DISTRIBUTED SCHEDULING FOR A CHANGING ENVIRONMENT

Phillip E. Krueger

Computer Sciences Technical Report #780

June 1988

DISTRIBUTED SCHEDULING FOR A CHANGING ENVIRONMENT

by

PHILLIP E. KRUEGER

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

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1988

ABSTRACT

Scheduling for distributed computing systems is significantly more complex than that for single-processor systems. In addition to allocating the resources local to a node among the processes residing at that node, a distributed scheduler controls *which* processes reside at each node. This latter element of distributed scheduling is referred to as *load distributing*.

The activities required for load distributing, most notably transferring processes between nodes, may carry substantial resource overhead. Consequently, the performance benefits that can potentially be achieved through these actions must be weighed against the performance costs that arise as a result of their overhead. In some instances, the cost of an action may exceed its benefit. If many of these *ineffective* actions are undertaken, overall performance is degraded and the stability of the system is threatened. Therefore, the *degree* of load distributing that is strived for must be carefully chosen, with the goal of performing as many effective actions as possible, while avoiding those that are ineffective.

Unfortunately, as we show, the degree of load distributing that provides the best performance, and the range in degree that maintains system stability, are dependent on characteristics of the system and its workload that may change over time. To be robust over the wide range of system environments that may occur over periods of minutes, hours or days, a load distributing algorithm must adapt in degree. The focus of this dissertation is on the design of such *adaptive* load distributing algorithms. Recognizing the diversity of performance objectives and resources available to load distributing algorithms, no attempt is made to design a single algorithm that is in some sense 'general-purpose'. Instead, a framework is developed for adding adaptivity to existing algorithms.

As examples, this framework is applied to dissimilar load distributing algorithms with differing performance objectives. Using simulation, the resulting adaptive algorithms are found to improve performance and maintain system stability over a substantially broader range of system environments.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my advisor, Professor Miron Livny, for his extensive advice and encouragement throughout my graduate studies. I am also especially grateful to Professor Mary Vernon, who taught me a great deal about performance analysis. My thanks to Professor Michael Carey, who provided many useful suggestions on the presentation of this dissertation. I would also like to thank the remaining members of my Examination Committee, Professors Marvin Solomon and Douglas Bates, for their constructive criticism.

Of course, I am deeply grateful to all of my family, particularly my wife, Mary, and my daughters, Samantha and Amanda, for their support through many difficult times.

This research was supported in part by NSF grant MCS-8105904 and by a Digital Equipment Corporation External Research Grant.

Much of the material in chapters 4 and 5 has appeared in [Krueg87], [Krueg87a] and [Krueg88].

CONTENTS

CHAPTER 1

Introduction

1.1. Background

Distributed computing systems have received much attention over the past decade, primarily due to three advantages they have over single-processor systems. First, distributed systems have the potential for increased reliability. A system missing some components, possibly as a result of preventive maintenance or failure, may retain its ability to perform the same functions as the full system, but at a slower rate. The second advantage is actually another perspective on the first: Distributed systems have the potential to be incrementally upgraded. A system may be made more powerful simply by adding new components, without replacing any existing hardware. Finally, distributed systems can potentially deliver service equivalent to that of a given single-processor system at a fraction of the cost. The focus of this thesis is on achieving this last advantage by gaining access to the processing capacity available in distributed systems.

The central issue in tapping the potential of a distributed system is scheduling. The best performance can not be attained unless the resource multiplicity inherent in a distributed system is used to its fullest advantage. A scheduler capable of exploiting this resource multiplicity, however, is considerably more complex than that required for a single-processor system. Because of this complexity, scheduling remains among the most pressing open problems in the design of distributed systems.

1.2. The Problem

In recent years, several studies, including [Livny82, Eager86], have shown *load distributing* to be an essential element of scheduling for distributed systems. The responsibility of this scheduling component is to distribute the system workload among the nodes through *process transfer*. Determining how the local resources at a node are allocated among its resident processes is then left to the *local scheduling* component.

The goal of load distributing is to improve performance by correcting anomalies that arise in the distribution of load among nodes. However, the performance *benefits* that can be gained through this redistribution of the workload must be weighed against the performance *costs* of load distributing actions. Load distributing costs arise as a result of its use of resources. This use may increase contention, increasing the queue delays experienced by user processes sharing these resources and decreasing the responsiveness of other instances of the distributed scheduler. If the cost of a load distributing action is greater than its benefit, it is not *cost-effective*, and performance is harmed. Furthermore, if a large portion of load distributing actions are not cost-effective, overall performance is degraded, rather than improved, and the stability of the system is threatened. Conversely, if actions that would have been cost-effective are not performed, the potential of the distributed system is not fully exploited. The *degree* of load distributing that is strived for must be carefully chosen, with the goal of performing as many cost-effective actions as possible, while avoiding those that are ineffective.

