terminate once the processor is no longer in an underloaded state. The invocation interval is a tunable parameter.

The above polling scheme is similar to the drafting scheme proposed by Ni *et al.* [Ni85]. Both schemes were designed to support distributed scheduling using a receiver-initiated approach. However, they differ in their location policies. The drafting scheme selects a location by broadcasting draft requests and comparing the replies, whereas the two phase polling scheme looks for any location that satisfies the migration requirements. The first-fit approach of the latter reduces its processing and communication overheads and shortens the length of the negotiation process in many cases. It is doubtful that the advantages of finding an optimal location for migration can justify the extra cost.

3.3.2 Sender-initiated deadline scheduling

The job migration mechanism in a sender-initiated approach is invoked when a local overloaded situation occurs. A processor is considered overloaded if one or both of the following conditions are met.

(1) At least one of the jobs in the queue is in a to-be-late or late state.

(2) There are more than *T2* (a tunable system parameter) jobs in the queue.

Upon the arrival of a job, the local scheduler examines the load. If the processor is in an overloaded state, the scheduler triggers the negotiation module to select a migration candidate and to initiate a migration attempt. The heuristic rules for selecting a migration candidate are as follows:

(1) From among the "to-be-late" jobs, select the job having the earliest deadline,

(2) If there are no "to-be-late" jobs, choose a "late" job,

(3) otherwise, from the "blessed" jobs that will not miss their deadlines after the delay incurred in a job transfer, select the job with the longest processing time.

Unlike the bidding algorithm [Stank84], in which only a new arrival is selected to be the candidate for job transfer, our approach selects the job that may benefit most from the transfer. Since the cost of initial placement generally is less expensive than migration, the benefit of migrating an old job may be outweighed by its cost. Our heuristic rules can be modified to give preference to the new arrival if the cost of migrating an old job is significantly higher than the cost of placing the new one.

The receiver accepts a migration request if one or both of the following conditions are met:

(1) the load is lower than the threshold *T1*.

(2) the candidate can be "blessed".

Combining the threshold policy [Eager84] with a deadline test, this location policy accepts a job transfer even if the load is high.

The sender may fail to locate any available processor for this particular candidate when the polling limit is reached. Since the sender needs time to process existing jobs in the processor, the sender-initiated algorithm does not re-invoke itself using an interval timer. In contrast, because the receiver has free capacity to use, it reinvokes the polling after a fixed interval *I* has passed.

3.4 A simulation study

A discrete event simulator was built to evaluate the performance of the two algorithms. The simulator is written in **DENET** [Livny86], which is a modula-2 based simulation language. Each algorithm was simulated for 9000 time units under different job arrival rates. The job arrival rates range from 0.2 to 0.9 jobs per time unit. All other simulation parameters and the values assigned to them are summarized in table I.

Parameter	Value(s)
Computer network size (N)	16
Workload distribution	Homogeneous
Network capacity (C)	200 (unit data / unit time)
Job processing demand	mean 1 (unit time) exponential
Deadline tightness (D)	mean 2 (unit time) exponential
Job size	mean 1 (unit data) exponential
Message size (MsgS)	.01 (unit data) constant
Transfer processing time (TransP)	.1 (unit time / unit data)
Message processing time (MsgP)	.004 (unit time) constant
Polling limit (K)	12 processors
Overload Threshold (T2)	4 (jobs)
Underload Threshold (T1)	1 (jobs)
Invoking interval (I)	.3 (unit time) constant

Table 3.1: Simulation parameters for distributed deadline scheduling algorithms

3.4.1 A Comparison of initiation approaches

Two performance metrics, deadline miss ratio and mean lateness, were selected as evaluation criteria. Deadline miss ratio is the percentage of jobs that do not meet their deadlines. The lateness of a job is the difference between its completion time and its deadline if it misses its deadline, and is 0 if it completes in time.

Figure 3.1 presents the deadline miss ratio of the sender-initiated, receiver-initiated and isolated scheduling algorithms as a function of job arrival rate. There is an interesting crossover point between the curves of sender-initiated and receiver-initiated algorithms. When the rate is less than .75, the sender-initiated algorithm (*Send-Init*) outperforms the receiver-initiated scheduling algorithm (*Rcv-Init*). However, when the arrival rate exceeds .75, the Rcv-Init algorithm becomes superior. Similar behavior was reported by Eager *et al* [Eager85] for load balancing algorithms. Figure 3.1 also contrasts between dynamic scheduling with *isolated* scheduling, under which processors do not transfer jobs. Except when the load is at .9, the deadline miss ratio for isolated scheduling is larger than that of dynamic scheduling algorithms.

The Send-Init algorithm becomes instable at high loads. By instable, we mean the algorithm causes the system to perform worse than when processors do not utilize this scheduling algorithm. Since the activity of Send-Init algorithm increases as load increases, it reaches a point where the cost of such scheduling is greater than the benefit. In contrast, receiver-initiated scheduling provides a built-in mechanism that lowers scheduling overhead as system utilization increases. At low loads, the Send-Init algorithm is more efficient at locating resources and thus achieves a lower deadline miss ratio. Receiver-initiated scheduling is less effective because scheduling is not invoked according to the needs of jobs, and thus the help often comes late.

The mean latenesses of the two algorithms as a function of arrival rate are shown in Figure 3.2. The two curves cross at .6 arrival rate. The mean lateness of each algorithm starts increasing at an arrival rate of .6. Moreover, the Send-Init algorithm increases faster than the Rcv-Init algorithm does. At an arrival rate of .9, the mean lateness of the Send-Init algorithm becomes 40 times that of Rcv-Init algorithm. To demonstrate the scheduling overhead of these two algorithms, Figures 3.3 and 3.4 plot the inquiry rate and the job transfer rate under varying offered loads.

Figure 3.3 shows that the inquiry rate of the Rcv-Init algorithm has a maximum point in the medium load. while the inquiry rate of the Send-Init algorithm increases monotonically. A crossover point between

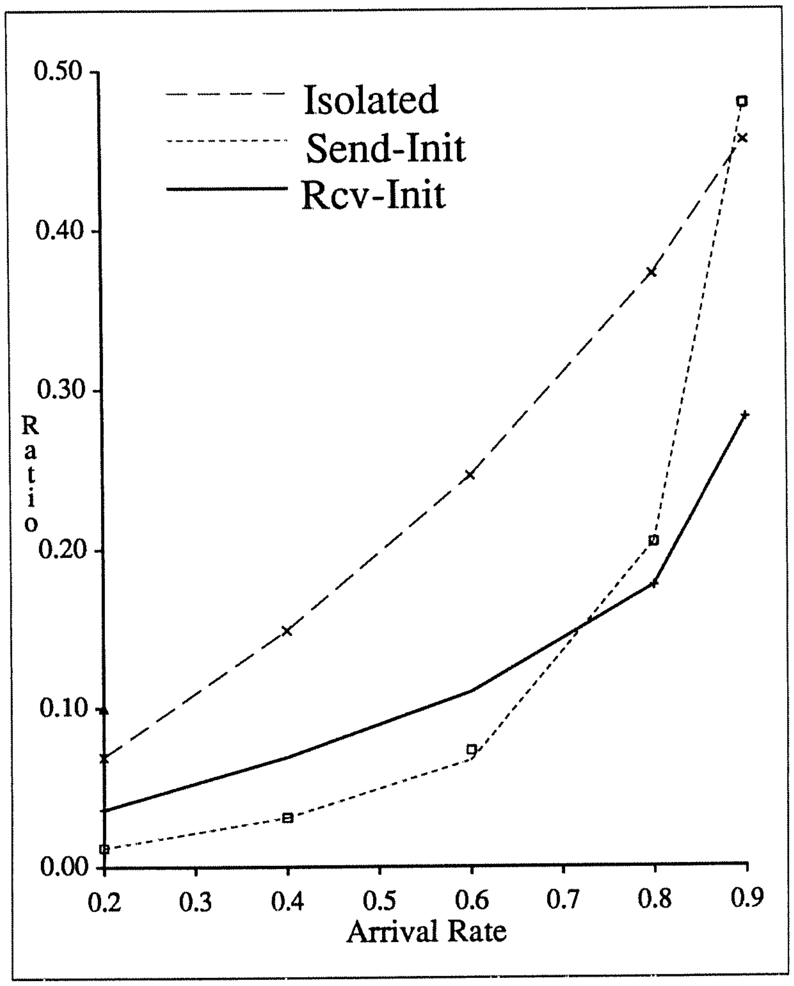

Figure 3.1: Deadline miss ratio by Arrival Rate

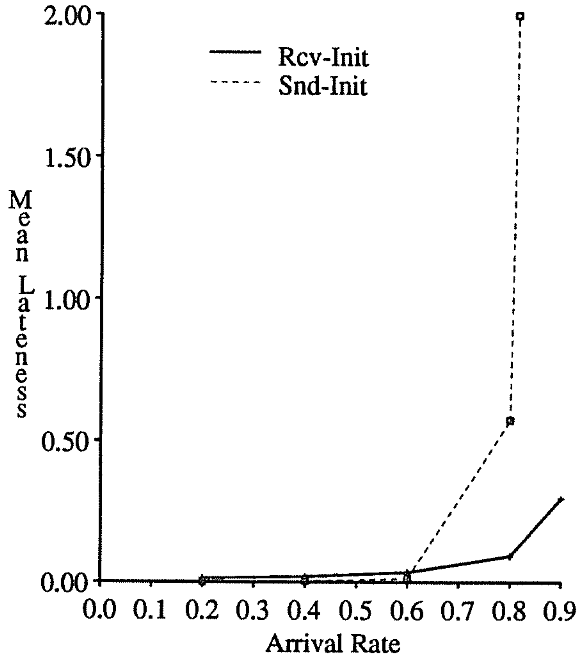

Figure 3.2: Mean Lateness by Arrival Rate

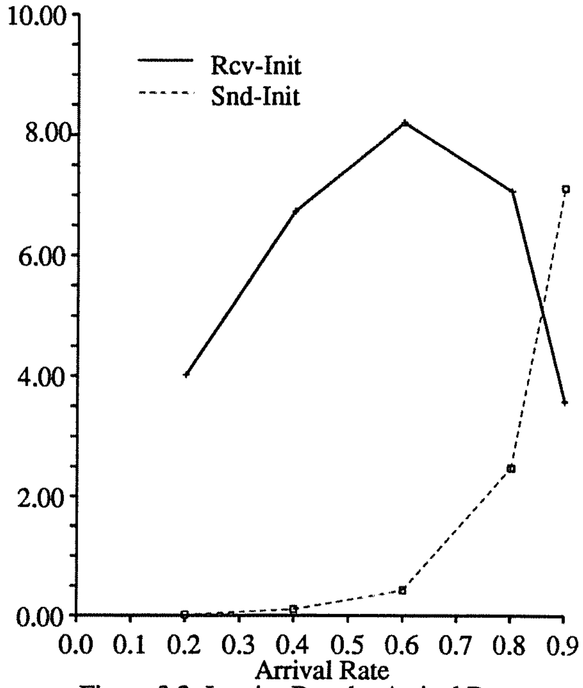

Figure 3.3: Inquiry Rate by Arrival Rate

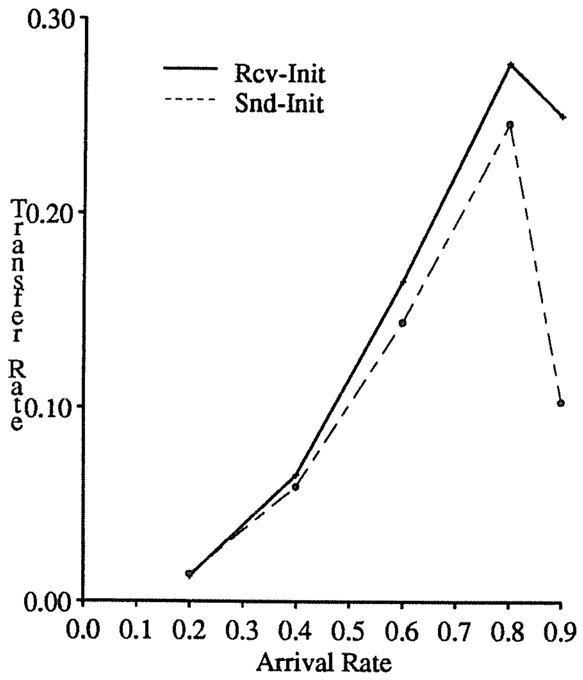

Figure 3.4: Transfer Rate by Arrival Rate

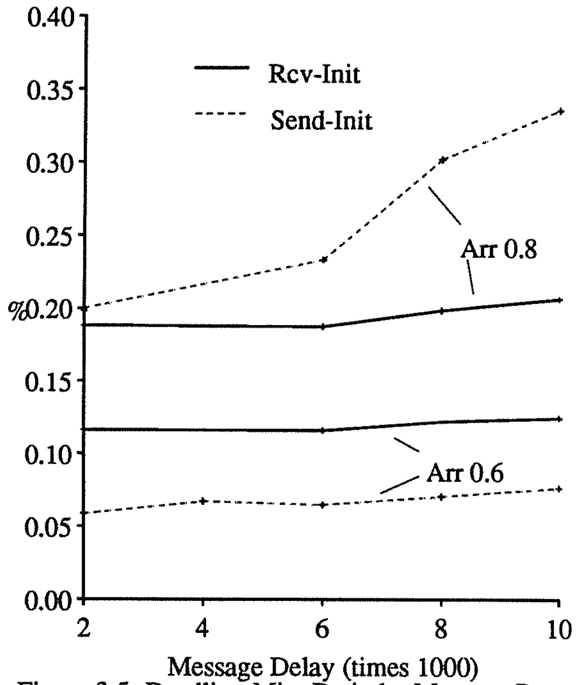

Figure 3.5: Deadline Miss Ratio by Message Processing Time

the two curves occurs at an arrival rate of .85. However, this crossover point occurs at an arrival rate significantly larger than those under the measurement of deadline miss ratio and mean lateness. This incongruity of crossover points will be discussed in detail later. Since the Rcv-Init algorithm is initiated when the transition from a busy state to an underloaded state occurs, it is not surprising that its inquiry rate curve follows the pattern of state-transition probability. The transition probability is low when the job arrival rates are either very high or very low, because the system is mostly empty in the former case and mostly occupied in the latter case. The maximum of the transition probability occurs at medium loads. In Figure 3.4 we find that the job transfer rate of the Send-Init algorithm drops to almost zero at high loads while the Rcv-Init algorithm continues to maintain its activity. Lacking job transfers, the load cannot be shared among processors. From Figures 3.3 and 3.4, we conclude that the instability of Send-Init at high loads is caused by both the high rate of inquiry, which incurs high overhead at a time when system resources are scarce, and the lack of job transfers. The Rcv-Init algorithm, in contrast, remains stable at high loads, with reduced inquiry activity and continuing job transfer activity.

The position of the crossover point between the two algorithms is different for the two performance measures, indicating that the crossover points are at .6 arrival rate for the mean lateness,
.75 arrival rate for the deadline miss ratio, and
.85 arrival rate for the inquiry rate.

A few remarks on the difference of the crossover points under the measurement of deadline miss ratio, lateness, and inquiry rate are in order. While two reasons have been given for the inferior miss ratio of the Send-Init algorithm at high loads, neither of these reasons explains the interval between the crossover point for miss ratio (.75) and the crossover point for inquiry rate (.85). Since the difference of transfer rates between both algorithms changes insignificantly within this rate range, the reason for a lack of job transfers cannot be established. We claim that the biased impact of inquiry overhead in the two algorithms plays the most important role within this rate range. The following reasoning offers the rationale of the claim. Although an underloaded processor has excess capacity to spend, an overloaded processor must send inquiry messages at the expense of current jobs. Therefore, the performance impacts of inquiry overhead to overloaded processors are more severe. In the Send-Init algorithm, an inquiry involves an overloaded processor and possibly another overloaded processor when the inquiry fails, while in the Rcv-Init algorithm, an inquiry is issued by an underloaded processor and taken by an overloaded processor only when the inquiry succeeds. The Send-Init algorithm, therefore, shows inferior miss ratios in this arrival rate range, while at the same time its inquiry rate is lower than that of the Rcv-Init algorithm, and its job transfer rate almost remains almost constant.

3.4.2 Sensitivity to message delay

We have shown that the Rcv-Init algorithm adapts to changes in load under a given set of system parameters. To determine how the performance of both algorithms changes when the processing time of messages increases, we simulating varying message overheads. Message overhead was varied from 0.002 to 0.01 while arrival rate was set to .6 and .8. Figure 3.5 summarizes the results of these simulations, presenting the deadline miss ratio as a function of message overhead. Each set of lines represents the deadline miss ratio for a different arrival rate. The results show that at .8 arrival rate, the Rcv-Init algorithm is much more resilient to the increase in message delay than the Send-Init algorithm. The performance of both algorithms is not affected by the message delay at .6 arrival rate due to the abundant free capacity of the system.

3.4.3 Sensitivity to deadline tightness

In dynamic environments, the expected deadline tightness of a job may change over time. Thus, it is important to evaluate a distributed deadline scheduling algorithm under different levels of deadline tightness. In Figure 3.6, we present the results of a study in which the relationship between the miss ratio and the deadline tightness was analyzed. In this study, the value of the mean tightness is varied from 1.5 to 5.0 (note that average job execution time is 1.0 unit). We present two sets of data, where one is obtained at arrival rate .6 and the other at arrival rate .8.

The Send-Init algorithm is more sensitive to changes in deadline tightness than the Rcv-Init algorithm. In Figure 3.6, we observe that when deadlines become tighter, the deadline miss ratio of the Send-Init algorithm increases faster than the ratio of the Rcv-Init algorithm. Furthermore, we see a crossover point at 3.1 for a system with a .8 arrival rate. This crossover arises because the Send-Init algorithm performs better than the Rcv-Init algorithm when the expected deadline tightness is higher.

3.4.4 A Comparison to a Poll-When-Idle algorithm

The complex decision rules of Send-Init and Rcv-Init incur overheads in both job scheduling and information collection. In order to justify these overheads, the performance of the two algorithms must be compared with that of a simple load balancing algorithm. The Poll-When-Idle (*PWI*) algorithm [Livny82] was chosen as an example of such an algorithm. The PWI algorithm is a receiver-initiated algorithm that does not utilize any deadline information. Whenever a processor becomes idle, it pools other processors in a search for jobs to run. The job chosen to be transferred is the last job waiting in the ready queue. Since jobs are ordered according to the SPT (shortest processing time first) local scheduling policy, this job has the largest processing demand at its processor.

In Figure 3.7, three sets of results are displayed. The solid lines are the percentage of improvement of the PWI algorithm over the Rcv-Init algorithm. The horizontal dashed lines are used to show where the two algorithms perform equally. The value of relative improvement is calculated by subtracting the value of the PWI algorithm by the value of the Rcv-Int algorithm, and then dividing this difference by the value of the Rcv-Init algorithm.

In terms of the deadline miss ratio, the Rcv-Init algorithm is superior to the PWI algorithm at medium to high loads, but is inferior at low loads. The same pattern appears when studying mean lateness. In contrast to the slow miss ratio degradation of the PWI algorithm as the load becomes higher, the fast mean lateness degradation shows that the PWI algorithm is doing more damage to late jobs than to other

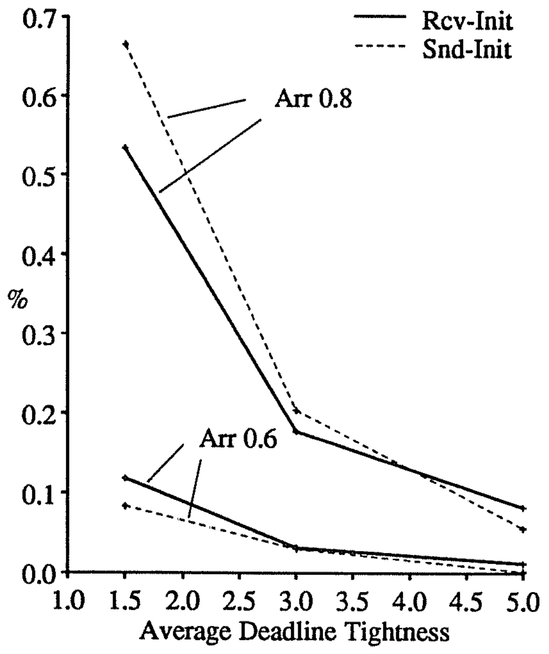

Figure 3.6: Deadline Miss Ratio by Deadline Tightness

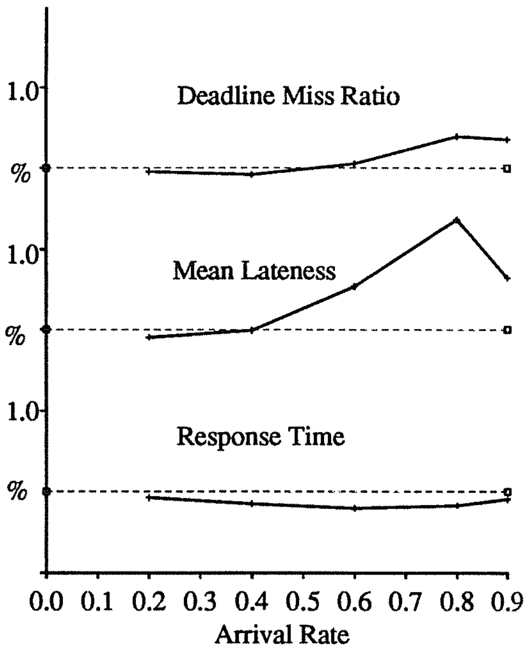

Figure 3.7: The relative improvement of Poll-when-Idle over Deadline Rcv-init

jobs. In terms of response time, the PWI is superior at all ranges due to its simplicity and its low overhead. The above results indicate that an algorithm that does not use any deadline information is worse at meeting deadlines, but better at minimizing response time. Furthermore, although such algorithms do not have enough information to prioritize jobs for deadline performance, at arrival rates below .4, with less resource contention and reduced overhead, the simple PWI algorithm results in better performance than the Rcv-Init algorithms even for miss ratio.