The degree of load distributing that is most cost-effective, providing the best performance, is dependent on characteristics of the system workload. Eager, Lazowska and Zahorjan note [Eager86], for example, that when the system is heavily utilized, foregoing process transfers between nodes that have little difference in load may be advantageous. The use of scarce resources to pursue such transfers is not cost-effective under these conditions. When the system is lightly utilized, however, the resources required for these transfers are in plentiful supply, and such transfers may become cost-effective.

In addition to system utilization, we identify several workload characteristics and characteristics of the system itself that take part in determining the degree of load distributing that is most cost-effective. These characteristics include the utilization of the communication device, the distributions of process sizes and service demands, the number of nodes participating in the distributed system and the subset of nodes that are actively generating work for the system. Unfortunately, each of these characteristics may be subject to change over

time. As a result, an algorithm that improves performance under the conditions that exist at some particular moment may *degrade* performance at a later time, possibly threatening the stability of the system. To be effective in a changing system environment, an algorithm must adapt its degree of load distributing to such changes.

The focus of this dissertation is on a problem that has not previously been addressed: the design of distributed scheduling algorithms that adapt to changes in the system environment. The goal of such algorithms is to be robust across the wide range of conditions that can occur in a distributed system over periods of minutes, hours or days. Recognizing the diversity of objectives and resources available to load distributing algorithms, no attempt is made to design a single algorithm that is in some sense 'general-purpose'. Instead, we develop a framework for adding adaptivity to existing algorithms. This framework is applicable across a broad range of algorithms having a variety of performance objectives.

1.3. Thesis Organization and Overview

This dissertation begins with an overview of related work in chapter 2. A framework for the thesis, contained in chapter 3, describes the distributed system model that is assumed, provides a taxonomy of load distributing actions, and motivates a collection of indices with which performance can meaningfully be evaluated.

Chapter 4 lays a foundation for this study. A scheduling algorithm suitable for a given distributed system can not be designed until, first, the *performance objective* that motivates the algorithm has been defined and, second, a *scheduling policy* that has the potential to meet this objective has been identified. We define this potential to be the performance achieved in the limiting case when scheduling overhead is negligible. We explore the sorts of performance improvements that are potentially available through distributed scheduling, as well as the types of distributed schedulers that are most applicable to achieving particular performance objectives. This chapter provides a unique perspective on the interaction between load distributing and local scheduling in meeting performance objectives.

Since, in practical distributed systems, scheduling overhead is not negligible, the potential of a distributed scheduling policy can not be fully attained. Under the burden of non-negligible overhead, the cost-effectiveness of distributed scheduling actions must be considered. Recognizing that the largest portion of distributed scheduling overhead arises from load distributing, chapter 5 studies the extent to which a load distributing algorithm should make compromises in pursuing its chosen policy, reducing its degree of load distributing in order to maintain cost-effectiveness. This chapter explores the role of system and workload conditions in determining the degree of load distributing that provides the best performance and, more important, the range in degree that improves, rather than degrades, performance. Since, for many distributed systems, several of these system and workload characteristics are likely to vary widely over time, we conclude that a load distributing algorithm having a 'static degree' is not sufficient for such systems. To maintain cost-effectiveness over the wide range of conditions that can occur in a distributed system over periods of minutes, hours or days, a load distributing algorithm must adapt to the system environment.

To accomplish this goal, chapter 6 addresses the design of adaptive load distributing algorithms. A framework is developed for extending a load distributing algorithm to be adaptive to changes in the environment. As examples of how this framework might be applied, it is used to extend two load distributing algorithms having dissimilar designs and differing performance objectives. The resulting adaptive algorithms are able to improve performance over a substantially broader range of system and workload conditions. In addition, under conditions in which performance can not be improved, these algorithms pose a significantly reduced threat to system stability. From these examples, we conclude that the framework is both simple enough to be easily applied, yet general enough to be applicable to diverse load distributing algorithms having a variety of performance objectives.

Finally, we summarize our results and suggest directions for future research in chapter 7.

CHAPTER 2

Survey of Related Work

Distributed scheduling has captured a good deal of interest over the past several years, with much work appearing in the literature. Distributed scheduling algorithms can be broadly categorized as *static* or *dynamic*. While dynamic scheduling algorithms base scheduling decisions, at least in part, on recent system state information, static algorithms make no use of such dynamic information. Because dynamic scheduling is more applicable to general-purpose distributed systems than static scheduling, we provide only a brief overview of static scheduling, followed by a more detailed survey of dynamic scheduling.