3.5 Summary

The dynamic workload of distributed soft real-time systems introduces new scheduling problems. Previous research in the area of load sharing in an environment free from real-time constraints suggests two basic approaches: Sender-initiated and Receiver-initiated. Among them, sender-initiated scheduling using bidding was evaluated in a hard real-time environment [Ramam84]. Sender-initiated and receiver-initiated approaches have been shown to complement each other. However, the performance of a receiver-initiated approach in a real-time system has not been profiled. This chapter presents and evaluates a receiver-initiated deadline-oriented scheduling algorithm for a soft real-time environment. Because this study addressed a new system environment, a sender-initiated algorithm was also devised for comparison purposes.

The sender-initiated algorithm is instable at high loads, though it is superior at low loads. Instability is caused by the low efficiency of the inquiry process for this type of distributed scheduling mechanism when the system is loaded. The superior performance of sender-initiated approach at low loads is the result of its higher efficiency in serving jobs that need to be transferred. Nonetheless, the performance gap at low loads is small compared to the difference at high loads.

Since a real-time system can afford less-than-best performance at low loads but cannot afford break-down at high loads, the receiver-initiated scheduling approach is clearly best if the distributed real-time system must face fluctuating workloads. Other considerations also favor the receiver-initiated approach. When the message processing overhead increases, which may be the result of a temporary network prob-lem, the receiver-initiated approach performs stably, while the performance of the sender-initiated approach deteriorates. The performance of the sender-initiated algorithm is more sensative to variation in the tightness of deadline than the receiver-initiated algorithm.

We have compared the performance of our deadline oriented algorithms with the Poll-When-Idle load balancing algorithm. The results show that there is no need to use complex deadline algorithms at low loads (0.4 or below) because simple rules are more effective. However, deadline-oriented algorithms pre-vail at medium to high loads because the deadline-oriented algorithms prioritize jobs to optimize deadline performance at the presence of resource contention.

Chapter 4
Distributed Priority Scheduling Algorithms

4.1 Introduction

Priority scheduling is a class of queueing disciplines that provides one group of customers with preferential service at the expense of others. Customers are grouped by their waiting costs to the system or by the system's profit in serving them. Priority scheduling allows a system to tailor its service for different groups of customers so that the system can minimize its cost or maximize its profit. Extending priority scheduling to a distributed environment adds a new dimension—queue multiplicity—to the problem of priority scheduling. This issue of managing multiple queues according to a priority principle has not been addressed before.

Without a centralized queue, the meaning of job priority is not well defined. For priority disciplines such as Preemptive-Resume (PR) and Head-Of-Line (HOL), the priority of a job uniquely determines its position in a centralized queue and thus the quality of its service. In contrast, jobs in physically separated queues receive service according to the loads at individual queues. Although the influence of priority still exists, it no longer plays the role of deciding a unique global ordering of jobs. In essence, the notion of priority needs to be extended for a multiple queue environment. The goal of this chapter is two-fold. First, the meaning of priority service in a distributed system is explored. Second, to support the notion of "Global Priority", a family of distributed priority scheduling algorithms is developed, and its performance is profiled.

4.2 The notion of global priority

The notion of priority implies a total ordering of jobs. In the most restricted sense, the objective of priority scheduling is to ensure that this total ordering is followed. For a centralized queue, this objective is easily met through a queueing mechanism in which jobs are serialized according to their priority. However, in a multiple-server-multiple-queue environment, jobs are separately enqueued and served by their respective servers. Although such an environment allows a partial ordering of jobs to be specified in individual queues, the objective of total ordering is not achievable.

Nonetheless, a notion of *global priority* is often necessary for distributed systems. Many distributed operating systems, such as Locus [Walker83], Charlotte [Artsy85] and Demos/MP [Powell84], promote a view of the system as a virtual single server. In such systems, the level of service that a job receives should be transparent to its location. That is, a job should be allowed to join any computer and enjoy service compatible to its priority. This notion of having compatible priority service in all computers is defined to be *global priority.*

Global priority supports a total ordering of all jobs in the system. This objective does not differ from the objective of priority in a single-server queue. The question is, how can priority disciplines for centralized systems be extended to become *global priority* disciplines for distributed systems?

A multi-server environment has two major attributes not found in single-server environments. First, several jobs can be served simultaneously. Second, in addition to the cost of preemption, cost is incurred through job transfers between servers. Because of these attributes, we cannot directly use priority disciplines such as Head-of-line (HOL) and Preemptive-Resume (PR). Instead, HOL and PR must be extended to permit multiple servers. These disciplines are defined as follows:

Global Preemptive Resume

A distributed system with **m** *computers must serve the* **m** *highest priority jobs in the system. A new job can preempt a job in service if the priority of the new job is higher than that of the running job.*

Global Head of Line

When a job completes, a distributed systems must select the highest priority job in the system to be the next to serve. Once a job is started it is allowed to continue until completion even if jobs with higher priority arrive.

Since a multi-server needs to know both *when* and *where* to run a job, the above definitions are insufficient to specify fully how to schedule priority jobs. These definitions only specify *when* to run jobs. Moreover, these definitions do not indicate what cost and efficiency compromises are permitted if the overhead of scheduling is significant. In fact, these definitions can only serve as objectives. There is a large gap between these objectives and their implementations. This property contrasts with centralized priority disciplines, such as HOL and PR, which can be directly translated into scheduling algorithms in a single queue environment.

For simplicity, we shall assume for the remainder of this chapter that there only 2 priority levels. Using a simple analysis method based on the m*M/M/1 dual priority model, the next section demonstrates the difference between global priority scheduling and pure load sharing algorithms.

4.3 An m*M/M/1 dual priority model

An M/M/1 queue models an open, single-server system with a Poisson arrival process and exponentially distributed service times. Hence, an m*M/M/1 model can be employed to represent an **m**-node multi-computer system, where the computers are loosely coupled and do not exchange loads. Furthermore, with dual priority, jobs have either high or low priority, and are scheduled according to the preemptive resume discipline in each computer.

Using the m*M/M/1 dual priority model, we evaluate the probability that the system is in the *wait-while-available* state (P_{wa}). In this state, there is at least one job waiting while there is at least one computer available for that job. Since the job can have either low or high priority, this state consists of two mutually exclusive states: first, the state in which there is at least one job waiting while there is at least one computer idle, second, the state in which there is at least one high priority job waiting while there is no computer idle and at least one computer running low priority jobs. Livny [Livny82] referred to the first of these two states as *wait-while-idle*, and derived its probability (P_{wi}). We define the probability of being in the second state as P_r: the probability of invoking *remote preemption*. Hence:

$$P_{wa} = P_{wi} + P_r$$

Let **m** represent the number of computers, ρ_L represent the load of low priority jobs, and ρ_H represent the load of high priority jobs. If ρ represents the total load, we have:

$$\rho = \rho_L + \rho_H$$

Using Livny's result [Livny84],

$$P_{wi} = \sum_{i=1}^{i=m-1} \binom{m}{i} (1-\rho)^i \rho^{m-i}(1-(1-\rho)^{m-i}) = 1 - \rho^m + \rho^m(1-\rho)^m - 1-\rho^2)^m \qquad (4.1)$$

And, the probability of remote preemption is:

$$P_r = \sum_{i=1}^{i=m-1} \binom{m}{i} A_i\, B_{m-i} = \rho^m - (\rho-\rho_H^2)^m - \rho_H^m(1-(1-\rho_H)^m) \qquad (4.2)$$

where A_i is the probability of having at least *i* computers running low priority jobs, and B_{m-i} is the probability that there is at least one high priority job waiting in *m-i* computers:

$$A_i = \rho_L^i$$

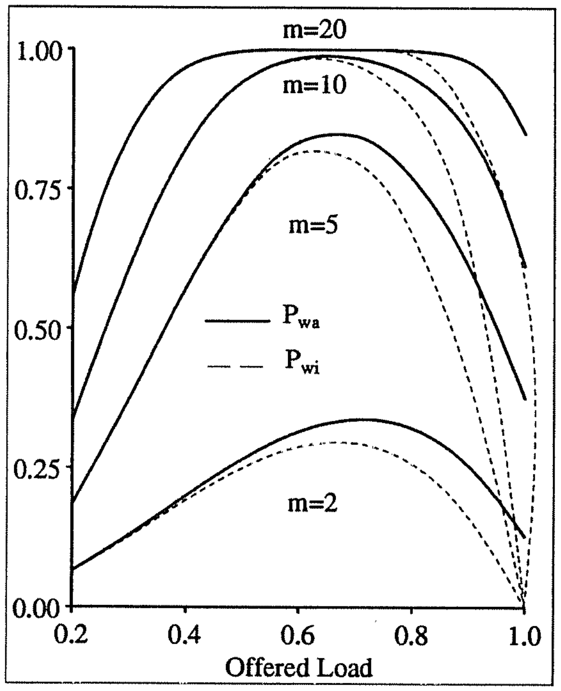

Figure 4.1 : Probabilities of wait-while-available and wait-while-idle (30% high priority jobs).

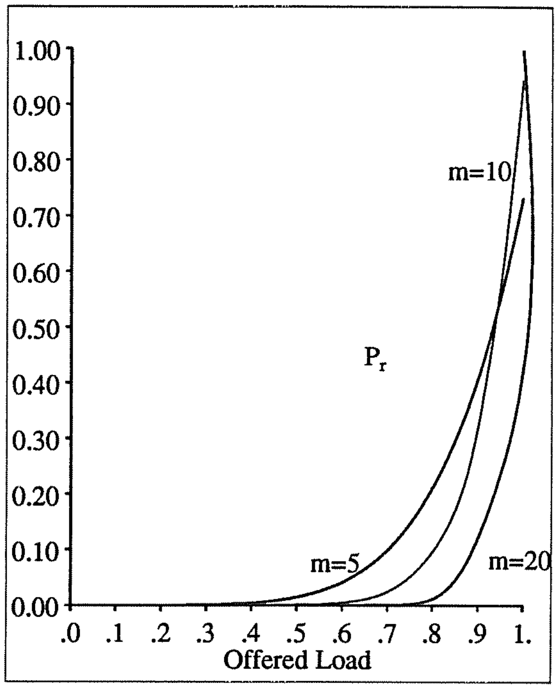

Figure 4.2: Probability of remote preemption (50% high priority jobs).

$$B_{m-i} = \rho_H^{m-i} - ((1-\rho_H)\rho_H)^{m-i}$$

Figure 4.1 plots P_{wa} and P_{wi} against load, assuming that the percentage of high priority jobs is 30%. It is worth noting that while P_{wi} drops to zero at 100% load, P_{wa} does not. This non-zero P_{wa} at high loads implies that although no free capacity is available, load transferring should still be activated to meet the objective of global priority.

Moreover, the probability of "wait-while-available" (P_{wa}), is always larger than the probability of "wait-while-idle" (P_{wi}), because the probability of remote preemption (P_r) is always positive (see equation (4.2)). However, the magnitude of P_r depends on three system factors: offered load (ρ), the percentage of high priority jobs, and the number of machines participating in the system.

Figure 4.2 shows changes in P_r under varying load. Curves are displayed for systems having 2, 5, 10 and 20 machines. These curves monotonically increase with increase in offered load. Hence, the maximum is located at the highest load, indicating that remote preemption is more likely to be needed at high loads than at low loads. Moreover, the curves have knees, after which the probability of remote preemption increases sharply. With larger numbers of machines, these knees occur at higher load. This trend is not surprising since load sharing is increasingly effective for larger systems, reducing the need for remote preemption.

It is intuitive that the need for remote preemption diminishes when the percentage of workload for high priority jobs is near either extreme, 0% or 100%. Figure 4.3 explores changes in the probability P_r at intermediate percentages of high priority jobs, with the offered load fixed at 70% of saturation. Each curve is found to have a maximum transfer probability at approximately the point at which 70% jobs have high priority.

Although a larger multicomputer might be thought to have greater need for remote preemption, this notion is dispelled by Figure 4.4, which shows the probability of remote preemption under variation in the number of machines in a system. It is surprising that the curves reach their maxima at relatively small

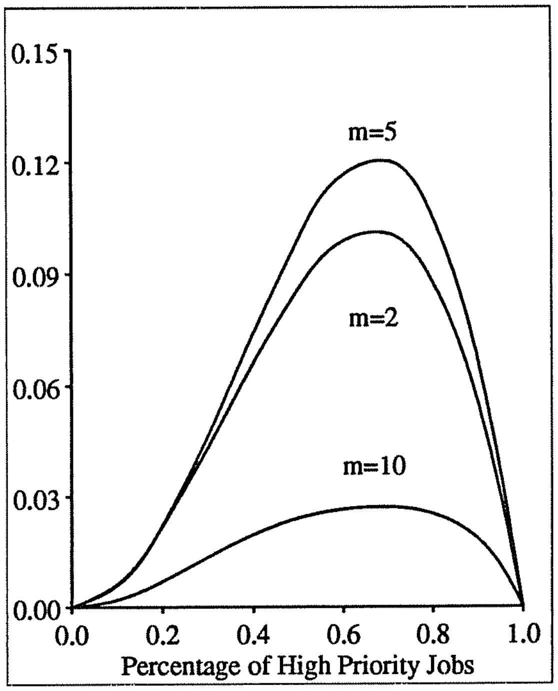

Figure 4.3: Probability of remote preemption under varying percentage of high priority workloads (ρ=0.7).

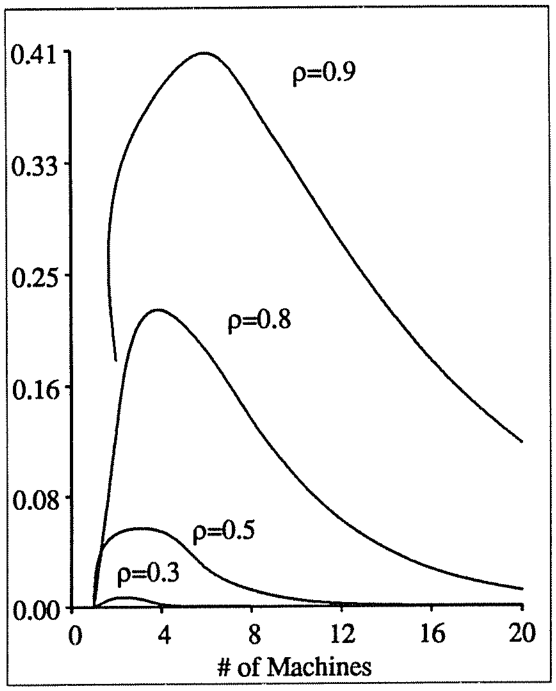

Figure 4.4: Probability of remote preemption vs. # of machines (50% high priority jobs)

system sizes— less than 15. Beyond this peak, the curves decrease exponentially. The magnitude of the peak increases with increasing load.

In summary, the difference between pure load sharing and global priority is significant when

- the load is high (60% or higher),
- the number of machines is small (less than 15),
- high priority jobs represent 30%-80% of the workload.

4.4 Design considerations

We assume that **m** computers are interconnected by a broadcast communication network. Each computer is subjected to an independent stream of arriving jobs whose priorities are either high or low. A job can be transferred among the computers and is executable on any computer. In addition, message passing is used as the primary means of transferring information.

Achieving the goal of global priority scheduling requires that priority jobs be redistributed among computers so that each job enjoys a level of service compatible with its priority. To support this load redistribution, a distributed resource management system must be employed. Such a management system consists of four components: information gathering, job placement, job migration, and local scheduling. Therefore, designing global priority scheduling algorithms within a framework of distributed resource management systems becomes a job of considering the trade-offs between these components.

The most important considerations of distributed resource management are the cost and the benefit of operations. For example, gathering status information entails both processor overhead and communication network delay. More frequent invocation of this operation allows the resource manager to acquire more up-to-date information. However, at some point the overhead outweighs the benefit of more current information. This effect is similar to that observed by Bryant and Finkel [Bryant81], who described a thrashing situation in which the system spends more time migrating jobs than executing jobs. Excessive management activities must be avoided.

In addition to guarding against excessive management activities, the management algorithm must avoid starving low priority jobs. Such starvation could occur when system capacity is not sufficient for both scheduling overhead and job execution. Starvation is more likely to occur in distributed systems than in single-processor systems, because scheduling overhead is greater and high priority jobs are scheduled at the expense of low priority jobs.

Centralized scheduling and distributed scheduling are two approaches to meet the objective of global priority. Under a centralized approach, all system state information is stored at a central site and job placement and migration decisions are made serially according to job priority. With such centralized control, the scheduling algorithm can best ensure that *the N highest priority jobs in an N-computer system are served at the earliest feasible moment.* This assurance does not preclude the possibility that a high priority job is being transferred while a low priority job is running. However, a centralized scheduling algorithm may initiate excessive management activities to meet its design goal. In addition, much overhead may be incurred by maintaining up-to-date information about the states of jobs and computers. Therefore, the overall performance of a centralized scheduling algorithm is not necessarily good.

Under a distributed scheduling approach, each computer maintains partial knowledge about system status, and each computer negotiates with other computers to make job placement/migration decisions. Since such an approach does not permit complete knowledge of all jobs in the system, it is impossible to guarantee that, at any moment, the N highest priority jobs in an N-computer system have been served as early as possible. However, the overhead of job scheduling may be significantly less than that of a central scheduling approach, because of the relaxed attitude toward preserving relative priority. To study this tradeoff, we propose one centralized and three distributed scheduling algorithms, each represents a different scheduling strategy.

4.5 Centralized scheduling approach

Under centralized scheduling, a central agent maintains a queue of job descriptors ordered by job priority. When two jobs have the same priority, the tie is broken according to the First-Come-First-Serve (FCFS) discipline. A job descriptor contains information regarding the priority, location, arrival time and statistics of the job. The central agent is notified by each computer whenever a job arrives or terminates. A new job, before beginning execution, must be registered at the central agent.

The central agent has complete control over local computers; before a local computer can start, preempt or transfer a job, the central agent must request it. The central agent commands local computers to run the top N jobs in the queue disregarding jobs in transfer. When remote preemption is necessary, the central agent sends a job transfer request to the source of the high priority job and asks the destination computer to prepare to accept a job that will preempt its currently running job. A detailed description of the algorithm is as follows:

Global Priority Scheduling (GPR) algorithm

E1. At the time of job arrival:
    The central agent enqueues the description of the new job in the common queue. If the job is within the top N positions of the queue, the central agent must allocate resources for the job immediately. To do so, it may be necessary to transfer the job to an available computer. During this allocation process, a low priority job may be preempted if its job descriptor falls out of the top N positions of the common queue. A preempted job stays where it is.

E2. At the time of job completion:
    The newly available computer is allocated to the next job in the common queue. If the next job is not

already located on the available computer, it is transferred over.

4.6 Distributed scheduling approach

Under the distributed scheduling approach, computers must negotiate with each other to meet a common scheduling objective. Although this approach suffers from a lack of complete system knowledge, it has the advantage of being robust (avoiding a single point of failure), and fast (allowing scheduling activities to occur in parallel).

Three distributed scheduling algorithms are considered in this study: symmetric polling (**SMP**), priority riding (**PRR**) and status broadcasting (**STB**). These algorithms all employ the same local preemptive-resume scheduling discipline, but differ in their information gathering, job placement and job migration strategies. The SMP algorithm uses priority information in negotiating job placements and migrations. Under this algorithm, load sharing is prioritized and maintained using a symmetrical polling technique. Extending one step farther, the PRR algorithm uses priority in the mechanism of job transferring. A low priority job is not permitted to be transferred if a high priority job is being served at either the source or destination. However, arrival of local low priority jobs is still allowed. Finally, using no priority information in the exchange of loads, the STB algorithm gives little support to the notion of global priority.

4.6.1 Symmetric polling algorithm (SMP)

Under this approach, a scheduler polls a region of neighboring computers for load information. The polling is symmetric because both the sender and the receiver of a transferred job can initiate the polling. The computers contained in this region are chosen randomly, though the maximum region size is fixed.

The goal of the SMP algorithm is to give *high priority jobs preemptive precedence over all low priority jobs in the polling region.* A high priority job is allowed to remotely preempt any low priority job in the region. The size of the region serves as a system parameter to control management activity.

The SMP algorithm is centered on the notion of an *available* computer. A computer Y is available to job X if it is a computer that is currently running a job with a priority lower than X. That is, job X can immediately start (or resume) service when it is transferred to computer Y. Therefore, both an idle computer and a computer running a low priority job are available to a high priority job. The negotiation protocol of the SMP algorithm is as follows:

E1. At the time of job arrival:

(1) Check if the local computer is available. If not, the scheduler polls computers in the region until either an idle computer is found or the polling limit is reached. If no idle computer is found, an available computer is selected for the new job.

(2) When a remote computer is selected to be the host of the new job, an "allocation request" is sent to the new host to negotiate a job transfer. This action may result in the preemption of the currently running job at the destination computer. The destination computer may choose to start another round of polling to find an available computer for the preempted job, which may cause a chain reaction until either an idle computer is found or the preempted job has such low priority that it cannot be relocated. (Load information collected by the first computer may be transmitted to the job receiver to facilitate its search.)

(3) If no computer is available in the region, the new job is placed in the local queue.

E2. At the time of job completion:

(4) If the next job in the queue has the absolutely highest priority of the system, it is immediately scheduled to run.

(5) Otherwise, the local scheduler polls computers in the region to invite a job that can take precedence. If polling fails to locate a high priority job and there is a local low priority job, the job is started. If polling fails and the local computer is idle, it is left idle.

4.6.2 Priority riding algorithm (PRR)

The PRR algorithm extends the SMP algorithm to provide high priority jobs with better performance at the expense of low priority jobs. The PRR algorithm is similar to SMP except in one important point: *the transfer activity is assigned the priority of the job it transfers.* In contrast, job transfer activity has the highest priority under the SMP algorithm. Therefore, no low priority jobs are allowed to be transferred in or out when the computer is doing the transfer or execution of a high priority job. Since the priority of the task transferring activity is ridding on the job transferring activity, this algorithm is named "priority riding".

4.6.3 Status broadcasting algorithm (STB)

The STB algorithm is a simple load sharing algorithm fashioned after the broadcast status algorithm [Livny82], in which every status change, new arrival or termination, is broadcast to a dynamic region of computers.

To reduce overhead, this algorithm is modified so that computers broadcast their current state only on becoming idle or when a new job arrives while the computer is serving a job. Using this state information, both the heavily loaded and lightly loaded computers are allowed to initiate negotiation for a load transfer. The goal is to transfer jobs from computers that are heavily loaded to those that are lightly loaded. The job chosen to migrate is at the end of the local queue.

Using no priority information in negotiating a job transfer, the STB algorithm aims primarily at reducing occurrences of the wait-while-idle state. However, owing to the local PR scheduling algorithm, the objective of global priority is indirectly supported. Surprisingly, the performance resulting from this minimal support of global priority compares favorably under some conditions with that resulting from algorithms giving comprehensive support.

4.7 The simulation model

To evaluate and compare the proposed algorithms, we have selected the m*M/M/1 queuing model [Livny82] as the base of this performance study. This model consists of **m** homogeneous M/M/1 queuing systems interconnected by a broadcast communication subnet. Each queue is subject to an independent stream of arriving jobs, each of which may have either high or low priority.

It is assumed that a job can be served by any server and that the cost of job placement and job migration are the same. This assumption is made for jobs that do not request dynamic memory and do not communicate with others. Our assumption is good for many real-time systems, because real-time jobs are less likely to grow significantly in size and often have only limited communication in the beginning and the end of the computation. The processing cost of a job transfer is chosen to be 5%-30% of the average execution time of a job. This cost is chosen to be large enough to demonstrate the accumulated overhead of scheduling activities.

Owing to the priority-based dynamic scheduling of jobs, there is no feasible analytical solution for the above performance model. Therefore, discrete event simulation is used to evaluate the performance of the proposed algorithms. The **DENET** simulation package [Livny87], which is a Modula-2 based simulation language, is used as our implementation language.

In our simulation, we represent the job transfer cost both as a *processor cost* and a network delay. The results of measuring the V-kernel [Cheriton83] show that the processor costs of packaging data for transmission and unpackaging it upon reception far outweigh the network costs of transmitting the data. Job transfer cost is modeled as a fixed overhead at the source and the destination computers. The simulation parameters are shown in Table 4.1.

Parameter Name	Distribution	Value(s)
Computers	Constant	6-20
Job service time	Exponential	mean = 1.0 unit
Job interarrival rate	Exponential	0.1-0.9
Status exchange delay		negligible
Job transfer delay	Constant	0.05 - 0.30 unit
Fraction of high priority jobs	Constant	30%

Table 4.1: Simulation Parameters

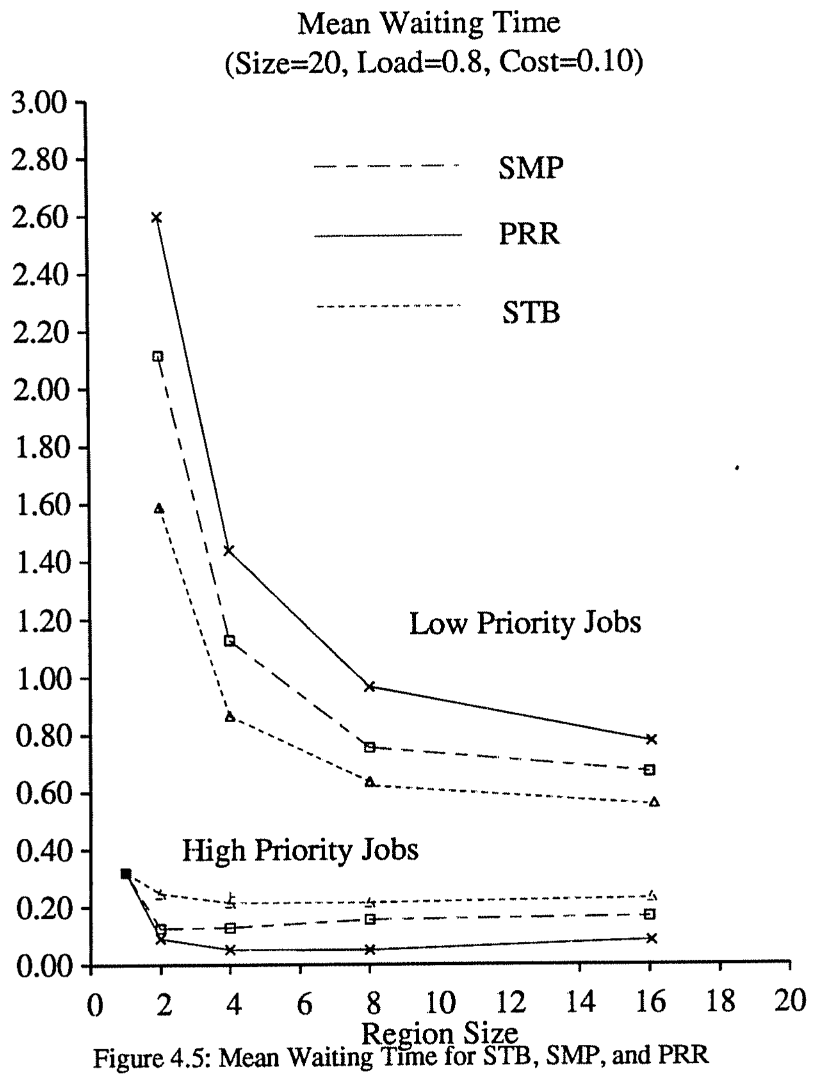

Figure 4.5: Mean Waiting Time for STB, SMP, and PRR

4.8 Performance issues

4.8.1 Major comparison

Figure 4.5 shows the results of the SMP, PRR and STB algorithms on a system of 20 computers, with an offered load of 80% saturation and 30% high-priority jobs. Performance is measured under varying region sizes, where a region is the polling limit for SMP and PRR and the area of broadcasting for STB. The region size controls the level of management activity in distributed scheduling. Several important observations can be made about Figure 4.5:

- For high priority jobs, PRR > SMP > STB, and for low priority jobs, PRR < SMP < STB.
- The SMP and PRR algorithms, which are global priority algorithms, outperformed the STB algorithm, a pure load sharing algorithm. The SMP an PRR algorithm improve the performance of high priority jobs, and widen the gap between low and high priority jobs.
- Low priority jobs benefits most from an increase in region size.
- High priority jobs receive best service with small regions.

4.8.2 Performance metrics — total weighted cost

Priority is a means to an end. In general, priority scheduling is adopted by a system to meet some performance objective, such as minimizing the cost. However, general purpose systems often do not have well defined performance objectives. Therefore, priority assignment of jobs is treated as a hint by the operating system. For example, VM/CMS[2] and Unix [3] allow job priority to be assigned at job starting time, but give no guarantee for their response time. The system documentation merely states that high priority jobs will receive preferential service.

In contrast, soft real-time systems use priority to indicate the urgency of a job when the time constraints in a real-time system cannot be met by all jobs. We use weighted cost as the evaluation criterion for priority scheduling. The cost difference between low and high priority jobs can be set to reflect the criticality of jobs. Let λ_i to be the arrival rate of priority class *i*, and t_i to be the average completion time of that class. If c_i is the cost of class *i*, then the total weighted cost *h* is defined as

$$h = \sum_{i=1}^{i=k} c_i \lambda_i t_i$$

Kleinrock [Kleinrock76] proved that the optimal scheduling algorithm to minimize total weighted cost for a single-queue-single-server system is the Preemptive-Resume priority discipline, under the condition that jobs with higher costs are assigned higher priority. Furthermore, the optimality of the Preemptive Resume priority scheduling on total weighted cost is irrelevant to the distribution of costs, as long as Kleinrock's conservation law holds.

However, for two reasons, the conservation law is not valid for distributed systems. First, the overhead of scheduling, which includes message passing and task transfer overhead, is not negligible. Hence, capacity is lost during the scheduling activity. Second, even with the perfect load sharing policy, capacity is not fully utilized due to the delay in transferring a job from one computer to another. Therefore, for different distributed scheduling algorithms, the total completion time of jobs is not conserved.

To simplify evaluation in a dual priority system, *cost ratio (CR)* is defined as the ratio of costs between high and low priority jobs. The total cost in this dual priority system is then:

$$h = (cr * t_1) + t_2$$

[2] VM/CMS is a trademark of IBM.

[3] Unix is a trademark of AT&T.

	Average response time		Total Cost	
Model	high	low	*CR*=100.	*CR*=1.1
M/M/1	1.32	14.02	146.02	14.47
M/M/2	1.06	6.86	112.86	8.02
M/M/4	1.01	3.48	104.5*	4.59*

Table 4.2: Average response time and the total cost under
the dual priority M/M/m model (Load=0.8, 30% high)
(* the lowest total cost)

Table 4.2 shows the performance results of preemptive resume scheduling in an M/M/m system using Buzen's derivations [Buzen83]. In addition, the value of total costs resulting from cost ratios 100 and 1.1 are displayed. The entry marked by a * sign has the lowest total cost under a cost ratio. These two cost ratios represent two drastically different combinations of costs. One favors high priority jobs and the other treats low and high priority jobs almost equally. Using a performance metric of total cost, M/M/4 is the winner regardless of which cost ratio is being used.

	CR=100		CR=1.1	
region size	2	8	2	8
SMP	11.7*	5.97*	2.71	1.02
PRR	14.7	16.3	2.26	0.93
STB	22.3	22.1	1.1*	0.87*

Table 4.3: Total weighted cost under
the dual priority 20 machine system (Load=0.8, 30% high priority jobs)
(* lowest cost under a cost ratio)

Table 4.3 shows the total weighted cost for algorithms PRR, SMP, STB with 20 machines, and transfer cost 0.10. In this table, the PRR algorithm has the lowest total cost when the cost ratio is 100, while the STB algorithm has the lowest total cost when the cost ratio is 1.1.

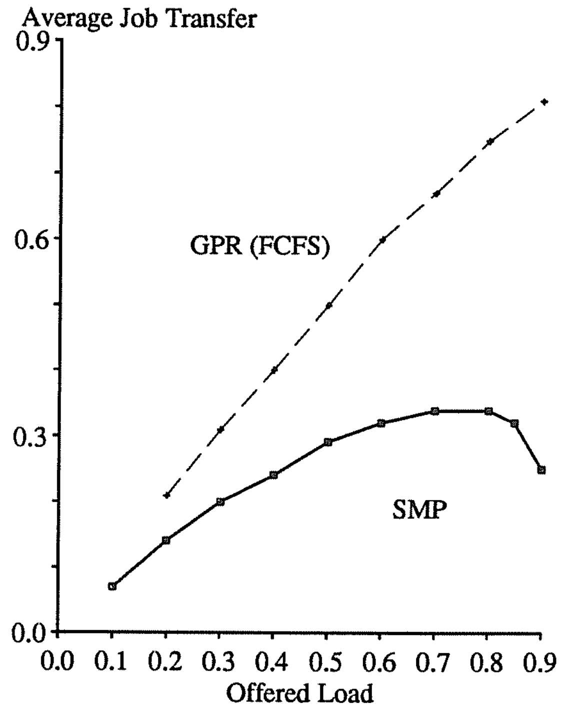

Figure 4.6: Mean Job Transfer for GPR and SMP

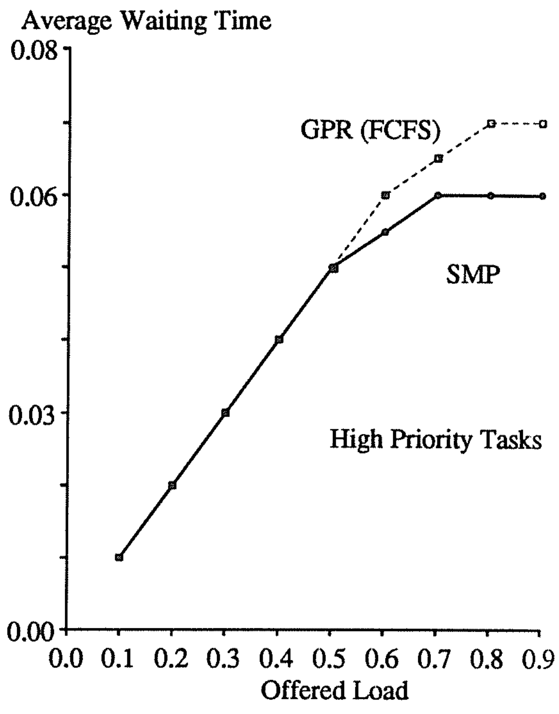

Figure 4.7: Mean Waiting Time for GPR and SMP (High Priority)

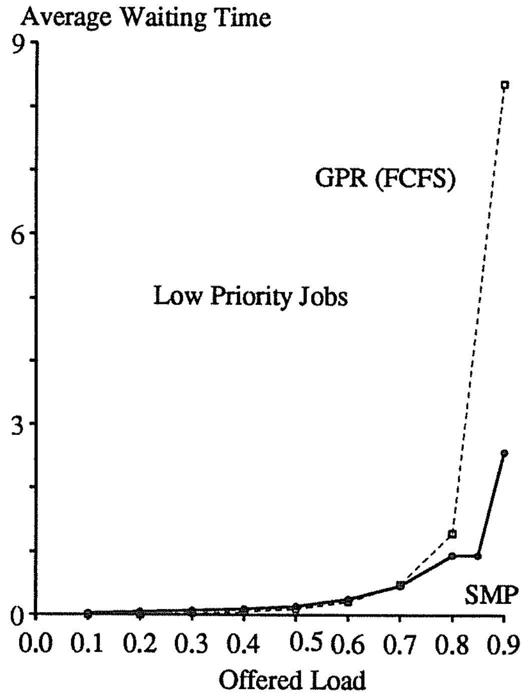

Figure 4.8: Mean Waiting Time for GPR and response time for SMP (Low Priority)

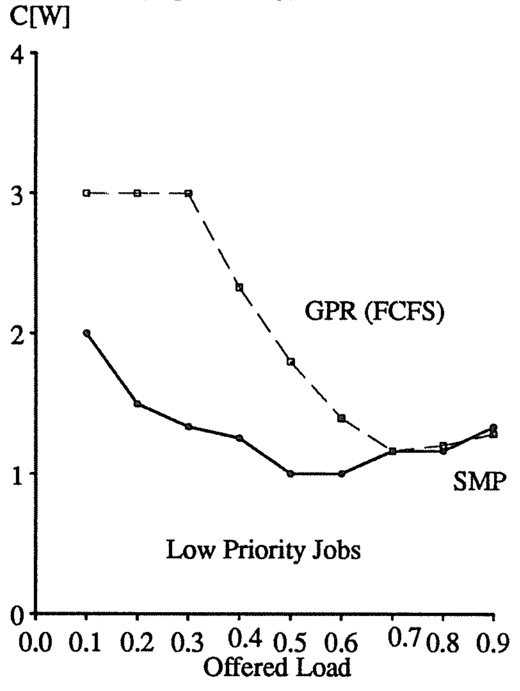

Figure 4.9: Coefficient of variance of GPR and SMP (High Priority)

4.8.3 GPR (FCFS) vs. SMP

The GPR algorithm is outperformed by the SMP algorithm in every aspect of performance. Figures 4.7 and 4.8 show the average waiting time, and Figure 4.6 shows the average number of job transfers per job for both algorithms. The SMP algorithm results in superior performance for both high priority and low priority jobs, and consumes less capacity for scheduling than GPR. At low to medium loads, these algorithms provide nearly equal performance. However, GPR results in a substantial increase in waiting time when the offered load increases to 90%, while SMP results in only a moderate rise.

The GPR algorithm invokes more job transfers than SMP over the entire range of loads. The behavior of the GPR algorithm at high loads can be attributed to the fact that it maintains a central queue with FCFS ordering. However, owing to FCFS requirement, a high priority job may be transferred to a remote computer in which another high priority job exists. These are unnecessary transfers in the sense that it does not help reduce the average response time of priority jobs. Moreover, it increases the overhead of the system. Thus, the mean number of job transfers grows sharply as the offered load increases.

Under the SMP algorithm, priority precedence is maintained without the global FCFS rule. The SMP algorithm gives preference to the local job whenever two jobs have the same priority. When high priority jobs are waiting at both the local and remote computers, the local job is selected first. In contrast, the GPR algorithm may choose the remote job if it had arrived earlier than the local one.

Figure 4.9 shows the coefficient of variation of waiting time for the high priority jobs. Although one of the motivations for using FCFS as a secondary priority index is to reduce deviation in waiting time, GPR has a larger coefficient of variation in response time than SMP.

In a unicomputer environment, FCFS ordering is commonly used because of ease of implementation and its fairness. In M/G/1 systems, FCFS has lower variance in job response time than other demand-independent non-preemptive queuing policies, such as Service-in-Random-Order and Last-Come-First-Served [Lavenberg83]. In contrast, global FCFS ordering results in larger variance than other policies in a distributed systems.

4.8.4 The sensitivity to region size

Region size is an important factor deciding the activity level of distributed priority scheduling algorithms. Increasing the region size increases the activity, and decreasing the region size slows down the activity. For distributed priority scheduling algorithms, the region size not only decides the level of load sharing activity, but also changes the relative performance of low and high priority jobs.

The level of load sharing activity

Since the average number of transfers per job reflects the activity level of load transfer, the difference in the needs of load transfer between both low and high priority jobs can be inferred from the number of transfers per job for two priority classes. Figure 4.10 shows how the average number of transfers per job increases as the region size increases. This figure indicates that while the number of transfers per high priority job levels off at region size 4, the number of transfers per low priority job increases until size 20 (the largest region size measured). Two reasons seem to adequately explain this higher and increasing activity level of low priority jobs. First is the population factor, because a high proportion (70%) of jobs have low priority; second is the pushing effect of global priority scheduling, in which a low priority job may be preempted more than once by high priority jobs, hence low priority jobs are pushed to transfer more times to receive service. For these reasons, low priority jobs log high transfer activities in large region sizes.