2.1. Static Scheduling for Distributed Systems

Static scheduling algorithms for distributed systems can be divided into three classes: optimal, heuristic and optimal probabilistic algorithms. Optimal scheduling algorithms depend on knowledge of the future resource requirements of all processes to identify a schedule that minimizes some cost function. Optimal algorithms that have been studied can be roughly categorized as being based on either graph theory [Bokha79, Bokha81, Chou82, Jenny77, Rao79, Shen85, Stone77, Stone78, Stone78a] or mathematical programming [Blaze86, Chu69, Chu80, Gylys76, Lee77, Lee77a]. Unfortunately, an optimal schedule can generally not be found in real time, as Gursky [Gursk] has shown that the problem of finding an optimal assignment for four or more processors is NP-complete. As an alternative, heuristic algorithms [Balan76] [Edwar75, Efe82, Garey78, Gylys76, Ohtsu75] are potentially more efficient, but identify schedules that may be suboptimal. This increase in efficiency may not be enough, however, as Krishnaprasad and Price [Krish81] have found that the simplified problems addressed by several of these heuristic algorithms are NP-complete, as well.

Optimal probabilistic algorithms [Ni85, Tanta85] require less information than optimal or heuristic algorithms. Instead of requiring information on individual processes, only aggregate information is necessary. Specifically, only the arrival rate of processes to the system and the process service rate at each node must be known. Based on this information, such algorithms find the optimal arrival rate for each node. However, transferring processes to maintain these optimal rates does not guarantee that individual transfers are optimal. Consequently, performance is generally less than optimal. Optimal probabilistic algorithms shed little light on the scheduling problem for homogeneous distributed systems. For such systems, the optimal arrival rate is shown to be the same for each node: the overall arrival rate divided by the number of nodes. For systems in which process arrival rates are already homogeneous, optimal probabilistic algorithms provide no improvement in performance.

2.2. Dynamic Scheduling for Distributed Systems

Instead of requiring knowledge of future resource use, dynamic scheduling algorithms make scheduling decisions based on system state information. Dynamic algorithms are more broadly applicable than static algorithms, since recent state information is generally more accessible than information on future resource requirements. For example, dynamic algorithms are applicable to general-purpose homogeneous distributed systems, while static algorithms are generally not. Because local scheduling was previously well-studied, most research on dynamic scheduling for distributed systems has focused on the load distributing problem.

2.2.1. Load Distributing in Homogeneous Systems

When either the processing bandwidths of nodes or their process arrival rates are heterogeneous, it is intuitively reasonable that performance can be improved by redistributing the system load among nodes. Some scheduling algorithms, such as optimal probabilistic algorithms, are solely directed at such systems. However, Livny and Melman [Livny82] show that performance can potentially be improved even when arrival rates and

processing bandwidths are homogeneous. When some node is idle while processes wait for service at other nodes, the processing power of the system is reduced and improvement in performance is possible. Livny and Melman refer to this state as 'wait while idle' (WI) and show that, for a system having exponentially distributed process arrival rates and service demands, the probability of its occurrence (P_{WI}) is determined by the system load and the number of nodes participating in the system. Significantly, for a distributed system composed of 20 nodes and having a system load between 0.33 and 0.89, the probability of a WI state is greater than 0.9. Thus, at typical system loads, there is nearly always potential for improvement in performance, even when nodes and process arrival rates are homogeneous.

Based on a model in which process transfers do not require the use of resources, Livny [Livny83] finds a lower bound for the transfer rate necessary to avoid the occurrence of WI states in homogeneous systems. This bound is dependent on process arrival and service rates, and on the number of nodes composing the system. To avoid WI states in a system having at least 10 nodes, a large portion of the processes must transfer. For some workloads, this transfer rate can be expected to consume much of the bandwidths of the resources used for load distributing.

2.2.2. A Taxonomy of Dynamic Load Distributing Algorithms

2.2.2.1. Preemptive and Non-Preemptive Algorithms

Dynamic load distributing algorithms can be divided into those that rely solely on non-preemptive process transfers and those that make use of preemptive transfers as well. Non-preemptive algorithms, such as [Chou86, Farbe73, Smith80, Zatti85], are concerned only with the initial placement of a process at the time it begins execution. No processes are transferred after execution has begun. As a result, if process completions or process resumptions after being blocked for I/O cause the distribution of load among nodes to become less than optimal, no correction can be made until new processes initiate. For example, if some node is servicing several processes, but all the processes at some other node have completed, no use can be made of the processing capacity of this now-idle node until new processes arrive. Processes must continue waiting for service at the overloaded node, rather than take advantage of the capacity of the idle node. As a second example, if several processes at a node resume execution after having been blocked for I/O, none may transfer to other nodes, even though some nodes may lie idle.