Low priority jobs

Figure 4.11 displays the average waiting time of low priority jobs at load 0.8 against region size. The waiting time decreases as region size increases. However, the rate of change decreases to almost zero at region size 15. The decrease indicates a diminishing point of return for the benefit of distributed

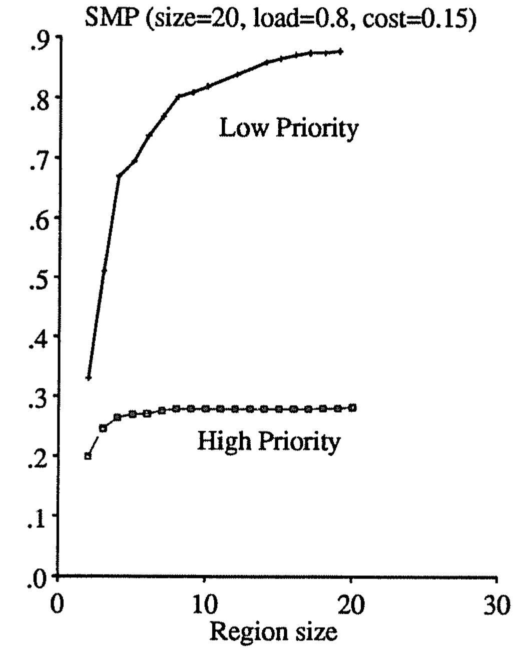

Figure 4.10: Average Job Transfers per Job for SMP

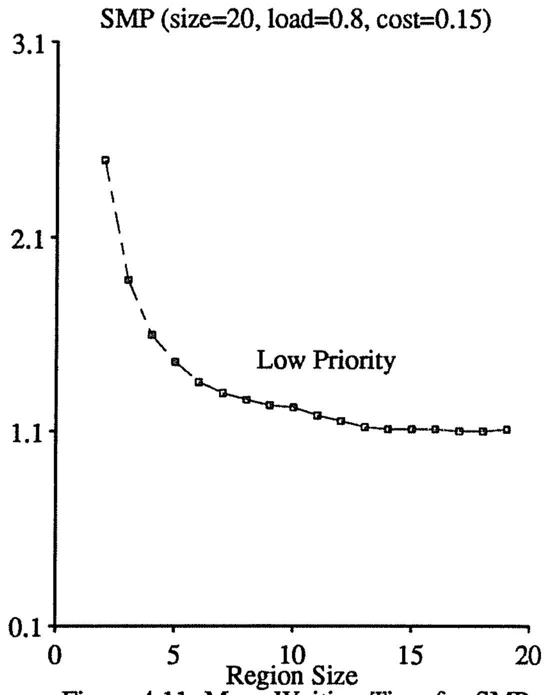

Figure 4.11: Mean Waiting Time for SMP (Low Priority)

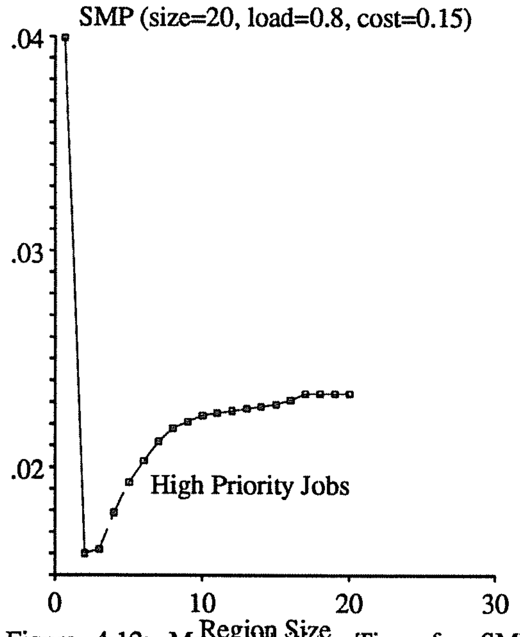

Figure 4.12: Mean Waiting Time for SMP (High Priority)

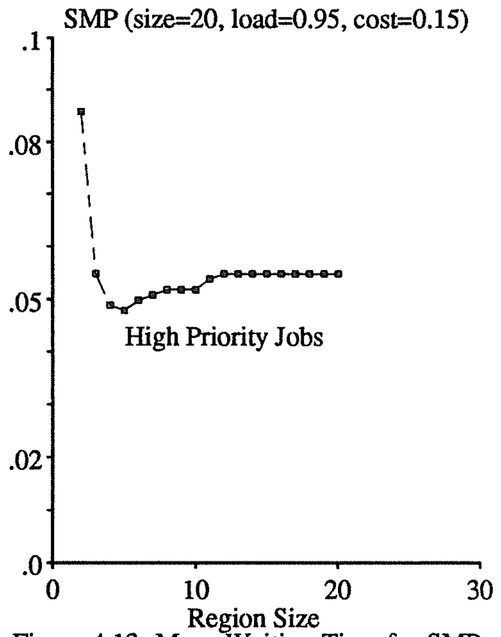

Figure 4.13: Mean Waiting Time for SMP (High Priority)

scheduling at large region sizes.

High priority jobs

In Figures 4.12-4.13, the waiting time of high priority jobs is shown to first decreases and then increases as region size increases. In other words, high priority jobs have worse performance at large region sizes than that at small region sizes. This phenomenon of bag-shaped curve is remarkable. This worsening of waiting time for high priority jobs in large regions is due to the load sharing activities of low priority jobs, with which the scheduling cost is paid at the expense of high priority jobs.

4.8.5 The amount of remote preemption

Owing to need for remote preemption, we predicted that distributed priority scheduling exhibits a higher level of load sharing activities than a pure load sharing algorithm does (see section 4.3). Figure 4.14 plots a comparison between the average number of job transfers per job with and without priority. The system with priority jobs has a higher average of job transfers at all offered loads, with a maximum in the difference located at load 70%.

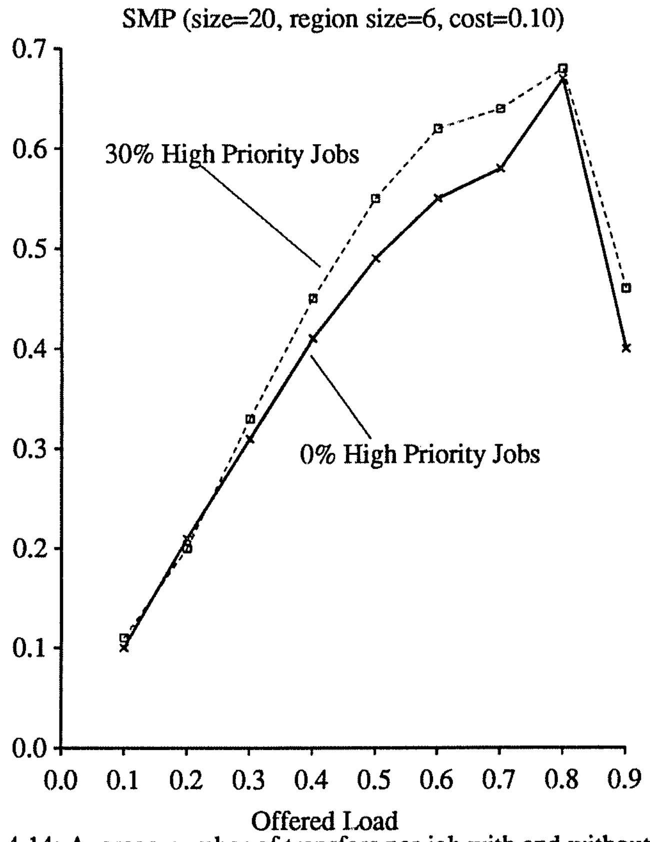

Figure 4.14: Average number of transfers per job with and without priority

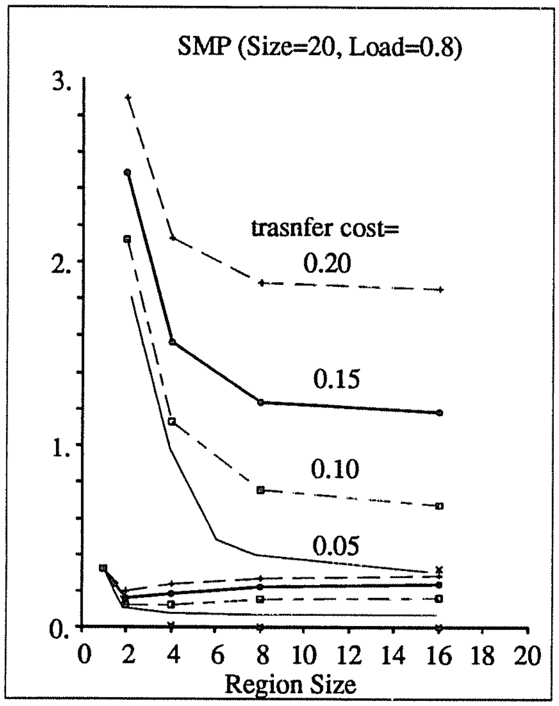

Figure 4.15: Mean Waiting Time under varying Transfer Cost (SMP)

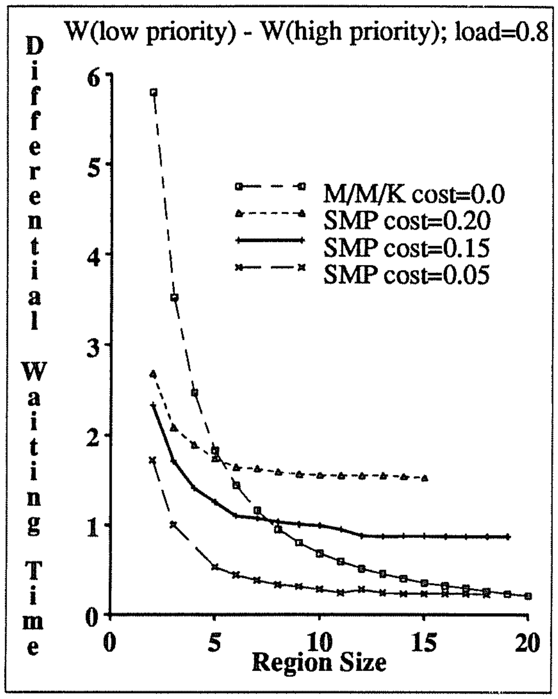

Figure 4.16: Waiting Time Difference under varying Transfer Cost (SMP)

4.8.6 The sensitivity to the cost of scheduling

The overheads of a priority resource management system can be divided into two groups — the overhead of status exchange and the overhead of job transfer. Both result from communication among computers. The sensitivity of a distributed scheduling algorithm to these overheads may well determine its cost-effectiveness in providing service to jobs under various communication overheads.

When the cost of communication increases, there are reasons to believe that the performance of low priority jobs would deteriorate more than high priority jobs. Low priority jobs are often transferred more than once, because of the possibility of being preempted. In contrast, a high priority job is transferred at most once. Therefore, when the cost of job transfer increases, the frequent migrators, low priority jobs, lose more. Besides, owing to the precedence of high priority jobs, low priority jobs also bear the cost of moving high priority jobs.

Figure 4.15 shows that the mean waiting time for jobs of both priority classes under varying transfer cost and region size. Although cost is increased, the pattern of variation in waiting time under varying region size remains the same: increased region size decreases job waiting time. However, when cost increases, low priority jobs degrade more than high priority jobs. Therefore, under high scheduling cost, while the benefit for high priority jobs remains, the gains of scheduling low priority jobs across machines diminishes.

The difference in mean waiting time between low and high priority jobs under varying transfer cost and region size is displayed in Figure 4.16. This figure displays curves with transfer costs ranging from zero overhead to an overhead of 20% of mean execution time. Since the curve with zero overhead is obtained from M/M/K analytical model, it is not the "limit" of other curves. Instead, it shows the difference of response time between low and high priority jobs under perfect global priority scheduling. Therefore, the figure demonstrates that the waiting time difference of the SMP algorithm is close to ideal value at region size 20 under cost 0.05.

4.9 Global priority levels

Although we focus on systems with two global priority levels, our algorithms can be easily generalized to more than two global priority levels. Since the need of remote preemption increases as the number of priority levels increases, the system may become thrashing if a high number of priority levels is provided. Therefore, it is better to limit the number of global priority levels, even though the local priority of jobs can be a real number (allowing infinite number of priority levels), requiring a mapping from local priority to global priority level.

For systems with more than two global priority levels, the relative performance of highest and lowest priority jobs is expected to be the same between SMP, PRR and STB algorithms. However, for jobs with intermediate priority levels more study is needed. Because of the expected increasing scheduling activity, the capability of PRR to prevent high priority jobs from being interrupted becomes very important.

4.10 Summary

The notion of global priority for distributed systems generalizes the concept of priority in a single queue system. Under global priority scheduling, a job receives a compatible level of priority service at any computer in the system. In this chapter, tour distributed priority scheduling algorithms that aim at supporting the goal of global priority, were proposed and examined. The GPR (FCFS) algorithm uses a centralized global priority scheduling in a decentralized system. By relaxing the strict FCFS requirement, the SMP algorithm switches to decentralized control using replicated schedulers. The PRR scheduling algorithm restricts the job transfer activity of low priority jobs when high priority jobs are being served. Finally, the STB scheduling algorithm takes a different approach combining local priority scheduling with a blind load sharing policy — status broadcasting.

In order to evaluate the performance of the proposed scheduling algorithms, a simulation framework of a dual priority m*M/M/1 model was constructed. Extensive simulation results show that global priority scheduling has achieved a significant performance improvement for high priority jobs, compared with a pure load sharing algorithm that does not include priority in its global resource management mechanism. To summarize:

(1) The **Global priority scheduling** algorithm is more expensive than other algorithms, owing to the overhead of maintaining both the priority and First-Come-First-Serve ordering. Performance is found to be sensitive to offered load with high loads leading to drastically worse average performance. Nonetheless, the absolute global ordering guarantees high priority jobs preferential treatment.

(2) The **Symmetric Polling** algorithm generates a significantly smaller number of job transfers than the GPR algorithm. Our simulation results show that by not worrying about "global" First-Come-First-Serve ordering, the number of job transfers decreases to an acceptable level. This results in an overall decrease in system overhead, which in turn leads to better response time for all jobs, including high priority jobs, and smaller variance in job waiting time within a job class.

(3) The **Priority Riding** algorithm is designed to correct a phenomenon of SMP, in which low priority jobs benefit from load sharing at the expense of high priority jobs. Therefore, the waiting time of high priority jobs decreases as the load transfer activity for low priority jobs increases. This phenomenon violates the purpose of priority scheduling, which is to provide preferential treatment for high priority jobs. The PRR algorithm corrects the phenomenon by associating the transfer priority with the job it transfers. This correction prevents high priority job from being interrupted by low-priority job transfers.

(4) The **Status Broadcasting** algorithm is a pure load sharing algorithm that does not use any global priority information. This algorithm provides a base to assess the effect of introducing global priority into distributed scheduling. Surprisingly, this algorithm performs well in supporting priority jobs across machines in terms of their waiting time. Nonetheless, a close examination shows that the gap of waiting time for low and high priority jobs is smaller for STB than that of other global priority scheduling algorithms.

We showed that the optimality of global priority scheduling algorithm depends on the cost distribution of jobs under a performance metrics of weighted costs. Therefore, a system manager must examine the job cost distribution to select a suitable distributed priority scheduling algorithm.

This chapter and the previous chapter conclude the study on scheduling sequential computations for distributed soft real-time systems. The various proposed algorithms serve to illustrate the point that distributed scheduling must be amended for distributed real-time systems. In contrast to the popular belief that load distribution should be limited to the recovery of important functions upon machine crashes, we show that it can provide real-time systems with services on a regular basis.

The next chapter opens up a new dimension for job scheduling, the scheduling of distributed computations. Owing to the complexity of such scheduling problem, the study will be limited to distributed computations having a pipeline structure.

Chapter 5
Priority Scheduling for Distributed Computations

In the previous two chapters, we extended priority scheduling algorithms from a traditional, central queue environment to a distributed environment. In addition, we explored the effect of applying the priority principle to sequential computations in such an environment. In this chapter, we explore resource allocation policies that support priority scheduling for distributed computations.

Priority is a means of expressing the urgency of clients waiting for service. A notion of *Composite Priority* is introduced to distinguish between *job priority* and *task priority*, since in distributed systems, a client can be either a single task or a job consisting of a group of cooperating tasks. A job priority represents the relative importance of a job compared with other jobs in the system, while a task priority represents the relative urgency of a task compared with other tasks in its job.

Composite Priority provides distributed computations with a means to specify the relative urgency at both the job level and the task level. This distinction between job and task is necessitated by the different roles they play in the system. Job priority is solely determined externally to meet system requirements, while task priority can be determined either internally or externally. For example, a task priority may be assigned internally by the scheduler to improve job response time. Based on an activity trace of a job, the scheduler may assign high priority to tasks on the critical path of the job execution graph. Alternatively, for external priority assignment, a user may give priority to tasks to improve the response time of a specific task.

Most of the previous work in managing resources for distributed computations has assumed a single computation with a dedicated computing environment. Previous work focused primarily on static task placement to minimize job response time and to reduce intermachine communication overhead [Gylys76] [Stone79] [Bokhari84] [William84]. For multi-computer systems with a dynamic workload, the resource allocation problem cannot be solved statically [Tilborg84]. Instead, a dynamic resource management approach is needed. Only a small number of studies addressing the issue of dynamically scheduling distributed computations have been reported. For a non-real-time distributed system, Tilborg [Tilborg84] proposed a wave scheduling technique that allows a network of computers to be dynamically partitioned to accommodate dynamically arriving distributed computations. His model assumes that each task has a processor dedicated to it, and thus deviates from the typical multiprogramming model of modern systems. Oursterhout [Oursterhout80] has studied a *coscheduling* mechanism which attempts to ensure that if one activity of a task force is executing, all runnable activities of the task force are also executing, each on a different processor. The coscheduling mechanism was shown to improve job response time owing to less number of context switchings. For a real-time distributed system, Cheng *et al.* [Cheng86] proposed a dynamic scheduling algorithm for a group of tasks with precedence constraints. This algorithm dynamically places and assigns tasks to processors so that the *hard* deadline of the entire group can be guaranteed. For a distributed database system, Carey and Lu [Carey86] have proposed an algorithm to dynamically distribute pipelined queries on multiple database sites.

The issue of providing composite priority services to distributed computations has not been previously addressed. The challenge of this study is to find resource management algorithms that support composite priority and pay special attention to the unique characteristics of a distributed computations, such as task synchronization pattern.

Our model of distributed resource management is discussed in Section 1. Section 2 presents a composite priority processor scheduling algorithm. The task placement algorithms that map a group of tasks to processors are examined in Section 3. Performance of these algorithms is evaluated through simulation, with results reported in Section 4. We summarize our results and discuss conclusions in Section 5.

5.1 Our Distributed Resource Management Model

In a distributed system with a dynamic workload, several resource allocation decisions must be made. These decisions include *how* to schedule tasks in a processor, *where* to place tasks, and *when* and *where* to dynamically migrate tasks. These resource management decisions are addressed by various policies. How tasks are scheduled in a processor is addressed by a processor scheduling policy, where tasks are to be placed is addressed by a task placement policy, and a dynamic migration policy addresses when and where to migrate tasks.

For this preliminary investigation, we make the following two simplifying assumptions regarding the distributed resource management system: First, the system does not allow dynamic task migration. Second, the system uses a centralized resource management system, for which the overhead of status reporting and scheduling are assumed to be negligible. Disallowing task migration reduces the complexity of resource management by restricting its function to *initial task placement* and *processor scheduling*. Using a central manager for task placement eliminates the need for negotiation among distributed resource managers and simplifies information gathering.

5.2 Processor Scheduling

5.2.1 A Composite Priority queueing discipline

Under composite priority scheduling, a task has two priority indices: a job priority index and a task priority index. One could imagine a simple way of combining these two priority indices for processor scheduling. When two tasks compete for the same resource, their job priority can be used as the primary criterion for comparison, with the task priority being used as the secondary criterion. Unfortunately, under this plan, a job may obtain unfair advantage for its tasks by malicious assignment of task priorities. For example, if two jobs with the same job priority assign different priorities to their tasks, the response time of the job with lower-priority tasks will become unacceptably long. To avoid this, we must clearly define how resources are divided among jobs having the same priority.

Since job priority specifies the ordering of jobs in the system, tasks belonging to a job should receive compatible service from the system. Based on this principle, task priority should be used solely to arbitrate between tasks belonging to the same job. Tasks having the same job priority but belonging to different jobs should be interleaved to give each job (group of tasks) a fair share of resource. At the same time, the partial ordering of tasks belonging to the same job based on task priority should be enforced.

To meet these requirements, we have devised the following processor scheduling scheme: Task priorities for tasks belonging to a job and residing on a processor are enumerated to be 0, 1, 2 ... (the lowest-priority tasks have priority 0). Then, tasks are scheduled at each processor by job priority and task priority, in that order. For example, Job A has four tasks residing on processor P1 and two tasks residing on processor P2. If task priorities are numbered from 20 to 25, they are renumbered to be from 0 to 3 on the processor P1 and from 0 to 1 on the processor P2. Job B has three tasks residing on processor P1 and one task on processor P2. If task priorities are numbered from 100 to 104, they are renumbered to start from 0 at each processor. This renumbering scheme allows tasks from different jobs to interleave, sharing resources equally.

5.2.2 Some Remarks on Task Level Priority

When tasks share processors, priority scheduling of tasks may improve the response time of a job. It has been shown for a multiprocessor environment that, if a job contains tasks with precedence constraints, the longest-critical-path-first scheduling algorithm significantly improves job response time relative to first-come-first-serve scheduling [Kohler75]. However, in a distributed environment with a dynamic workload, the significance of task priority scheduling is less certain. Since the discriminating power of task priority is limited to a single processor, many environmental factors, such as load distribution of the system and network contention, can undermine its effect. Moreover, the significance of task priority depends on the job's synchronization pattern.

We have carried out experiments on two types of distributed computations: *pipeline* and *master-slave.*

A pipeline job consists of a sequence of tasks, numbered 1...N, in which task i ($1 < i < N$) repeatedly receives a unit of work from task i-1, computes for some time and passes a unit of work to task i+1. Task 1 and N are similar, except that task 1 generates its own work, and task N does not pass work on. We assume each pair of adjacent tasks shares buffer space sufficient for 1 unit of work; task i may block waiting to receive work from task i-1 or send work to task i+1.

In our experiments, we set $N = 20$, and let the amount of computation per unit work be an exponentially distributed random variable with mean 20. The pipeline runs for 20 cycles — i.e., until 20 units of work have been generated by task 1 and passed through the pipeline. Two such pipelines are randomly assigned to 10 processors with the number of tasks in each processor kept even. Under the experiment, five local processor scheduling algorithms were tested, including Multiprogramming, Processor Sharing, Shortest Processing Time First, Later Pipeline Stage First, and Early Pipeline Stage First. The stage number of a task is determined by its position in the pipeline relative to the head of the pipeline. Among the local processor scheduling algorithms, the Shortest-Processing-Time-First algorithm performs best. However, among algorithms that do not rely on the execution time of a task, there is no significant difference in the performance.

A master-slave job consists of one master task and K slave tasks, in which slave tasks repeatedly receive work requests from the master, compute for time Tm, and send results back to the master. At each work cycle, the master task computes for time Ts, sends requests to slaves, and waits until all slaves return results.

In our experiments, we set $K = 4$ and let both Tm and Ts be exponentially distributed with mean 10.0. Each task runs for 100 work cycles. We distributed four such master-slave jobs to a 5 processor system with the number of tasks in each processor kept even. We tested three scheduling algorithms, including Multiprogramming, Master First and Slave First. The performance of Master First is slightly better than Multiprogramming, while Slave First performs poorly.

From these two experiments, we observe that using task priority to improve job response time may not be effective. Further research is needed to identify task priority assignment for other types of parallel computations and to determine in general the usefulness of task priority scheduling.

5.3 Task Placement Algorithms

Finding the optimal solution for task placement is difficult. Even with a job consisting of independent tasks, the problem of placing them on multiple computers to minimize the job completion time is an NP-complete problem [Garey81]. Nonetheless, in order to respond to a dynamic workload, task placement decisions must be made.

Initial task placement becomes more complicated when the number of jobs in the system increases. The simplest task placement problem occurs when the system has more idle processors than the number of tasks belonging to a job. Nonetheless, the communication and processing requirements of tasks may make the selection of an appropriate number of processors difficult.

When tasks must share processors, additional issues include how to group tasks, how to reduce the interference between tasks belonging to different jobs, and how to balance loads to improve resource utilization. However, additional engineering constraints [Mai82] such as access to secondary storage and sufficient memory size can greatly reduce the possible locations for a task, thereby simplifying the placement problem.

The problem of task placement can be divided into two parts: *location selection* and *task assignment.* A location selection policy deals with allocating resources for a job, with the goal of providing a fair share of processing capability to a given job. A task assignment policy aims at maximizing the performance of a single job with a given allotment of resources. This separation allows the issue of interference among jobs to be dealt with in the location selection policy, while keeping the task assignment policy simple.

One may ask when this separation of concerns for aggregate location selection and task assignment is legitimate. The separation is not acceptable when a strong dependency exists between tasks and processors. Such being the case, tasks need to be individually placed to match its specific requirements with the service provided by processors. For example, in a real-time system with deadline constraints, the resource requirements of a task must be checked against the existing tasks at each processor to ensure that the coming task can complete in time. Cheng et. al.[Cheng86] adopted this one-by-one search approach to schedule time-constrained task groups with precedence constraints. In contrast, our two-step approach allocates processors under the load balancing principle first, then places tasks into these allocated processors.

5.3.1 Location Selection Policies

Design Consideration: Partitioning vs. Mixing

Two distinct approaches for allocating processors to jobs are *partitioning* and *mixing*. These two approaches are at the extremes of a continuum ranging from machine dedication to machine sharing. The partitioning approach dedicates a set of processors to one job, while the mixing approach allows tasks from several jobs to share machines.

Without a task migration facility, the partitioning approach creates a system with a high potential of waste. A job partition, once set up, cannot be changed. When a high priority job arrives, it may preempt a low priority job to claim part or all of the partition from the preempted job. However, when the high priority job terminates, the preempted job resumes only if the entire partition becomes available. Similarly, a new job must also wait until all processors needed for its partition are allocated. The amount of capacity wasted is high, since that partially allocated partitions are not used during the waiting period.

In contrast to the partitioning approach, the mixing approach allows new arriving jobs to join the system without waiting. Although task migration is disallowed, this approach results in an efficient system, owing to the overlapping of resource usage by more than one job. When a job arrives at the system, tasks having the same job priority are allowed to share the processors on an interleaved basis as was proposed in Section 5.1. Moreover, tasks with high job priority preempt tasks with low job priority.

This mixing approach is especially advantageous to low priority jobs. By multiprogramming with high priority jobs, low priority jobs can utilize capacity not used by a high priority job during message waiting or device blocking. The low parallelism of a job can easily be compensated for by other jobs sharing the same set of processors. However, the major disadvantage of the mixing approach is the interference between different jobs. Tasks belonging to different jobs are contending for both CPU and communication services. As a result, the scheduling algorithm must take into account both the dependency between tasks of a job and the resource contention between different jobs.

Global Priority-Based Load Balancing

Load balancing subscribes to the idea of balancing the loads of processors so that the performance of the system can be improved. Based on the load balancing principle, we propose a Priority-Based Mixing algorithm (PBM). The PBM algorithm balances only those loads that can affect an incoming job during resource contention. Under this scheme, a high priority job is placed on the system regardless of the load of low priority jobs. The outline of the PBM algorithm is as follows.

(1) For each processor P and priority-level i, let $L(P, i)$ be the load on processor P due to tasks of priority greater than or equal to i (Here, the load is represented by the number of tasks).

(2) When a job of priority i and k tasks arrives, we produce a list of processors in the ascending order of $L(P, i)$, and assign tasks to the first processor on the list until the processor load $L(P, i)$ becomes equal or greater than the new average load,

$$\frac{k + \sum_{P=n}^{P=1} L(P, i)}{n}$$

where n is the number of processors in the system. Remaining tasks, if any, are assigned to the second processor on the list until its load exceeds the average, and so on until all tasks are assigned.

For the purpose of comparison, we also define a Load-Balancing Mixing (LBM) algorithm, which ignores priorities in placing tasks. LBM is as same as PBM, except that $L(P, i)$ is defined to be the total number of tasks on processor P, rather than those of priority bounded by i.

To illustrate the difference between these two algorithms, Figure 5.1 presents load distributions resulting from the PBM and LBM algorithms. Within each box, rows represent processors. The second and third boxes represent the load distribution after the job arrival. With the PBM algorithm, the two new tasks are assigned to the two processors that do not currently have a high priority task, balancing the load of high priority tasks. The LBM algorithm assigns the two tasks to an arbitrary pair of processors, since all processors have the same initial load, if priority is ignored.

The PBM algorithm allows a high priority job to be assigned to a heavily loaded processor that contains low priority tasks, increasing the parallelism of a high priority job.

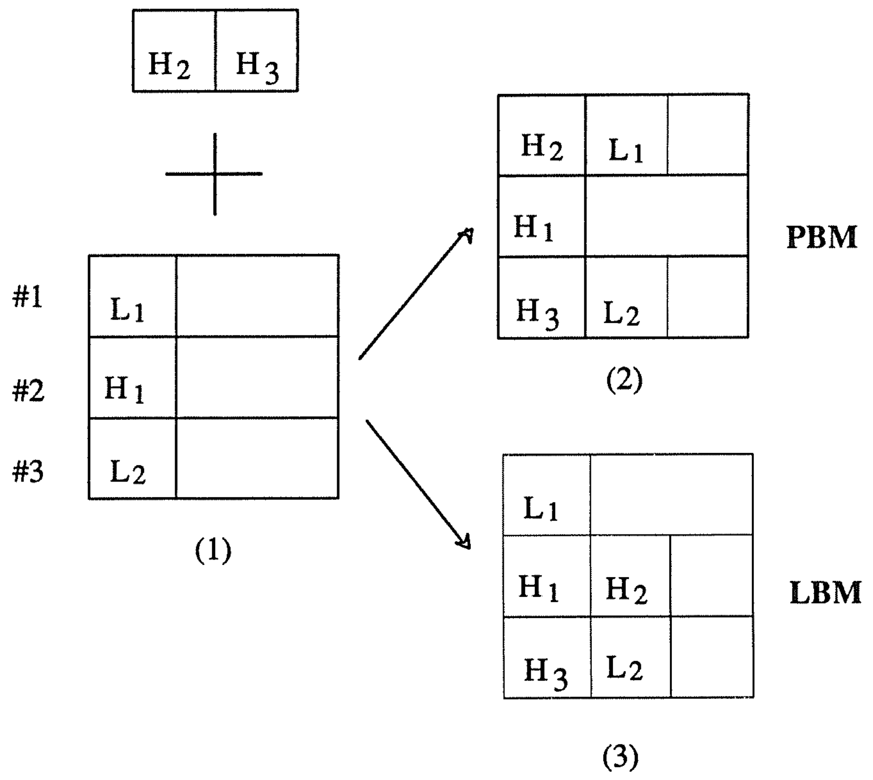

Figure 5.1: An illustration of the load distribution for the PBM and LBM algorithms

5.3.2 Task Assignment Policies

Design Considerations: Task Grouping

A task assignment policy is needed to select tasks for previously allocated processors. The key question to be answered by such a policy is how to group tasks together. Tasks differ in their *resource requirements, communication demands* and *synchronization patterns.* An ideal dynamic task assignment algorithm is one that groups tasks in a way that balances the loads, reduces interprocessor communication, and maximizes the parallelism among tasks.

Owing partly to the complexity of identifying synchronization patterns, and partly to the difficulty of designing general-purpose placement algorithms, the distinct synchronization patterns of distributed computations have not been previously employed in the decision making of task placement. However, many distributed computations are built with very simple parallel structures, such as pipeline [Feijoo85] or master-slave [Finkel86]. Based on the synchronization pattern of a job, we can derive task assignment heuristics as follows:

> From the interaction of tasks belonging to a distributed computation, one can find sets of tasks that with high probability can proceed in parallel, such as the slaves of a master. This probabilistic parallelism is inherent in the synchronization pattern of a distributed computation. Once the sets of potentially parallel tasks are found, the response time of the job may be improved through placing tasks exhibiting high parallelism in separate processors.

However, we need a way to clearly specify the parallel structure of a distributed computation in order to derive task assignment heuristics. For this purpose, a Parallel Structure language **PS** was devised and implemented. PS encapsulates both the synchronization structure and the stochastic resource requirements of a distributed computation in a structured form. The next chapter gives a full account of the PS language and the synthetic workload generator.

Task assignment algorithms based on knowledge of a job

We devise a family of task assignment algorithms for distributed computations with a pipeline structure, which contains multiple tasks connected together to process a stream of data. A task is blocked if its predecessor does not supply data or its successor does not receive data. These algorithms differ in their usage of information about the pipeline structure. The *Random* algorithm uses no knowledge of the structure of the job, while the *Task-Clustering* algorithm uses knowledge of task communication traffic. Finally, the *Parallel-Task-Declustering* algorithm employs knowledge of the parallel structure of jobs.

Random (RA)

Tasks are randomly assigned to the allocated processors while satisfying the load balancing requirement.

Task-Clustering (TC)

This algorithm attempts to ensure that tasks that communicate with each other are clustered at a processor without violating the load balancing requirement. The rationale behind this algorithm is to reduce the inter-processor communication traffic. The algorithm is to assign tasks to processors in groups of $\frac{n}{k}$, where n is the number of tasks, and k is the number of processors allocated to the job. The outline of the TC algorithm for the pipeline structure is as follows:

(1) List tasks according to their processing stage number in the pipeline.

(2) For each processor, assign tasks according to the order of the task list until the load of the processor is equal to or greater than the average load.

Parallel-Task-Declustering (PTD)

The parallel-task-declustering algorithm aims to prevent tasks that can proceed in parallel from being assigned to the same processor. Under this guideline, we developed a PTD algorithm for a distributed computation with pipeline structure.

To improve the parallelism of a pipeline structure, one should check and balance the progress of each task to ensure that its current progress does not cause future blocking. However, for the startup and shutdown phases, balancing the progress of tasks can be achieved by physically separating neighboring tasks. The rationale is that when a task completes a cycle of work, it produces immediatably executable work for its successor, and becomes ready to process new work for its predecessor. The outline of the PTD algorithm for pipeline is as follows:

(1) List tasks according to their processing stage number in the pipeline.

(2) Assign tasks one by one in a round-robin fashion to all allocated processors.

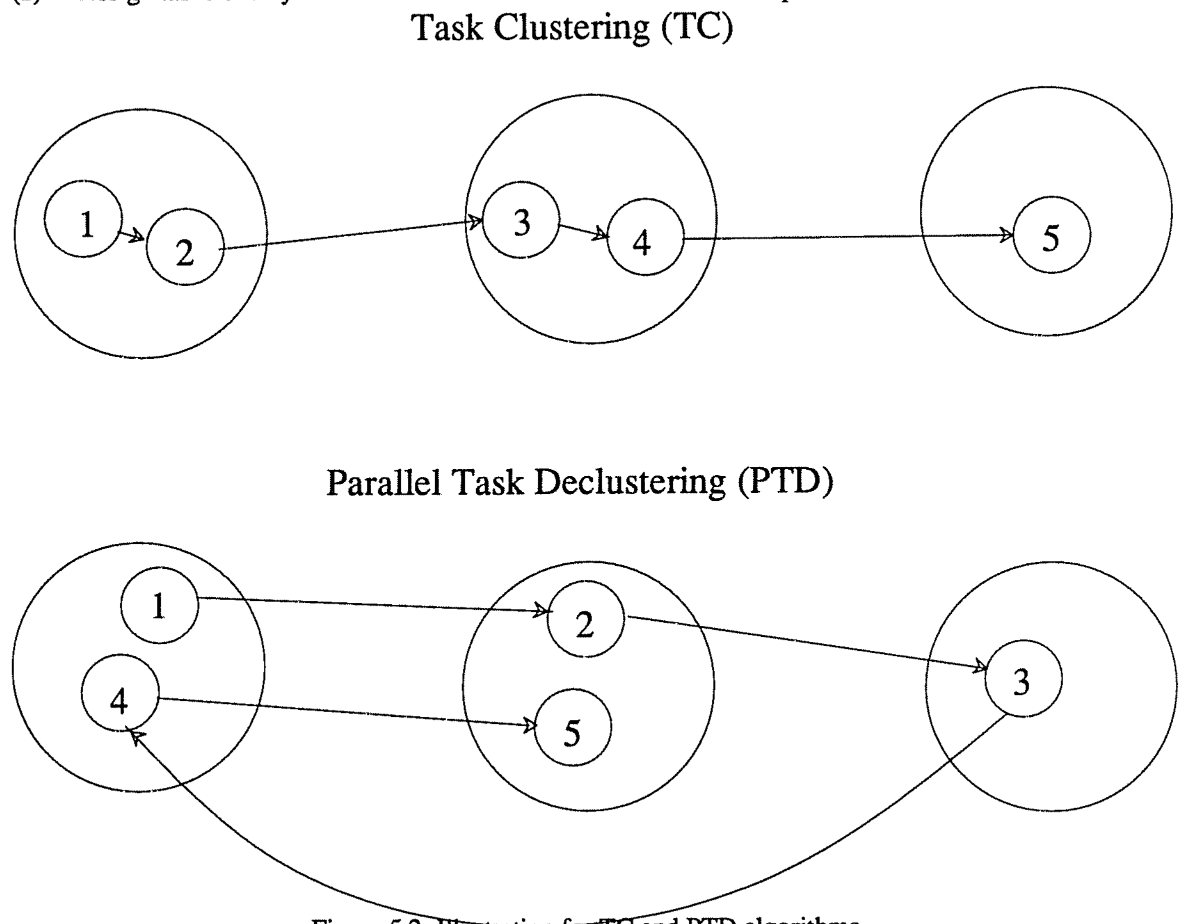

Figure 5.2: Illustration for TC and PTD algorithms

5.4 Performance Evaluation

Each Composite Priority Scheduling algorithm consists of three components: a processor scheduling discipline, a location selection routine and a task assignment routine. For the processor scheduling, we use a modified version of Preemptive Resume as described in section 5.1. For the location selection routine, PBM and LBM are the two candidates. For the task assignment routine, RA, TC, and PTD are the three alternatives. As a result, we have six algorithms for consideration. To compare these six Composite Priority scheduling algorithms, we have constructed a simulation testbed, whose organization is described in the next chapter, to simulate a distributed system with a dynamic workload.

5.4.1 The Simulation Environment

System Model

The simulated system consists of M computers interconnected by a local area network. The overhead of initial task placement is assumed to be the same for all tasks on all processors. In addition, the network is targeted for light loads, whose performance characteristics can thereby be modeled by an infinite server, which provides constant response time regardless of number of customers in the system, or a first-come-first-serve queue.

Several assumptions about operating system support are made. An interprocess communication (IPC) facility for tasks is provided by the operating system. This facility performs basic *send* and *receive* operations. The cost of intermachine communication is different from that of intramachine communication. The send and receive operations incur both processor and network overheads. These costs are simulation parameters. A task is allowed to run on any computer in the system, but a task once started, cannot be migrated.

The simulated system contains two priority classes, with high priority jobs accounting for 30% of the offered load. Interarrival times of jobs are exponentially distributed. A dynamic workload of tasks with the pipeline structure (see Section 5.2.2 for a detailed description) is assumed. The parameters of the simulated system are summaried in Table 5.1.

System Parameters	Values
Network Size	10
System Load	0.1-0.8
Job Arrival Process	Poisson (determined by the load)
Priority Classes	2
Priority Jobs	30% of jobs having high priority
Intermachine Message Cost	0.1 - 8.0
Intramachine Message Cost	0.0
Job structure	Pipeline
Pipeline Parameters	**Values**
Number of Stages	Constant 20
Service Time per Cycle	Exponential (mean= 20)
Number of Cycles	20
Buffering Scheme	Dual Buffering

Table 5.1: System Parameters

Performance Metrics

The two performance metrics used in this study are job response time and *average parallelism* (AP). The AP of a job is defined to be the ratio between the total CPU demands of all tasks in the job and the job response time.

$$AP = \frac{\text{Total CPU Demands}}{\text{Response Time}}$$

The average parallelism is an estimate of the average number of concurrent processors used by a particular job [Eager86].

5.4.2 The Effect of Task Assignment

Figure 5.3 plots the response time curve of PBM+RA, PBM+TC, and PBM+PTD with message overhead 1.0 under varying offered load from .1 to .4. The RA algorithm results in the worst performance among the three at most cases. This demonstrates that using knowledge of task properties can improve the system performance. However, to determine when and what task structure can result in significant improvement is an open question. PTD performs better than TC in the range of loads examined. However, the improvement is subject to the cost of message transfer, because PTD inherently generates more intermachine communication than TC does. Therefore, we examine the performance of these algorithms under the range of message cost from .1 to 8.0. Since system utilization reaches 90% at load 0.5 when the message cost is 8.0, the offered load is set at .4 to keep the system from becoming saturated. Figure 5.4 plots the response time of PBM+RA, PRM+TC, PBM+TC algorithms under varying message cost. PTD is superior at low message costs while TC is superior when the message cost is high. The response time improvement of PTD over TC for high priority jobs under PBM has decreased from 30% to 4% when the message cost changes from 1.0 to 0.1. A crossover point of both curves is at a message cost of 2.0, which is 20% of the task execution time (the cycle time is 20 and 2 messages are transferred per cycle). However, the difference of performance between PTD and TC at low message costs is much less than that at high message costs. Therefore, the TC algorithm is preferable as being a robust algorithm against increased network overhead under adverse conditions.