Preemptive load distributing algorithms, such as [Bryan81, Krueg84, Livny82], are able to adapt to changes in the system state with finer granularity than non-preemptive algorithms. If, at any time, the location of a process becomes less than optimal, it can be transferred to a better node. As a result, preemptive algorithms are better able to exploit the processing capacity of a distributed system than non-preemptive algorithms. The consequent improvement in performance was studied by Eager, Lazowska and Zahorjan [Eager88], and is addressed in chapter 5 of this dissertation.

Making use of a 'no-arrivals' analytic model, Eager, et al. find that there are no conditions under which preemptive transfers result in *major* performance benefits. Studying a more detailed model through simulation and taking a broader perspective on performance, however, we show in chapter 5 that substantial performance gains can be achieved through preemptive transfers.

2.2.2.2. Centralization

Dynamic load distributing algorithms differ in their degree of centralization. Algorithms may be centralized, hierarchical, fully decentralized, or some combination of these. Potentially, algorithms having centralized components, such as [Farbe73, Stank85], are less reliable than fully decentralized algorithms, since the failure of a central component may cause the entire system to fail. A solution to this problem is to allow a redundant central component to become active when one fails, as in [Stank85]. A second weakness of centralized algorithms is not so easily cured: A central component is potentially a bottleneck, limiting the amount of load distributing activity in the system. While hierarchical algorithms, such as proposed by Van Tilborg and Wittie [Tilbo84], alleviate both of these problems, the complete solution lies in fully decentralized algorithms, such as [Bryan81, Krueg84, Livny82].

2.2.2.3. Amount of State Information

Another dimension in which dynamic load distributing algorithms vary is the amount of state information used in making load distributing decisions. Algorithms may require knowledge, possibly inaccurate, of the states of all nodes, a subset of nodes or local state information only. Eager, Lazowska and Zahorjan [Eager86] consider how much state information is sufficient to make good load distributing decisions. To simplify analytic study, a distributed system is modeled as containing an infinite number of identical nodes with identical workloads. In such a system, each node is stochastically independent, allowing each to be analyzed in isolation. Using this model, this study shows that mean process response time can be significantly reduced by using only local state information and choosing a transfer partner randomly. An additional substantial reduction in mean response time can be achieved by examining the states of a limited number of randomly chosen nodes, selecting as a transfer partner the first to meet a set criterion. However, only slight improvement beyond this is provided by examining the states of all nodes and choosing the *best* node meeting the criterion.

2.2.2.4. Sender, Receiver and Symmetrically-Initiated Algorithms

Many load distributing actions can be categorized according to the goals of the nodes that initiate them. An action is referred to as *sender-initiated* if it is originated by a node that is the potential source of a process transfer. For example, a negotiation session is sender-initiated if it is begun by a node seeking to transfer a process to another node. Conversely, a *receiver-initiated* action is originated by a node seeking a process that can be transferred to it. At the extremes, a sender-initiated load distributing algorithm, such as [Bryan81], allows *only* potential senders to initiate actions, and a receiver-initiated algorithm, such as [Livny82], allows actions to be initiated only by potential receivers. A symmetrically-initiated algorithm, such as [Krueg84], makes use of both sender-initiated and receiver-initiated actions.

Eager, Lazowska and Zahorjan compare preemptive sender-initiated with receiver-initiated load distributing algorithms [Eager86a]. As in [Eager86], modeling a distributed system as having an infinite number of identical nodes with identical workloads allows each node to be analyzed in isolation. Using this model, neither strategy is found to be best for all workload characteristics. At low system loads, sender-initiated algorithms have an advantage. For example, while a node that becomes overloaded has little difficulty finding an idle node to accept a transferred process under such conditions, a node that becomes idle may have great difficulty finding a node with an overabundance of processes. As a result, unless every node is examined, an idle node may remain idle even though some node is overloaded. In contrast, at high system loads, receiver-initiated algorithms have the advantage. For example, while a node that becomes idle can easily find a node that would like to divest itself of some processes under such a workload, a node that becomes overloaded may be unable to find an idle node. Again, unless every node is examined, an overloaded node may remain overloaded even though some node lies idle. Using simulation, Chang [Chang86] finds similar trends for a load distributing algorithm targeted for a real-time system. Whether either of these extremes has an advantage over symmetrically-initiated load distributing is unknown, however, since neither study compares these extremes with symmetrically-initiated algorithms. We address this problem in chapter 5.