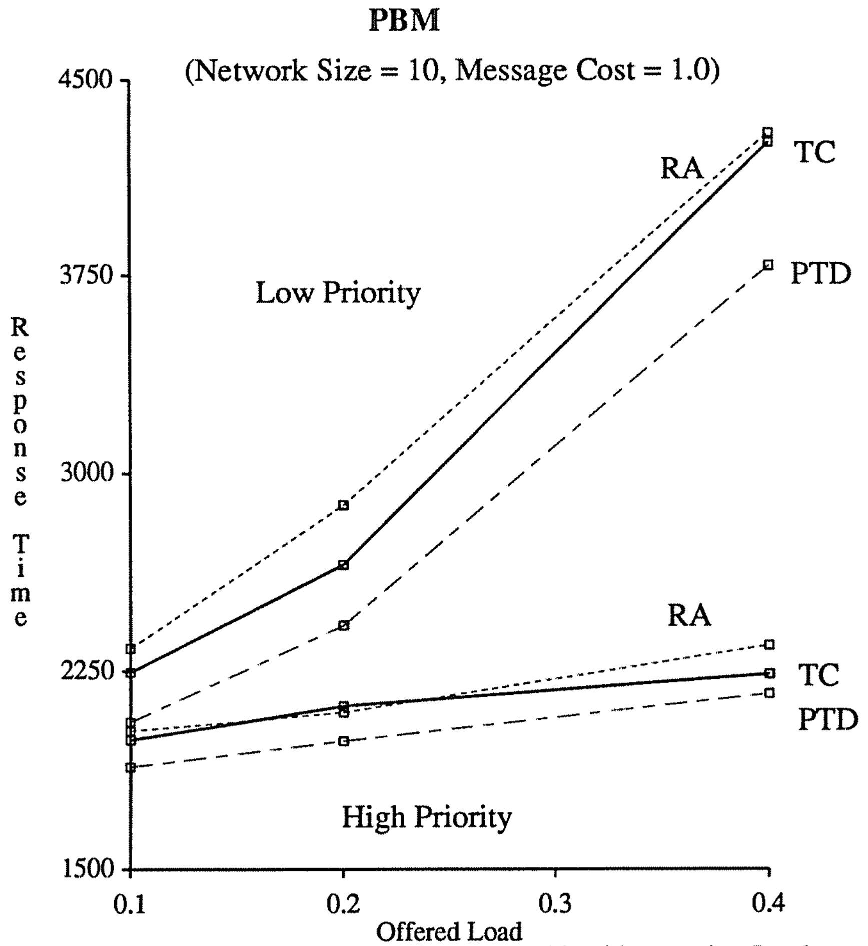

Figure 5.3: Response Time for RA, TC and PTD Algorithms against Load

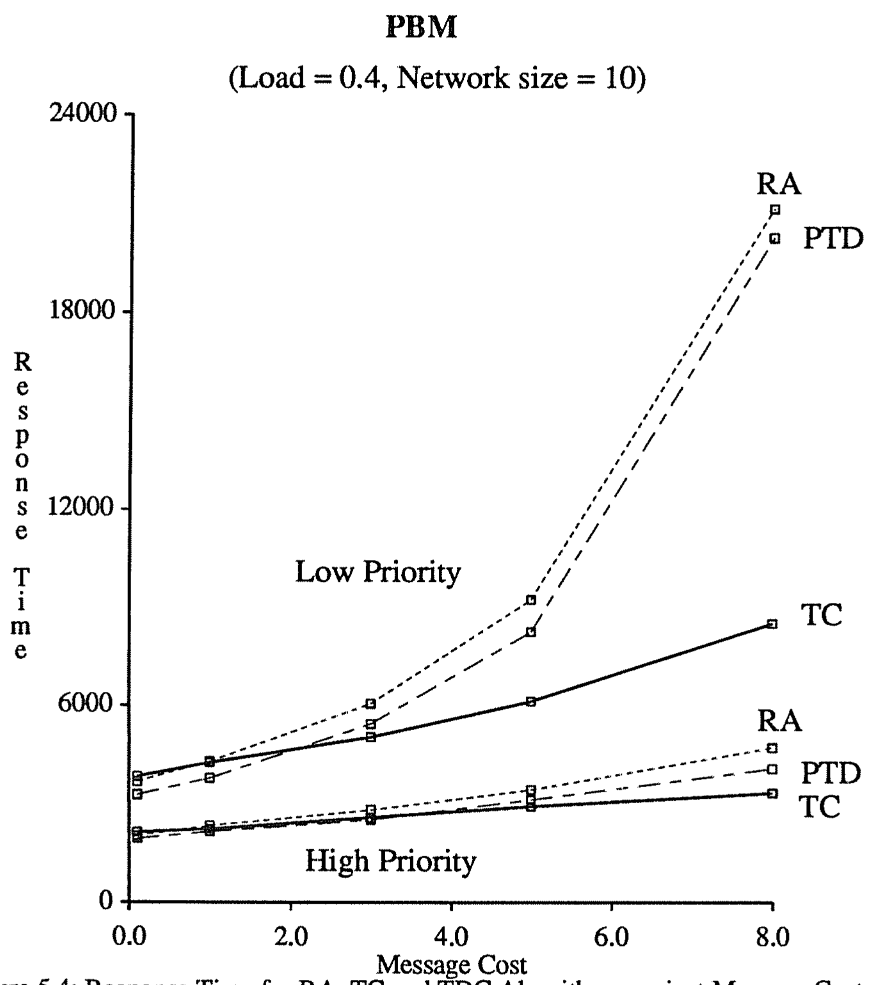

Figure 5.4: Response Time for RA, TC and TDC Algorithms against Message Cost

The selection of a workload that runs a 20 stage pipeline on 20 cycles of work accentuates the effect of startup and shutdown. This bias explains why the PTD algorithm performs better than TC under low message cost. With a large number of cycles, the whole pipeline becomes evenly loaded, and there is no advantage of PTD over TC in terms of promoting parallelism.

5.4.3 The Effect of Priority-based Location Selection

While both the PBM and LBM location selection policies attempt to balance the load of the system, they differ in their distribution of priority-based loads. As we discussed in the previous section, the PBM algorithm results in a higher probability of task preemption when tasks are initially placed. This fact implies a possibly greater parallelism in servicing a high priority job than under the LBM algorithm.

Figure 5.5 plots the response time curves of high priority jobs for six algorithms under varying offered load. These six algorithms are PBM+TC, PBM+PTD, PBM+RA, LBM+TC, LBM+PTD and LBM+RA. Figure 5.6 plots the response time curves of low priority jobs for the same six algorithms. One expects high priority jobs to receive better service under the PBM algorithms because with high probability high priority jobs get more parallelism. Note that we are using a Preemptive Resume Priority scheduling algorithm at each processor. Therefore once a high priority task is placed on a processor it

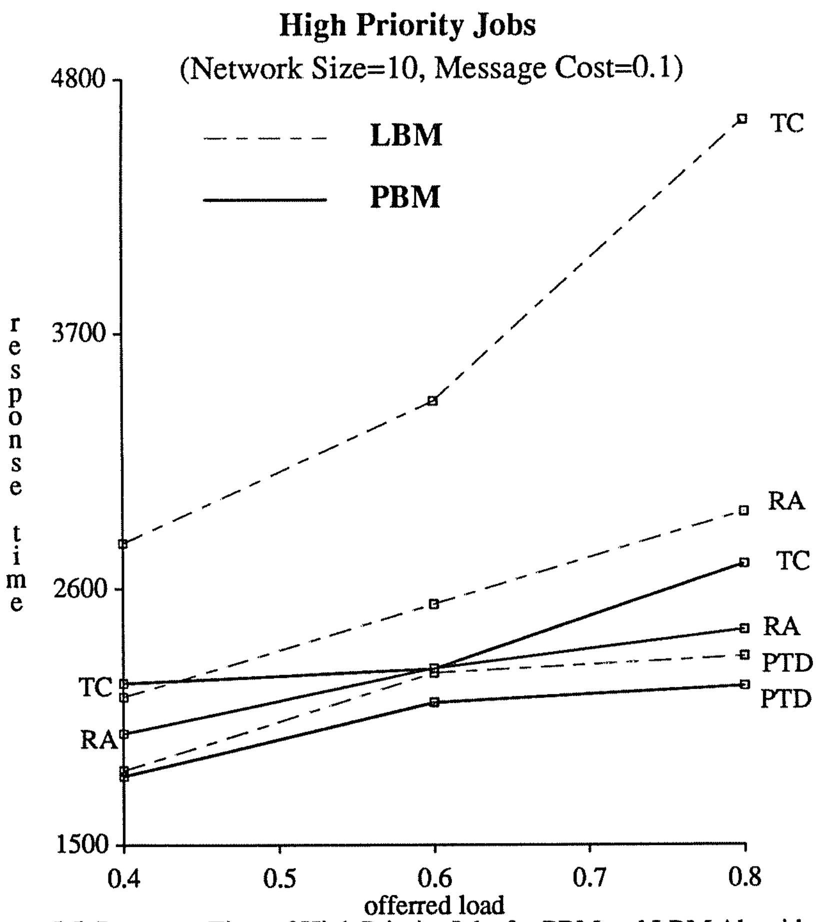

Figure 5.5: Response Time of High Priority Jobs for PBM and LBM Algorithms

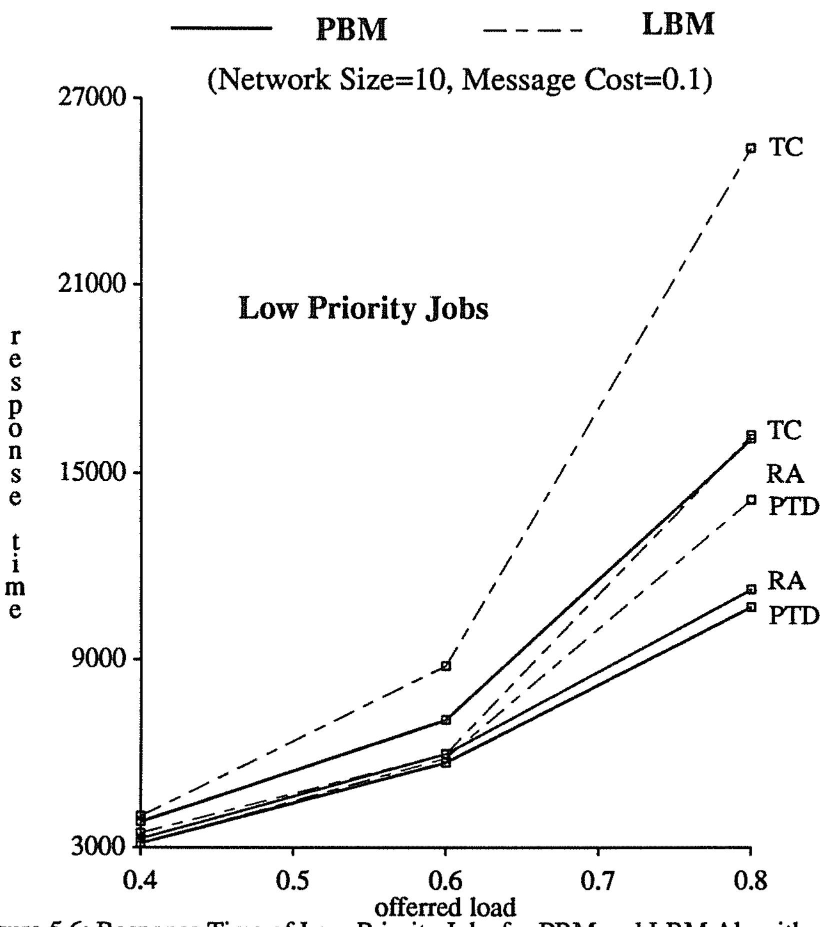

Figure 5.6: Response Time of Low Priority Jobs for PBM and LBM Algorithms

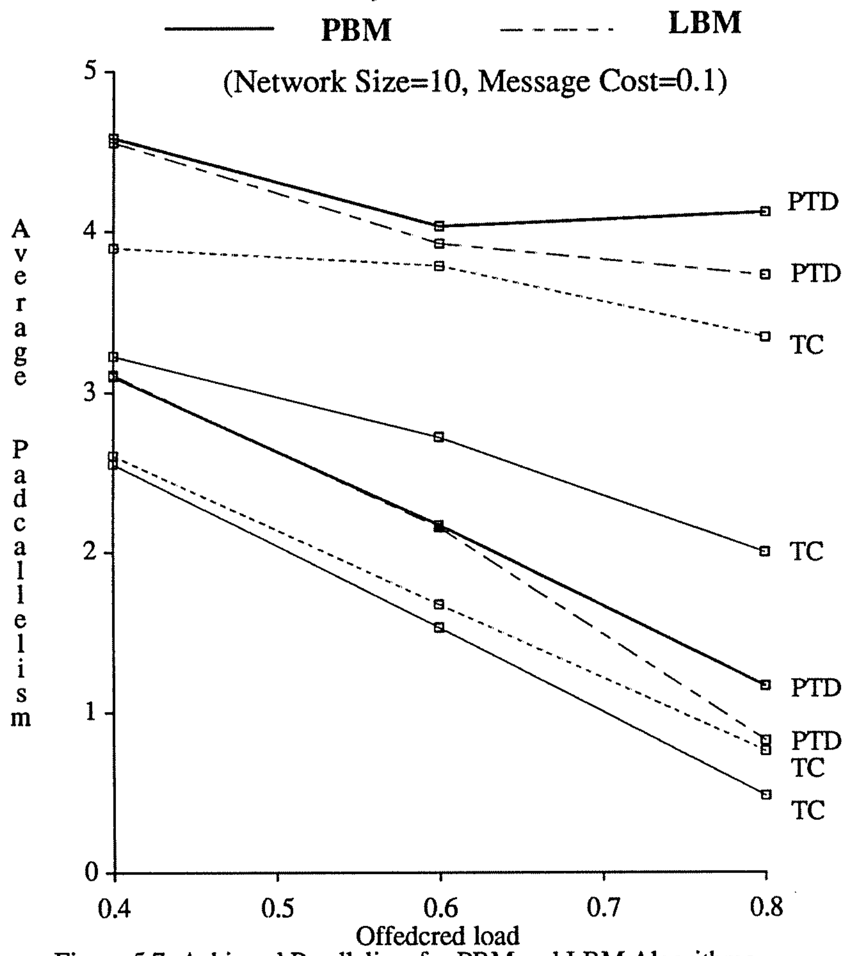

Figure 5.7: Achieved Parallelism for PBM and LBM Algorithms

can preempt currently running low priority tasks. Surprisingly, low priority jobs under PBM also perform better. As is shown in figures 5.5 and 5.6, both high and low priority jobs receive better quality service under the PBM algorithm (solid lines) than under the LBM algorithm (dotted lines). The improvement of PBM over LBM increases with the offered load. The improvement in percentage of PBM over LBM is most significant under the TC task assignment algorithm. For low priority jobs, the improvement increases from 6% to 50% as loads increase from 0.4 to 0.8, while for high priority jobs the improvement increases from 30% to 70%. However, under the PTD algorithm the improvement is less. It is on the average 5% for high priority jobs and 20% for low priority jobs.

The poor performance of low priority jobs under LBM can be attributed to the blocking caused by synchronization. Since a pipeline structure contains highly correlated tasks, low priority jobs are most susceptible to this type of blocking with a high probability of incurring long queueing delay. Since the LBM location selection algorithm does not balance the load of high priority jobs, it causes drastically uneven delays to low priority tasks in different processors. Therefore, although the service received by high priority jobs is worse under LBM than PBM, the low priority jobs do not receive the benefit of the available capacity. Since TC task assignment algorithm aggregates neighboring tasks, the probability to form a pipeline bottleneck is higher than that of the PTD scheme, showing a larger performance difference under PBM and LBM.

Figure 5.7 plots the curve of *average parallelism (AP)* for PBM and LBM. This figure gives the average speedup of jobs in the same priority class.

5.5 Conclusion

Dynamic priority scheduling for distributed computations is extremely complex because of the interaction between workloads and scheduling algorithms. A notion of Composite Priority is introduced in this chapter to allow a distributed computation to be assigned priority on both job level and task level. To support Composite Priority, we have explored and identified a collection of alternative policies for processor scheduling, location selection, and task assignment. We evaluated these proposed alternatives under a workload with computations of a pipeline structure. We simulated varying system loads and workload parameters. The major conclusions from the simulation results are:

(1) Assigning task priority may not be able to bring significant performance improvement to the distributed computation. Our experiments on assigning task priority to the tasks of pipeline and master-slave jobs showed no performance improvement due to priority scheduling of tasks.

(2) Under the Priority-based Mixing algorithm the performance of high priority jobs can be significantly improved. Owing to the correlation between the tasks of a distributed computation, it is crucial to both high and low priority jobs to do priority-based global task placement.

(3) The startup and shutdown effects of a pipeline job favor the Task-Clustering (TC) task assignment algorithm when the message cost is high, but the Parallel-Task-Declustering (PTD) performs better when the message cost is low. With a large number of pipeline cycles, TC is best. The result shows the sensitivity of task assignment algorithm to message cost, pipeline cycles, and system loads. Moreover, the difference between the performance of TC and PTD increases when the offered load increases.

(4) The Random Assignment algorithm performs worse than both PTD and TC under most message costs and system loads, indicating the benefit of using knowledge of the computation structure.

Chapter 6
A Testbed of Parallel Structures

To study the interactions between the structure of distributed computations and the performance of resource allocation algorithms, we built a testbed that simulates distributed computing systems, allowing performance statistics to be collected under a control environment. This testbed has proved useful in obtaining the results presented in chapter 5.

To avoid the implementation details of an operating system while capturing the overhead of system services, we base our testbed on a high level simulation model. For this testbed, we need a way to specify the synthetic workload of distributed computations. Since distributed computation contains complex correlations among tasks, we choose to represent distributed computations with a high level language PS (Parallel Structure). Therefore, this testbed allows us to separate workload representation, using PS, from the system model representation, using DENET [Livny87]. In this chapter, a detailed description of the testbed is presented, The salient features of this testbed include a flexible language, PS, with which the parallel structure of distributed computations can be specified, the capability to instantiate copies of distributed computations based on PS templates, and modularity of replaceable functions, such as network management and job placement.

In the next section, we give a description of the PS language. The structure of the testbed is presented in section 2. Section 3 describes the runtime interface between the synthetic workload generator and the system simulator. Section 4 discusses some experience with the testbed.

6.1 PS — a language for Parallel Structures

Serving as the frontend for the synthetic workload generator, the PS language is a flexible vehicle to specify the parallel structure of distributed computations. Any parallel structure that can be specified in terms of CPU delays, device delays, and receive and send primitives for interprocess communication, is representable in the language. This language is similar in the power of workload representation to the B-language used in cm* [Singh80]. While B-language is equipped with a graphical interface, this language uses a text representation of interconnections between tasks.

Since the goal of the language is to model the high level behavior of a distributed computation, control structures are kept simple in PS. Two kinds of control structures are permitted in the language: iteration and switch. In addition, the language provides a particular type of variable called *stochastic variable* that represents an active random number. Such a variable can be used in conjunction with other language constructs to express the stochastic behavior of the program execution time. These variables, whenever being referenced, return a new value following the designated distribution. For example, a stochastic variable may be used as a loop counter inside an iteration construct, causing the loop to iterate a random number of times.

6.1.1 A Central Server Workload

By following through a simple example, we can demonstrate some of the basic facilities of PS. The example includes a central server and a number of clients.

The server processes client requests in a First-come-first-served manner as follows:

> Every client request requires an average of 20 units of time to process, with the exact processing time following an exponential distribution. The server accesses a local disk with 60% probability. If an disk access is chosen, the information transferred follows a normal distribution with mean 5K and variance 1K.

In addition to requests to the central server, a client behaves as follows:

> A client goes through the cycle of processing, sleeping, and waiting for service by the central server. The processing time is exponentially distributed with a mean of 100 time units. The sleep time is a constant of 200 time units. In addition, with probability 0.4, a client may terminate itself after waking up (from sleep).

In order to represent these resource demands, two *stochastic variables,* representing IO requests and CPU demand, are used. The resource usage of the central server takes the following form:

```
VARIABLE
        IO_request : FIXED NORMAL (5000.0, 1000.0);
        CPU_demand : EXPONENTIAL (20.0);
CODE
        CPU with CPU_demand;
        RANDOM SWITCH {
        0.6:
                DISK with IO_request;
        ELSE
                (* nothing *)
        }
```

Here, with the RANDOM SWITCH, the program selects a path to execute according to the branching probability of cases. By stating *CPU WITH x,* the program requests *x* amount of a CPU resource. Similarly, *DISK with* y requests y amount of disk service.

Let's now turn our attention to a client. Since a client iterates through the CPU-sleep-request cycle, the program needs a looping construct, which terminates after each iteration with probability 0.4. However, the termination condition can also be expressed as the number of iterations following a Bernouli distribution with probability 0.4. Expressed in PS terms, it takes the following form.

```
(* For the client *)
VARIABLE
        CPU_work : EXPONENTIAL (100.0);
        nloop : BERNOULI (0.4);
CODE
        ITERATE (nloop) {
                CPU with CPU_work;
                SLEEP with 200.0;
        }
```

The statement *ITERATE (K) { ... }* indicates that the loop body is repeated K times. *SLEEP with X* indicates that the program is to be suspended for X amount of time.

Though above code is not a complete program, it shows the essential characteristics of a PS program. It is clear that a PS program is a stochastic model of a real program. PS provides stochastic variables and other language constructs to describe the stochastic behavior of a real program. One problem with such a facility is that the integrity of a probability distribution depends strongly on the initial values of a random number generator. For example, if the ITERATE statement in the previous example only repeats 1 time (this case is possible, because the loop counter follows a Bernouli trial distribution), then the value of cpu_work would be used once only.

PS also provides facilities for modeling the interaction of processes. Since message passing is the most common communication paradigm of distributed operating systems, we introduce two simple primitives: nonblocking send and blocking receive. These primitives are relatively low level, allowing high level communication abstractions such as a remote procedure call [Spector82] to be simulated. To transmit data from a client task to the central server task, the client task must issue a *SEND* request, while the central server task issues a *RECEIVE* request. To synchronize a client and the central server, the client would issue a RECEIVE request to wait for the completion notification from the central server. Meanwhile, after the server completes the processing of a client's request, it may issue a SEND to notify the client about service completion. Therefore, the server and the client programs are:

```
(* the central server *)
VARIABLE
        IO_request : FIXED NORMAL (5000.0, 1000.0);
        CPU_demand : EXPONENTIAL (20.0);
        client : TASK_POINTER;
CODE
        RECEIVE ANY client;
        CPU with CPU_demand;
        RANDOM SWITCH {
        0.6: DISK with IO_request;
        ELSE
                (* nothing *)
        }
        SEND client;

(* the client *)
PARAMETER
        MyServer : TASK_POINTER;
VARIABLE
        CPU_work : EXPONENTIAL (100.0);
        nloop : BERNOULI (0.4);
CODE
        ITERATE (nloop) {
        CPU with CPU_work;
        SLEEP with 200.0;
        SEND MyServer;
        RECEIVE MyServer;
        }
```

The declarations of *client* and *MyServer* establish these identifiers as pointers to tasks. Since *MyServer* is to be assigned at run-time, it appears in the form of a parameter. The statement *"RECEIVE ANY client"* allows message reception from any task, with the pointer of the sending task being saved in the local variable *client* for acknowledgement purpose.