2.2.2.5. Gathering of State Information

Dynamic load distributing algorithms can be categorized according to the strategy used for gathering system state information. Some algorithms that have been studied, such as [Livny82, Stank84, Zatti85], maintain tables of state information. When a load distributing decision must be made, these tables are consulted. Other algorithms, such as [Bryan81, Livny82, Krueg84], use state information that is gathered at the time a transfer decision must be made, disregarding old information. Hybrids of these schemes are also possible. For example, in [Krueg84], information that has previously been gathered is used in making a transfer decision if it is recent, and new information is gathered if it is not.

Algorithms that Maintain State Information

Among algorithms that maintain state tables, some, such as those studied by Stankovic [Stank84], require each node to periodically issue an update message containing its current state. Since the period is fixed, the accuracy of the state information varies with workload characteristics, particularly with process arrival and service rates. As the arrival rate increases, the number of transfers increases, but the *quality* of transfers decreases.

For some arrival and service rates, the state information is not accurate enough to be useful, and transfers are essentially random. A second problem occurs if the number of nodes participating in the system varies. The fixed update period may result in heavy resource use when the number of nodes is large.

In a later paper, Stankovic [Stank85] proposes a preemptive algorithm, containing centralized components, that adapts to the effect that changing workload characteristics have on the accuracy of state information. Decreased accuracy in state information results in more conservative transfers, decreasing the transfer rate. This is an unfortunate consequence, since the workload characteristics that decrease accuracy, such as increased arrival rates, require at least as high a transfer rate, if not higher, to significantly improve performance.

Through clever exchange of state information, Barak and Shiloh [Barak85] are able to reduce the number of messages required to attain a given level of accuracy for a given set of workload conditions. Their preemptive algorithm requires that no nodes leave or rejoin the system; the number of nodes must be fixed. Alternatively, Zatti [Zatti85] proposes a non-preemptive algorithm that reduces the number of update messages by sending them only when the state of the node has changed within the update period.

Chou and Abraham [Chou86] analyze a non-preemptive algorithm that maintains previous as well as current state information at each node. Using linear root-mean square estimation, the state of each node at the end of the next update period can then be predicted. This algorithm is sensitive to the fixed number of past observations used in the estimation. Too few observations contain too little information to result in accurate predictions, while too many observations allow irrelevant information to be used. The optimal number of observations depends on changing workload characteristics.

Instead of relying on periodic updates, load distributing algorithms, such as Livny and Melman's preemptive BST algorithm [Livny82] and the Distributed Drafting Algorithm of Ni, Xu and Gendreau [Ni85a], update state tables only when the system state changes. These algorithms have an advantage over those using periodic updates, since updates occur at appropriate times, allowing better accuracy in information. In addition, the rate of updates is never greater than necessary to support the transfer rate. However, no limit is placed on the update or transfer rate. As a result, for some workload conditions or numbers of nodes participating in the distributed system, the utilization of resources for load distributing may be high, resulting in long queues.

Algorithms that Gather State Information as Needed

Rather than maintain state tables, the remaining algorithms gather information only when it is necessary to make a load distributing decision. The earliest load distributing algorithms proposed were of this type.

The original load distributing algorithm, a non-preemptive algorithm based on bidding, was proposed by Farber [Farbe73] for the Distributed Computing System. In this system, each resource is associated with a process. To initiate a process, a user sends a broadcast message to all resource allocators, each of which reply with a bid to perform the requested service. The requester selects the best bid according to its criteria and sends an acceptance message to the corresponding resource allocator. A weakness in this algorithm is that, when arrival rates are relatively fast or the number of nodes is large, the heavy use of resources required for bidding may degrade performance more than it is improved by distributing the load.

Smith presents a similar, though more detailed, non-preemptive algorithm [Smith80]. In his Contract Net Protocol, the overhead of bidding is reduced by requiring only idle nodes to submit bids. However, as in the Distributed Computing System, the resource use of load distributing is bounded only by resource bandwidths, allowing long queues to develop.

Livny and Melman's BID and PID algorithms [Livny82] are similar to Smith's algorithm, though they are preemptive and receiver-initiated. When a node becomes idle, it broadcasts its state (BID) or polls a random subset of nodes (PID) in search of a suitable transfer partner. Similar algorithms in which a node that becomes overloaded polls a subset of nodes in search of a transfer partner are found in [Bryan81, Eager86, Eager86a]. Again, these algorithms may cause long queues to develop, as the resource use of load distributing is bounded only by resource bandwidths.

In the Above-Average algorithm of Krueger and Finkel [Krueg84], a node that becomes overloaded broadcasts its state and waits for a response from an underloaded node. However, as the utilization of the communication device increases, causing transfers to become more expensive, the criteria defining overloaded and

underloaded nodes become more strict, allowing only the most advantageous transfers to occur. Thus, this preemptive algorithm places a reasonable bound on the communication device bandwidth used for load distributing.

CHAPTER 3

Framework

3.1. Distributed System Model

The models used in this study are closely related to the m*(M/M/1) family of distributed system models proposed by Livny [Livny82, Livny83]. Though these models are quite general, they have been augmented for this study to allow hyperexponentially distributed task service demands and to allow the processor queuing discipline, or local scheduling discipline, to be specified as a parameter. Hyperexponential distributions are important to consider, as those task service demand distributions that have been observed [Rosin65, Ander74, Trive82, Lelan86, Zhou86] are poorly approximated by exponential distributions. Predictably, we refer to these models as m*(M/H/1). Figure 3.1 illustrates a system of this type, which consists of m functionally identical nodes, fully connected by a communication device. Tasks arrive independently at each node and join the local queue. After arriving, a load distributing algorithm allows process transfers. The distribution of interarrival times is exponential, so the external task arrival process of the system consists of m independent Poisson processes.

For this study, we assume a token ring communication device. Processes are assumed to execute independently, with no intercommunication. Furthermore, we assume that nodes have equal processing bandwidths and that the service demands of processes arriving at different nodes are identically distributed. However, the *rates* at which processes initially arrive (as opposed to arriving as the result of transfer) may be

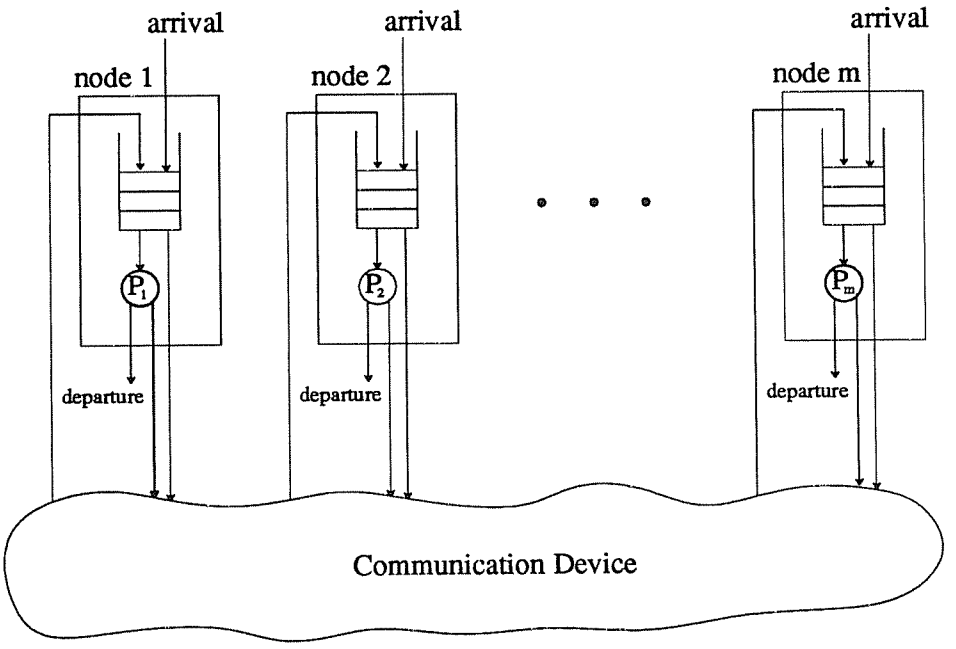

Figure 3.1 An m*(M/H/1) system

different at different nodes. We refer to such a workload as having *heterogeneous initiation rates*. Such workloads may be common for some types of distributed systems, particularly those composed of workstations [Mutka87]. Since we are interested in scheduling for general-purpose computer systems, we assume that the scheduler has no deterministic a priori information about process service demands. In addition, we assume that processes do not leave the system before completing service.

A special case of the m*(M/H/1) model is the M/H/m-like system [Livny83]. Such a system simplifies analysis by making the assumption that no work is associated with scheduling: both context switches and process transfers are instantaneous and without cost. The M/H/m-like system is the distributed-queue analogue of the M/H/m queue, from which it derives its name. An M/H/m-like system is *work-conserving* [Klein76] if its distributed scheduling policy does not allow a server to lie idle while processes wait for service. To meet this goal, the local scheduling component must not allow its server to lie idle while processes wait in the queue, and the load distributing component must not allow nodes to lie idle while processes wait for service at other nodes. A resulting distributed scheduling policy, together with the above assumption, assures that "no work (service requirement) is created or destroyed within the system" [Klein76]. Work-conserving systems are convenient analytically because, when local scheduling is non-preemptive and process service demands are exponentially distributed, the mean process response time ($\overline{RT}$) and the mean number of processes residing in the system ($\overline{N}$)[1] are [Laven83]:

$$\overline{RT} = \overline{X}\,\overline{N} / m\rho \qquad \text{for } \rho < 1 \tag{3.1}$$

$$\overline{N} = m\rho + \frac{\rho(m\rho)^m}{m!A(1-\rho)^2}$$

where:

$$A = \sum_{i=0}^{m-1} \frac{(m\rho)^i}{i!} + \frac{(m\rho)^m}{m!(1-\rho)}$$

regardless of which non-preemptive discipline is chosen. The above equations also hold for the preemptive disciplines studied in this thesis. For Processor Sharing [Klein67], in particular, these equations hold regardless of the distribution of process service demands. Kleinrock sums up the lack of discriminatory power inherent in $\overline{RT}$ ([Klein76] page 171): "One may therefore conclude that the average response time by itself is not a very good indicator of system performance." In the next section, we identify a set of performance indices that, together, are better able to discern differences in performance.

While much of this study is centered on the more realistic m*(M/H/1) model, the M/H/m-like system is used in some analyses to establish bounds.

3.2. Performance

Since the central issue of scheduling is performance, we must identify suitable performance indices with which to evaluate distributed scheduling algorithms. However, computer performance is not easily defined. The goal of a scheduling policy is to allocate resources in such a way that the performance expectations of the users are most nearly met. User performance expectations generally center on the quality of service provided to the processes they initiate, for which both wait time and wait ratio are accepted measures. **Wait time** is the total amount of time a process spends waiting for resources, while **wait ratio** is the wait time per unit of service. These measures are closely related to *response time* and *response ratio*, with:

$$\textit{response time} = \textit{wait time} + \textit{service demand} \tag{3.2}$$

$$\textit{response ratio} = \textit{wait ratio} + 1$$

The response ratio of a process, then, is the reciprocal of its service rate. The use of wait (response) ratio,

1. A summary of notation is contained in appendix A.

similar to such human institutions as grocery store checkout lines for '10 items or less,' carries the assumption that a process requesting more resource use should expect to wait longer. Particularly for systems in which processes are associated with 'owners', such as multiuser timeshared systems, these measures may more accurately reflect the user's perception of performance than wait (response) time. Alternatively, the use of wait (response) time may be more appropriate for systems in which all process have the same owner, such as real-time systems.

While the average quality of service provided is clearly an important performance index, how fairly service is allocated is also a common concern. For example, two users simultaneously initiating equivalent processes expect to receive about the same quality of service. Similarly, a user submitting the same job several times, under equivalent workloads, expects each to receive about the same quality of service. While fairness is certainly a complex issue, a good first-order measure is the level of variation in quality of service provided to processes having the same priority. The most commonly used measures of variation, due to their pleasant mathematical properties, are the standard deviation and the coefficient of variation, which is the standard deviation normalized by the mean. A reduced standard deviation results in more predictable quality of service in absolute terms, though it may be less predictable relative to the mean. Alternatively, a reduced coefficient of variation results in more predictable quality of service relative to the mean, though it may be less predictable in absolute terms.

Fairness may also imply that scheduling is *non-discriminatory*: variation in quality of service received by processes should be strictly random. The correlation between the quality-of-service metric and service demand can be used to measure an important aspect of discrimination in scheduling.

Because no performance objective is uniquely appropriate to all systems, we study performance in terms of a broad range of performance indices. While previous performance studies have often been limited to mean wait time, we consider performance in a broad sense, examining the means and standard deviations of process wait (response) times and wait (response) ratios, and the correlations between wait time and service demand and between wait ratio and service demand.

3.3. Load Distributing Algorithms

We are interested in dynamic, decentralized load distributing algorithms applicable to m*(M/H/1) systems. To aid in the description and analysis of load distributing algorithms, we use the following taxonomy of activities involved in such algorithms:

Negotiation:
A node that has identified itself as either a suitable sender or receiver of a process transfer contacts other nodes, searching for a complimentary node, or *transfer partner*, with which to arrange a transfer.

Process selection and transfer:
The sending node chooses an advantageous process to transfer to the receiving node. Information gathered locally, during the negotiation process, or through remote information maintenance, may be used in making this decision. If the process chosen has already begun execution, its state must accompany it to the receiving node.

Remote information maintenance:
Information is maintained at nodes concerning the state of other nodes, aggregates of nodes, or the distributed system in general. This information may be necessary or helpful to the load distributing algorithm, and may include the load at individual nodes, the mean load over the entire system, or the types of processes resident at specific nodes.