In the next and final version of the task template, we complete our text such that it becomes a genuine PS program.

```
JOB MODULE CentralMonitor;

                CentralServer = {
                (* the central server *)
                VARIABLE
                        IO_request : FIXED NORMAL (5000.0, 1000.0);
                        CPU_demand : EXPONENTIAL (20.0);
                        client : TASK_POINTER;
                CODE
                        RECEIVE ANY client;
                        CPU with CPU_demand;
                        RANDOM SWITCH {
                                0.6: DISK with IO_request;
                        ELSE
                                (* nothing *)
                }
                SEND client;
        }

        AClient = {
        (* the client *)
        PARAMETER
                MyServer : TASK_POINTER;
        VARIABLE
                CPU_work : EXPONENTIAL (100.0);
                nloop : BERNOULLI (0.4);
        CODE
                ITERATE (nloop) {
                        CPU with CPU_work;
                        SLEEP with 200.0;
                        SEND MyServer;
                        RECEIVE MyServer;
                }
        }

        JOB TaskForce (num : INTEGER) = {
        TASK
                S : CentralServer;
                clients : ARRAY [1..num] OF AClient;
        INIT
                clients [1..num].MyServer = S;
        }

END CentralMonitor.
```

The essential addition in this step is the outer structure of task templates and the text of a job template. In PS, task templates have names, which later serve as the names of task type in the job template. The job template begins with the word *JOB*, followed by a name, a set of formal parameters and the job body. A job template describes how the job is to be initialized, which includes generating tasks, and initializing their parameters. In this example, the statement *S : CentralServer* invokes a central server task, while the declaration of an array of AClient in *"clients : ARRAY [1..num] OF AClient,"* invokes a group of *num* client tasks. To initialize a task, the parameters of the task must be assigned. Here, *S* representing the central server has a type of *task_pointer*, while the *MyServer* parameter of clients has also a type of *task_pointer*. The assignment of S to MyServer connects client tasks to the central server.

The complete program is called a *job module*, given a name (CentralMonitor), and has the following format:

```
JOB MODULE modulename ;

(* a task template *)
name = {
        <parameter declarations>
        <variable declarations>
        <statements>
}

(* a job template *)
JOB name ( <argument list> ) = {
TASK
        <task declarations>
INIT
        <statements>
}
END modulename.
```

A few more comments concerning our example are in order. This discussion of our first example has been very informal, because the goal was to explain an existing program. However, the purpose of PS is to describe a large group of parallel structures. For this purpose, only a precise, formal description of our tool is adequate. In the following sections of this chapter, we introduce a formalism to define the syntax and to discuss the semantic meaning of various features.

6.1.2 Expressing Resource Demands

The resource demands of a computation are expressed separately for different devices. Special device names, such as CPU, SLEEP, and DISK are included as permanent parts of the language, while other devices may be added using the DEVICE primitive. Since the behavior of each device is subject to the interpretation of the simulator at run-time, the language purposely leaves out the definition of devices' physical interpretation. The syntax for expressing resource demands is:

```
CPU WITH <expression> ';'

SLEEP WITH <expression> ';'

DISK WITH <expression>  ';'

DEVICE <integer> WITH <expression>  ';'
```

6.1.3 Intertask Communication Primitives

Two intertask communication primitives are provided: *send* and *receive*. The send primitive is non-blocking, while the receive primitive blocks. A non-blocking send implies that the operating system buffers those messages that were sent but not yet received by the destination task. The receive primitive permits three kinds1 of message screening: receive from a task, receive from any task of a task group, and receive from any task. Although there are other forms of intertask communication primitives, in many cases it is possible to simulate other primitives using these two primitives. For example, a Remote Procedure Call can be simulated by posting a receive immediately after a send. However, the system overhead of a Remote Procedure Call is different from the overhead of the acknowledged send and receive. Therefore, to simulate other IPC primitives, the system level testbed must also be modified to reflect the correct system overhead. Lots of other variants of intertask communication primitives cannot be easily modeled by these two primitives, such as message screening according to the message content. However, one of the virtue of PS is its flexibility, since the parser generator FMQ [Fischer80] allows us to modify the PS compiler easily. The syntax of send and receive is:

```
<task_name> ':' TASK_POINTER;
<task_group_name> ':'
ARRAY '['<expression>'..'<expression>']'
        OF TASK_POINTER;

(* receive *)
RECEIVE <taskname>
RECEIVE <taskgroup> '[' <expression> ']'
RECEIVE ANY <task_variable>
RECEIVE ANY <task_variable> OF <taskgroup>
```

```
(* send *)
SEND <taskname>
SEND <taskgroup> '[' <expression> ']'
```

6.1.4 Stochastic Variable

The stochastic variable is of particular importance to the language. Since the goal of the language is to represent the workload of a large class of distributed computations, the capability to represent resource demands in stochastic terms is vital. These variables allows the testbed to simulate a distributed computation without using a detailed trace of a parallel algorithm. A stochastic variable is declared as:

```
<variable_name> ':'
        [ FIXED ] <function> '(' parameters ')'
```

Example:

```
(* Uniform distribution : lower bound, upper bound *)
<randvar>   ':'
        UNIFORM '(' <LowerBound> ',' <UpperBound> ')'

(* Hyper-exponential distribution : α,υ1,υ2 *)
<randvar>   ':'
        HYPER '(' <alpha> ',' <mean1> ',' <mean1> ')'

(* Exponential distribution : υ *)
<randvar>   ':' EXPONENTIAL '(' <mean> ')'

(* Normal distribution : mean, variance  *)
<randvar>   ':' NORMAL '(' <mean> ',' <variance> ')'
```

6.1.5 Iteration

An iteration construct contains an iteration counter and a body. The body may contain any combination of statements. The counter is evaluated once at loop entry time. This construct has the form:

```
ITERATE '(' <expression> ')' '{'
        <statements>
'}'
```

If <expression> is null, this is an infinite loop.

6.1.6 Random switch

Random switch provides a facility for doing probabilistic branches. Every branch is labeled with the probability of taking that particular path except the *ELSE* part, which has a probability equal to one minus the previous total. The compiler checks that the total probability is less than or equal to one. This construct takes the following form:

```
RANDOM SWITCH '{'
        { <probability_constant> ':'
                <statements> }
        ELSE
                <statements>
```

```
                    '}'
```

6.1.7 Task Template

A task template is a combination of task parameters, variables and an execution body. The template serves as a generic module to be used in the declaration of a distributed computation. It is convenient to write one template for multiple tasks having slight variations. For example, one task template can be used to represent all the stages of a pipeline structure, with variations to be specified as parameters. In this way, the complexity of maintaining similar tasks can be greatly reduced.

The parameters of a task template must be initialized at runtime. These parameters may have the type of TASK_POINTER, INTEGER or REAL. Initialization of these parameters is discussed in the next section.

```
name = {

        PARAMETER
                <parameter declarations>
        VARIABLE
                <variable declarations>
        CODE
                <statements>
        }
```

6.1.8 Job Template

A job template declares a job and all its related tasks. The template itself is translated into an executable routine to be invoked at runtime to initialize the template.

In a job template, all parameters of task templates must be initialized. PS provides an assignment statement for initialization. To initialize a TASK_POINTER, one must first declare a new task, which has a previously defined task template as its type. One can also declare a group of tasks in the format of an array of TASK_POINTERS. An implicit loop is provided in the assignment of array of TASK_POINTERs, for example, we assign a task with index i to a task with index i+1. This implicit loop scheme takes the following syntax:

```
GroupOne[i:1..10] = GroupTwo[i+1];
```

Moreover, a new task may be connected to a nil task. We introduce a special constant, BLACKHOLE, to represent a NIL task pointer. To send or to receive on the BLACKHOLE has no side effect but falling through.

To conveniently initialize a group of tasks, we provide a *with* construct having the following syntax:

```
<parameter_name> '=' <expression> ';'

WITH <task_group_name>
        ['[' 'i' ':'<integer> '..' <integer> ']' ] DO
        <parameter_name> '=' <expression> ';'
END;
```

For example,

```
WITH PipelineStages [i:1..4] DO
        NextStage := PipelineStages[i+1];
END;
```

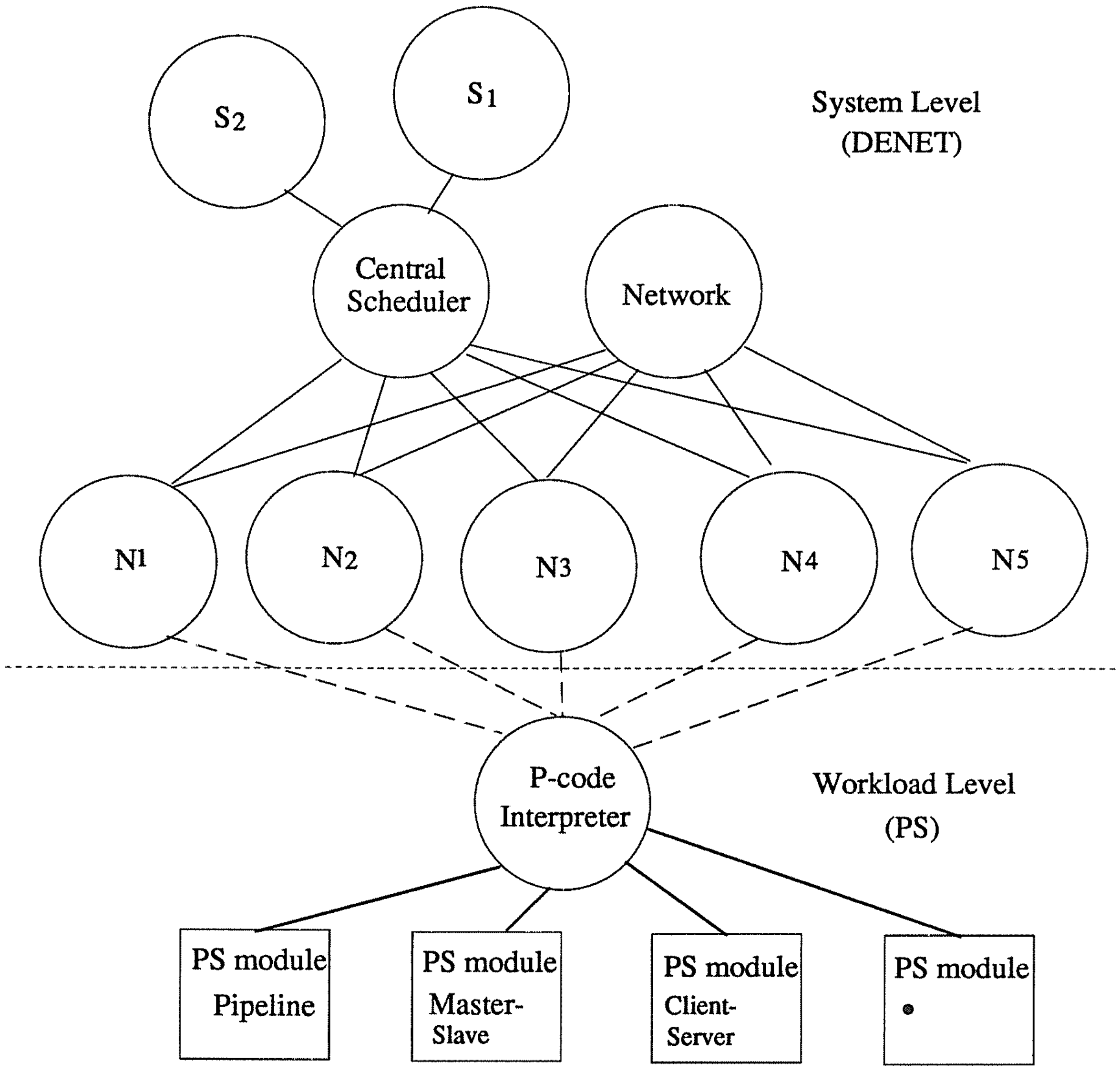

Figure 6.1: the structure of the testbed

6.2 The structure of the testbed

Our testbed consists of a compiler and a simulator. The compiler translates PS for Modula-2 [Wirth82]. The output consists of an array containing the pseudo code of the parallel computation and an invocation routine that produces interpretable instances of such pseudo code.

The simulator has two levels: the workload level and the system level. The workload level provides the interpreter for pseudo codes, and controls the invocation of parallel jobs. The system level controls the simulation, and is responsible of simulating the local area network, the CPU processing, and the specific resource allocating algorithm under study. The system level interacts with the workload level through a procedure call interface, through which the workload level supplies a stream of resource demands of distributed computations. At the end of a simulation, the simulator outputs a set of statistics describing the performance of the system.

Figure 6.1 shows the overall structure of the testbed. The upper half represents the system level, while the lower half represents the workload level. The system level is composed of four kinds of resource management modules. The first kind is a CPU server, labeled N_x with x representing the index of the server. The second kind is a network server, which supports message passing and task transfer among CPU servers. The third kind is a global scheduler, which provides the system with job placement functions. Lastly, there is a pattern-specific scheduler, labeled S_x with x representing its index. A pattern-specific scheduler provides the global scheduler with rules for scheduling a particular parallel structure.

The workload level contains an interpreter, which interprets the pseudo codes of parallel structures, and initialization routines for generating tasks from PS templates. The boxes in figure 6.1 represent these pseudo code and initialization routines of PS templates.

The compiler is written in PASCAL, using a scanner generator and FMQ parser generator developed by Fischer [Fischer80] at the University of Wisconsin. The simulator is written in DENET [Livny86], which is a Modula-2 based language specialized for distributed network simulation.

6.3 The runtime interface

This section describes the runtime interface between the simulator and the PS templates. The description is divided into two parts: the runtime creation of new jobs/tasks and the pseudo-code interpreter.

Creation of New Jobs and Tasks

The memory allocation scheme provided by Modula-2 is extensively used to maintain the dynamic context for new jobs and tasks. The sequence of events for the creation of new jobs/tasks is as follows.

(1) The workload management module in the simulator decides when to create the next job. The module selects a job type, such as a pipeline, and furnishes all the related parameters for this job, such as the number of stages and the mean execution time of each stage.

(2) Based on this information, the workload management module calls the job template routine. The job template routine allocates a context block for each task it creates and fills it with all the necessary initial parameter values.

(3) Finally, the job template routine returns the pointer of the job context block, in which the pointers to task context blocks are stored along with other structures, such as the mailbox for send/receive.

The Pseudo Code Interpreter

The pseudo-code interpreter is an integral part of the PS compiler, because it must interpret the pseudo code generated by the compiler. The interpreter has one routine accessible by the simulator:

```
Interpret ( taskptr : taskpointer; node : INTEGER;
            VAR request : request_rec);
```

This routine allows the simulator to interpret a new task pointed to by the *taskptr*. The *interpret* routine keeps track of the last instruction of the task and continues to interpret the pseudo code to find the next breakable point, which is a point at which the task needs an outside service. The *interpret* routine returns to the simulator with a request, when the breakable point is encountered.

There are five types of request from the interpreter to the simulator: CPU, DEVICE, SEND, RECEIVE, and SET_RANDOM_NUMBER. A CPU service specifies the amount of CPU usage, a

DEVICE service specifies the DEVICE related parameters, a SEND request specifies the destination and the content, a RECEIVE request is similar to the send except that the source of the message needs not be specified, and the SET_RANDOM_NUMBER request asks the simulator to find a random number sequence and reset the sequence for the interpreter. Therefore, the simulator is responsible for all resource usage of the interpreter. The pseudo code, then, represents the resource demand, while the simulator represents the resource allocator.

Extensive tracing capability is provided for the interpreter, so that various levels of traces can be printed, triggered by setting appropriate flags at runtime. This capability has proved useful when debugging the PS programs and resource scheduling algorithms.

6.4 Experiences with the Testbed

6.4.1 PS programs

Since the development of a distributed workload using the PS language is the most unusual part of this testbed, some experiences are worth mentioning. The preparation of a new PS program is an interesting job. Almost without exception, the connection between tasks and the flow of control must first be written on paper. After the pattern of connection and intertask control flow becomes clear, a PS program can be written without difficulty. The need for graphic aid is due to the nature of the PS language, which permits task interaction patterns to be specified. Graphic aid for the construction of PS programs would be an appropriate future improvement for the testbed.

Debugging a PS program is somewhat tricky. A PS program does not produce verifiable numeric output, nor does it generate tangible nonnumeric data. However, a PS program, through execution in a simulated system, produces a pattern of interactions among tasks and tasks' stochastic resource demands. Although both products are not immediately identifiable, they can be observed and checked based on the comprehensive traces generated during simulation. However, the interaction between tasks can be discerned more easily than the stochastic program behavior. In order to check the stochastic property, one must go to the trouble of collecting and analyzing traces, which contain the sample paths of tasks' resource demands.

6.4.2 Storage problem

The testbed may exhibit a storage explosion problem, since every task simulated in the testbed requires 1K bytes of dynamic storage[4], and every job requires 1/2 K bytes. Moreover, the number of event descriptors, queue element carriers, and message carriers grows in proportion to the number of tasks in a simulated system.

[4] Most of the 1K bytes dynamic storage of a task are devoted to buffering intertask communication messages.

Chapter 7
Conclusions and Future Research Directions

7.1 Conclusions

In the past decades, the form of computing has shifted from mainframes to networks of processors. For large real-time control applications, networks of processors promise high performance, high reliability and high extensibility. Since high performance is essential to real-time control, job scheduling is very important to distributed real-time system. However, most previous work on real-time scheduling is restricted to a workload of periodic jobs. There is a notable lack of real-time scheduling algorithms for distributed systems that support a dynamic workload. By proposing and profiling algorithms to fill this need, this dissertation systematically studies the problem of dynamic job scheduling under deadline constraints or priority requirements.

This work is a major departure from previous work in distributed resource management that focuses on improving average response time of jobs. Chapters 3 and 4 address the distributed deadline scheduling and distributed priority scheduling issues for sequential computations, while Chapter 5 examines the distributed priority scheduling issues for distributed computations. From another perspective, while Chapter 3 deals with deadline constraints, Chapter 4 and Chapter 5 investigate how to support global priority.

In Chapter 3, a deadline-driven *triage* rule for job selection and migration was devised to decide which job should emigrate or immigrate. Moreover, compared with pure load sharing algorithms that do not use the triage rule, deadline-driven distributed scheduling was found to improve the deadline miss ratio by more than 50% at medium and high loads. Algorithms based on either sender-initiated, or receiver-initiated approaches were evaluated. Based on the triage rule, the receiver-initiated approach was found to be superior at high loads, high communication overheads or tight deadline allowances.

Since the notion of priority implies an ordering of jobs contending for the resource, priority in distributed resource management implies a global ordering of all jobs in the system. Scheduling approaches that aim to support such ordering under different system constraints have been identified and investigated. A taxonomy of these approaches based on the type of resource, the organization of resource manager and the type of jobs was presented. This taxonomy is summarized as follows:

Priority Scheduling System	Multiplicity of Resource	Organization of Resource Manager	Job Composition
Simple Priority	Single	Centralized	Sequential
Global Priority	Multiple	Centralized	Sequential
Distributed Priority	Multiple	Multiple	Sequential
Composite Priority	Multiple	Centralized	Distributed

Table 7.1: Types of resource, resource manager, and jobs in priority scheduling

Table 7.1 displays priority scheduling systems and their attributes. These attributes are the key factors that differentiate priority scheduling systems. The attribute three-tuple includes type of resources (single or multiple), organization of resource managers (centralized or distributed), and type of computations (sequential or distributed).

Simple Priority systems are traditional, centralized priority scheduling systems.

Global Priority is a generalization of the *simple priority* scheduling principle from a central queue environment to a multiple physically separated queues. Under such a priority system, top N priority jobs are served in an N processor system except jobs in transit. However, such a scheduling rule incurs unacceptable overhead owing to extensive job transfer activities needed to maintain the global priority service order. Chapter 4 describes this global priority scheduling algorithm in detail.

Distributed Priority compromises between reducing the cost of scheduling and preserving global priority order of service. Multiple identical schedulers provide distributed control for resource allocation, by which global priority ordering is maintained through polling and negotiation among the schedulers. Since polling need not reach the entire system, and since each scheduler need not keep global state information up-to-date, the system can relax the global priority ordering requirements to reduce the cost of polling and negotiation. To explore the trade-offs in this compromise, three distributed priority scheduling algorithms, Symmetrical Polling (**SMP**), Priority Riding (**PRR**) and Status Broadcasting (**STB**) were proposed and evaluated in Chapter 4. These algorithms were found to improve response time for all priority classes, while at the same time closing the gap in service-level between priority classes. Among them, **PRR** results in the best response time for high priority jobs.

Composite Priority provides a basic infrastructure of priority systems for jobs made of parallel communicating tasks exhibiting large-grained parallelism. A Composite Priority system distinguishes between job priority and task priority. While job priority specifies the priority relative to other job, task priority specifies the priority relative to the other tasks of the same job.

In Chapter 5, composite-priority-based task placement strategies were proposed and evaluated for jobs having a pipeline synchronization pattern. With local priority scheduling, two load balancing strategies were evaluated. By balancing load for each priority class, the priority-based task mixing (**PBM**) was found to perform better than pure load balancing (**LBM**) for all jobs. Owing to the synchronized progress required for jobs, PBM is better in keeping tasks of the same pipeline proceed in parallel than LBM regardless of priority class. Moreover, three task assignment strategies based on knowledge of pipeline structure were proposed as follows: Random (**RA**), Task Clustering (**TC**) and Parallel-Task-Declustering (**PTD**). We focused on the startup and shutdown effect of a pipeline job, and found TC to be best when message cost is high (at least 10% of job's CPU demand) while PTD is best when message cost is low.

In Chapter 6, we describe the structure of the simulation testbed used by the performance study in Chapter 5. The key ingredient of this testbed is a synthetic workload generator for distributed computations. With the flexible **PS** (Parallel Structure) language, we were able to represent, specify and generate distributed workloads to study the interaction between resource allocation algorithms and parallel structures. A set of facilities necessary for modeling a message-based distributed computation exhibiting large-grained parallelism is and supported by the language.

7.2 Future Research Directions

Priority-Based Deadline Scheduling

An important objective for priority scheduling in soft real-time systems is to provide high priority jobs with better chance of meeting deadlines. Under such an objective, a high priority job has a higher probability of meeting its deadline than low priority jobs.

A global priority algorithm can addresses this issue by giving high priority jobs better service without using deadline information. However, it is well known that a deadline-oriented scheduling algorithm can result in a better deadline miss ratio, because it allows low priority jobs to meet their deadlines without hindering high priority jobs from finishing on time [Baker84]. Scheduling algorithms that meet both the priority requirements and the deadline demand must minimize a function of weighted deadline misses. We believe that such algorithms are very important for distributed real-time systems.

Task Assignment Heuristics for Other Parallel Structures

In Chapter 5, our study of task assignment strategies for the Composite Priority was limited to tasks having a pipeline parallel structure. However, the investigation of priority-based task placement can and must cover a wider range of parallel structures so that a full conclusion can be drawn regarding the sensitivity of algorithms to varying parallel structures. We believe that resource scheduling algorithms, especially in the task assignment stage, must be tailored for specific parallel structures. Interesting parallel structures include master-slave, precedence-graph, client-server and dining philosopher style resource sharing. However, with a flexible testbed facility, more complex parallel structures can be readily included.

The ultimate goal of studying parallel task structures is to provide an encapsulating scheduling facility for all kinds of distributed computations. This facility should combine specific scheduling knowledge of parallel structures with a flexible and adaptive distributed resource manager.

Task Group scheduling in Shared-Memory Systems

Highly parallel multi-processor systems have gained attention in recent years through the design of systems such as Sequent [Sequent84] and RP3 [Pfister86]. To sufficiently utilize the huge processing capacity available in such systems, it is necessary to allow more than one task group to share this resource. However, because these systems are aimed primarily at supporting single parallel computation, they have only rudimentary facilities for users to set up partitions. This simple facility is inadequate for production use, which requires automatic and flexible partitioning of systems to maximize job throughput. This problem of scheduling multiple parallel computations is most relevant to our study presented in Chapter 5. Among our discussions, the debate over partitioning vs. mixing is applicable for such systems. It appears that dynamic scheduling problems for parallel computations in multiprocessors will be one of the most exciting research areas in forthcoming years.

Bibliography

[Aczel60] Aczel, M. A. "The Effect of Introducing Priorities," *Operations Research*, 8, (1960), 730-33.

[Andrew83] Andrews G.R. and F.B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys*, 15, 1, (Mar. 1983), 3-44.

[Artsy84] Artsy, Yeshayahu, Hung-Yang Chang, and Raphael Finkel. "Charlotte: Design and Implementation of a Distributed Kernel," *Technical Report 554*, University of Wisconsin at Madison, Computer Sciences, (Aug. 1984).

[Artsy85] Artsy, Yeshayahu, Hung-Yang Chang, and Raphael Finkel. "Processes Migrate in Charlotte," *Technical Report 655*, Univ. of Wisconsin at Madison, Computer Sciences, (Aug. 1986).

[Artsy87] Artsy, Yeshayahu, Hung-Yang Chang, and Raphael Finkel. "Interprocess Communication in Charlotte," *IEEE Software*, 4, 1, (Jan. 1987), 22-28.

[Baker84] Baker, K.R.. "Sequencing Rules and Due-Date Assignments in A Job Shop," *Management Science*, 30, 9, (Sept. 1984), 1093-104.

[Barak85] Barak, A. B., and A. Shiloh. "A Distributed Load-balancing Policy for a Multicomputer," *Software —Practice and Experience*, 15, 9 (Sept. 1985), 901-13.

[Benn66] Benn, B. A. "Hierarchical Car Pool Systems in Railroad Transportation," Ph.D. Thesis, Case Institute of Technology, (1966).

[Bokhari81] Bokhari, S. H. "On the Mapping Problem," *IEEE Trans. on Computers*, C-30, 3, (Mar. 1981), 207-14.

[Bourn84] Bourne, D. A. "Autonomous Manufacturing: Automating the Job-Shop," *IEEE Computer*, (Sept. 1984), 76-86.

[Bryant81] Bryant, R.M. and R.A. Finkel. "A Stable Distributed Scheduling Algorithm," *Proc. 2nd Int. Conf. Distri-biuted Comput. Syst.*, (Apr. 1981), 314-23.

[Buzen83] Buzen, J.P., and A. B. Bondi. "Preemptive Resume in M/M/m," *Operation Research*, 31, 3, (May-June 1983), 456-65.

[Cabrera86] Cabrera, L.F. "The Influence of Workload on Load Balancing Strategies," *Proc. 1986 Summer USENIX Conference*, Atlanta, GA, (June 1986), 446-58.

[Carey86] Carey, M., and H. Lu. "Load Balancing in a Locally Distributed Database System," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, (May 1986).

[Carry76] Carry, M.R., and D.S. Johnson. "Scheduling Tasks with Uniform Deadlines on Two Processors," *Journal of ACM*, 23, 3, (July 1976), 461-67.

[Chang85] Chang, H.Y. and Miron Livny. "Priority in Distributed Systems," *Proceeding of IEEE 1985 Real Time Symposium*, (Dec. 1985), 123-32.

[Chen85] Chen, L. P. and V. O. Li. "An Optimal Algorithm for Processing Distributed Star Queries," *IEEE Trans. on Software Eng.*, SE-11, 10, (Oct. 1985), 1097-107.

[Cheng86] Cheng, S., J.A. Stankovic and K. Ramamritham. "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proceeding of IEEE 1986 Real Time Symposium*, (Dec. 1986), 166-74.

[Cheriton83] Cheriton, D. R. and W. Zwaenepoel. "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth Symposium on Operating Systems Principles*, (Dec. 1983), 110-19.

[Coffman76] Coffman, E.G. *Computer and Job-Shop Scheduling Theory*, NewYork: John Wiley & Sons, Inc, (1976).

[Craft83] Craft, D.H. "Resource Management in a Decentralized System," *Ninth ACM Symposium on Operating Systems Principles*, (Oct. 1983).

[Duffie82] Duffie, N. "An Approach to the Design of Distributed Machinery Control Systems," *IEEE Trans. Industrial Application*, (Aug. 1982).

[Eager84] Eager, D. L., E.D. Lazowska and J. Zahorjan. "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. on Software Eng.*, SE-12, 5, (May 1986), 662-675.

[Eager85] Eager, D. L., E.D. Lazowska and J. Zahorjan. "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load sharing", *Performance Evaluation Review*, 6, 1, (Mar. 1986), 53-68.

[Feijoo85] Feijoo, B. "Piecewise-linear Approximation Methods and Parallel Algorithms in Optimization," *Technical Report. 598*, University of Wisconsin-Madison Computer Sciences Department, (1985).

[Ferrari78] Ferrari, Domenico. "Computer Systems Performance Evaluation," Englewood Cliffs, New Jersey: Prentice Hall, (1978).

[Finkel85] Finkel, R.A. and U. Manber. "DIB — A Distributed Implementation of Backtracking," *ACM Trans. on Programming Lanuages and Systems*, (Apr. 1987), 235-56.

[Finkel86] Finkel, R. ed. "Experience with Crystal, Charlotte, and Lynx," *Technical Reports 630, 649 and 673*, University of Wisconsin-Madison Computer Sciences Department, (Feb., July and Nov. 1986).

[Fischer80] Fischer,C., D. Milton and S. Quiring. "Efficient LL(1) Error Correction and Recovery Using Only Insertions," *Acta Informatica*, 13, Fasc. 2, (1980), 141-54.

[Ford62] Ford, L.R. and D.R. Fulkerson. *Flows in Networks*, NJ:Princeton Univ. Press, 1962.

[Garey79] Garey, M. R. and D. S. Johnson. *Computers and Intractability--A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman and Company, 1979.

[Garey81] Garey, M. R. and D. S. Johnson. "Approximation Algorithms for Bin-Packing Problems — A Survey," in *Analysis and Design of Algorithms in Combinatorial Optimization*, Eds. G.

Ausiello and M. Lucertini. New York: Springer-Verlag, (1981), 147-72.

[Gylys76] Gylys, V.B. and J.A. Edwards. "Optimal Partitioning of Workload for Distributed Systems," *COMPCON*, (Fall 1976), 353-57.

[Indurkhya86] Indurkhya, B., H.S. Stone, Xi-cheng Li. "Optimal Partitioning for Randomly Generated Distributed Programs," *IEEE Trans. on Software Eng.*, SE-12, (Mar. 1986), 483-95.

[Jaiswal68] Jaiswal, N.K. *Priority Queues*, New York: Academic Press, 1968.

[Jackson60] Jackson, J. R. "Queues with Dynamic Priority Discipline," *Management Sci.*, 8, (1960), 18-34.

[Jensen85] Jensen, C.D. , C.D. Locke, and H. Tokuda. "A Time-Driven Scheduling Model for Real-Time Operating Systems," *IEEE 1985 Real-Time Systems Symposium*, (Dec. 1985), 112-22.

[Kleinrock67] Kleinrock, L. "Optimum Bribing for Queue Position," *Operations Research*, 15, (1967), 304-18.

[Kleinrock76] Kleinrock, L. *Queueing Systems*, , John Wiley & Sons, 1976.

[Kohler75] Kohler, Walter. "Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems," *IEEE Trans. on Computers*, C-24, (Dec. 1975), 1235-38.

[Krueger84] Krueger, Phil and R. Finkel. "An Adaptive Load Balancing Algorithm for a Multicomputer," *Technical Report 539*, University Wisconsin-Madison Computer Sciences Department, (Apr. 1984).

[Lavenberg83] Lavenberg, Stephen S. ed. *Computer Performance Modeling Handbook*, Academic press, 1983.

[Lawler77] Lawler, E. L. "A Pseudopolynomial Algorithm for Sequencing Jobs to Minimize Total Tardiness," *Ann. Discrete Math.*, 1, (1977), 331-42.

[Leinbaugh80] Leinbaugh, Dennis. "Guaranteed Response Times in a Hard Real-Time Environment," *IEEE Trans. on Software Eng.*, SE-6, 1, (Jan. 1980), 85-91.

[Leland86] Leland, W. and T. Ott. "Load-balancing Heuristics and Process Behavior," *Proc. Performance '86 and ACM SIGMETRICS Conf on Measurement and Modeling of Computer Systems*, (May 1986), 54-69.

[Lelann81] Lelann, Gerard. "Motivatins, Objectives and Characterization of Distributes Systems," *Distributed Systems, Architecture and Implementation: An Advanced Course*, Eds. B.W. Lampson, M. Paul, and H.J. Siegert, (1981).

[Lesser83] Lesser, V. and D.D. Corkill. "The Distributed Vehicle Monitoring Testbed: A Tool For Investigating Distributed Problem Solving Networks," *The AI Magazine*, (Fall 1983), 15-33.

[Liu73] Liu, C.L. and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of ACM*, 20, 1, (Jan. 1973).

[Livny82] Livny, Miron and Myron Melman. "Load Balancing in Homogeneous Broadcast Distributed

Systems," *ACM Computer Network Performance Symposium*, (Apr. 1982), 47-55.

[Livny84] Livny, Miron. "The Study of Load Balancing Algorithms For Decentralized Distributed Processing Systems," *Tech. Report 570* University of Wisconsin-Madison Computer Sciences Department, (1984).

[Livny85] Livny, M. and U. Manber. "Distributed Computation Via Active Messages," *IEEE Trans. on Software Eng.*, SE-11, 12, (Dec. 1985).

[Livny86] Livny, Miron. "DENET User's Guide" (in preparation) University of Wisconsin-Madison Computer Sciences Department, (Sept. 1987).

[Ma82] Ma, R. P., E. Lee and M. Tsuchiya. "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, C-31, 1, (Jan. 1982).

[Manacher67] Manacher, J. "Production and Stabilization of Real-Time Task Schedules," *Journal of ACM*, 14, 3, (July, 1967), 439-65.

[Manber87] Manber, U., N. Alon and A. Barak. "On Disseminating Information Reliably without Broadcasting," to appear in the *The 7th International Conference on Distributed Computing Systems*, Berlin, (Sept. 1987).

[Martel82] Martel, C. "Preemptive Scheduling with Release Times, Deadlines, and Due Times," *Journal of ACM*, 29, 3, (July 1982), 812-29.

[Miller85] Miller, B.P. "Performance Characterization of Distributed Programs," Ph.D. Thesis, EECS U.C. Berkeley, (Jan. 1985).

[Mok78] Mok, A.K. and R. Dertouzos. "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Seventh Texas Conf. Computing Systems*, 1978.

[Mok83] Mok, A.K. "Fundamental Design Problems of Distributed Systems for the Hard Real-time Environment," Ph.D. Thesis, Massachusetts Institute of Technology, 1983.

[Moor69] Moore, J.M. "Sequencing *n* Jobs on One Machine to Minimize the Number of Tardy Jobs," *Management Sci.*, 15, (1968), 102-09.

[Muntz70] Muntz R.R. and E.G. Coffman. "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *Journal of ACM*, 17, 2, (Apr. 1970),324-38.

[Nelson86] Nelson, R., D. Towsley and A.N. Tantawi. "Performance Analysis of Parallel Processing Systems," *Performance Evaluation Review*, 15, 1, (May 1987).

[Ni85] Ni, L., C. Xu, and H. Gendreau. "A Distributed Draft Algorithm for Load Balancing," *IEEE Trans. on Software Eng.*, SE-11, 10, (Oct. 1985), 1153-61.

[Ousterhout80] Ousterhout, John. "Partitioning and Cooperation in a Distribution Multiprocessor Operating System: Medusa," Ph.D. Thesis, Department of Computer Science, (Apr. 1980).

[Peterson81] Peterson, J. L. *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, New Jersey:Prentice-Hall, 1981.

[Pfister85] Pfister, G.F., W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton and J. Weiss. "The IBM Research Parallel Processor Prototype

(RP3): Introduction and Architecture" *Proc. 1985 International Conference on Parallel Processing,* IEEE Computer Society press, (1985), 772-81.

[Popek81] Popek G., B. Walker, J. Chow, D. Edwards and C. Kline. Rudisin G., and Thiel G., "LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles,* (Dec. 1981).

[Powell83] Powell, M. L. and B.P. Miller. "Process Migration in Demos/MP," *Proceeding of the Ninth Symposium on Operating Systems Principles,* (Dec. 1983), 110-19.

[Quirk83] Quirk, W.J. "Recent Developments in the SPECK Specification System," *HARWELL Report CSS.146,* (1983).

[Ramam84] Ramamritham, K. and Jone Stankovic. "Dynamic Task Scheduling in Distributed Hard Real-Time Systems," *IEEE. Software,* 1, 4, (May 1984), 65-74.

[Rao79] Rao, G.S., H.S. Stone and T.C. Hu. "Assignment of Tasks in a Distributed Processor System with Limited Memory," *IEEE Trans. on Computers,* C-28, (Apr. 1979), 291-99.

[Ritchie74] Ritchie, D. "The Unix Time-Sharing System.," *Communication ACM,* 17, 7, (July 1974), 365-75.

[Sauer82] Sauer, C. and K. M. Chandy. *Computer Systems Performance Modeling,* NJ:Prentice Hall, 1981.

[Schoeffler84] Schoeffler, James D. "Distributed Computer Systems for Industrial Process Control," *IEEE Computer,* 17, (Feb. 1984), 11-19.

[Sequent84] Sequent Computer Systems. *Balance 8000 System Technical Summary,* Oregon: Sequent Computer Systems, (Dec. 1984).

[Sha87] Sha, L. "Solutions For Some Practical Problems in Prioritized Preemptive Scheduling," *Proceeding of IEEE 1986 Real Time Symposium,* (Dec. 1986), 182-93.

[Shwan87] Shwan, K, P., Gpinath and W. Bo. "CHAOS--Kernel Support for Objects in the Real-Time Domain" *IEEE Trans. on Computers,* C-36, 8, (Aug. 1987).

[Smith80] Smith, Reid G. "Control in a Distributed Problem Solver," *IEEE Trans. on Computers,* C-29, (Dec. 1980).

[Spector85] Spector, Alfred and David Gifford. "Case Study: The Space Shuttle Primary Computer System," *Communication ACM,* 27, 9, (Sept. 1984), 872-901.

[Stankovic84] Stankovic, J.A. "Simulations of Three Adaptive, Decentralized Controlled, Task Scheduling Algorithms," *Computer Networks,* 8, 3, (June 1984), 199-217.

[Stankovic85] Stankovic, J.A. "Stability and Distributed Scheduling Algorithms," *IEEE Trans. on Software Eng.,* SE-11, 2, (Apr. 1985), 445-65.

[Stankovic81] Stankovic, John and I. S. Sidhu. "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Group," *Proc. 1981 Int. Conf. Parallel Processing,* (Aug. 1981).

[Stone79] Stone, H.S. "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Eng.,* SE-3, 1, (Jan. 1977), 85-93.

[Teixeira78] Teixeira, T. "Static Pirority Interrupt Scheduling," *Proc. Seventh Texas Conf. Computing Systems*, (1978), 5:11-13.

[Tilborg84] Tilborg, Andrew. "Wave Scheduling — Decentralized Scheduling for Task Forces in Multi-cmputers," *IEEE Trans. on Computers*, C-33, 9, (Sept. 1984), 835-44.

[Uhr85] K. Preston and L. Uhr eds. *Multicomputers and Imange Processing Algorithms and Programs*, New York: Academic Press, 1982.

[Wah85] Wah, B.W. and J.Y. Juang. "Resource Scheduling for Local Computer Systems with a Multiaccess Network," *IEEE Trans. on Software Eng.*, SE-11, 12, (Dec. 1985), 1144-57.

[Walker83] Walker B., G. Popek, R. English, C. Kline and G. Thiel. "The LOCUS Distributed Operating System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, (Oct. 1983).

[Wang84] Wang, Yung-Terng and R. Morris. "Load Sharing in Distributed System," *IEEE Trans. on Computers*, C-33, 3, (Mar. 1984).

[Watson81] Watson, Richard. "Distributed System Architecture Model," in *Distributed Systems, Architecture and Implementatin: An Advanced Course*, Ed. Lampson, B.W., Paul, M. and Siegert, H.J., (1981).

[William84] Williams, Elizabeth. "Processor Queueing Disciplines In Distributed Systems," *ACM Distributed computing Conference*, (1984), 113-19.

[Wirth82] Wirth, Niklaus. *Programming in Modula-2*, Springer-Verlag, New York, 1982.

[Zeigler84] Zeigler, Bernard P. *Multifacetted Modelling and Discrete Event Simulation*, New York: Academic Press, 1984.

[Zhao85] Zhao, W and K. Ramamritham. "Distributed Scheduling Using Bidding and Focused Addressing," *Proceeding of IEEE 1985 Real-time symposium*, (Dec. 1985), 103-11.

[Zhao86] Zhao, W and K. Ramamritham. "Preemptive Task Scheduling under Multiple Resource Constraints," *IEEE Trans. on Computer*, C-36, 7, (July 1987).

[Zhou86] Zhou, S. "A Trace-Driven Simulation Study of Dynamic Load Balancing," *Technical Report UCB/CSD 87/305*, (Sept. 1986).

[Zhou87] Zhou, S. "An Experimental Assessment of Resource Queue Length as Load Indices," *Proc. Winter USENIX Conference*, Washington, D.C., (Jan. 1987), 73-82.