While some load distributing algorithms contain all of these components, the only necessary component is process selection and transfer.

The family of dynamic, decentralized load distributing algorithms includes random, polling, and broadcast algorithms.

Random algorithms compose the simplest class of dynamic load distributing algorithms, since they contain no negotiation phase and maintain no information on the state of remote nodes. A random algorithm does

not attempt to choose a particularly appropriate transfer partner, but simply picks a node at random when its load exceeds some given level. An example of such an algorithm is the Random algorithm [Eager86].

A **polling** algorithm is constrained in its use of the broadcast communication device. While it may make use of the fully-connected nature of the device, it does not use broadcast messages. In its negotiation phase, individual nodes are polled to locate an appropriate transfer partner. Examples of polling algorithms include the PID [Livny82] and Threshold [Eager86] algorithms, as well as [Bryan81].

A **broadcast** algorithm makes use of broadcast messages for negotiation and/or remote information maintenance. Since the destination of a broadcast message is unspecified, a broadcast algorithm easily adapts to appearances and disappearances of nodes from the system. This property is particularly useful for distributed systems that allow nodes to leave and rejoin the system. Examples of broadcast algorithms include the Above Average [Krueg84] and BID [Livny83] algorithms.

3.4. Measuring Local Processor Load

The mechanism for measuring local load, the load at a single node, is at the heart of a load distributing algorithm. To allow load distributing to be effective, this load metric must be closely correlated with the performance objective of the distributed scheduling policy. In addition, it must be quickly and efficiently evaluated, so it is responsive to changes in the local state and has minimum effect on the local state. The length of the CPU queue at a node has been shown to be closely correlated with mean process wait time [Zhou87]. In addition, the length of the CPU queue measures the instantaneous wait ratios of processes residing at the node, as well as instantaneous processor utilization. Finally, the CPU queue length is quickly and efficiently evaluated. Throughout this study, we equate local load with CPU queue length.

CHAPTER 4

Laying a Foundation for the Design of Distributed Scheduling Algorithms

4.1. Introduction

Before embarking on a study of distributed scheduling algorithms, a firm foundation must be laid. A suitable algorithm can not be designed until, first, the *performance objective* that motivates the algorithm has been defined and, second, a scheduling policy that has the potential to meet this objective has been identified. Once this policy has been identified, it can serve as an archetype, and algorithms that implement it can be devised.

A distributed scheduling policy for a general-purpose system can be divided into two components: a *local scheduling discipline* determines how the local resources at a single node are allocated among its resident processes, while a *load distributing strategy* distributes the system workload among the nodes through *process transfer*. Because several choices exist for each of these components, we are faced with a variety of potentially viable distributed scheduling policies from which to choose. The goal of this chapter is to simplify this choice by identifying the performance objectives of a representative set of policies. From this set, the policy having an objective most closely matching that of the system designers can be chosen.

Numerous load distributing algorithms have been proposed over the past several years, with widely varying characteristics. However, Eager, Lazowska and Zahorjan [Eager86] have noted that among these algorithms lie two distinct strategies for improving performance. While *load sharing* algorithms simply attempt to conserve the ability of the system to perform work by assuring that no node lies idle while processes wait for service, *load balancing* algorithms go a step farther by striving to equalize the workload among nodes. Eager, et al. go on to point out that under certain assumptions, the additional work that is required to balance the workload is unnecessary; equal performance can be achieved more simply through load sharing. However, several of these assumptions are too restrictive to allow a realistic model for many types of distributed systems. First, process service demands are assumed to be exponentially distributed. Several measurement studies, however, including [Rosin65, Ander74, Zhou86], have found that process service demands are poorly approximated by exponential distributions. Second, process arrival rates are assumed to be homogeneous across nodes. As noted in section 3.1, however, initiation rates that are heterogeneous may be common for many types of distributed systems. Finally, performance is assumed to be adequately characterized by mean process wait time. As pointed out in section 3.2, however, other indices may also be necessary to accurately reflect the users' perception of performance. In pursuing the primary goal of this chapter, we address the question: Does load balancing have the potential to improve performance with respect to load sharing under more general assumptions?

Similar to load distributing, several choices exist for the second component of a distributed scheduling policy, the local scheduling discipline. These disciplines range from simple non-preemptive disciplines to those that are complex and preemptive. Preemptive disciplines, such as Round-Robin, have been found useful in improving the *fairness* of scheduling for single-processor systems [Klein76]. However, load distributing, by transferring processes from overworked nodes to idle nodes, reduces the average queue length at individual nodes. Since processes are forced to share a processor more rarely than in a single-processor system, fewer opportunities arise for the local scheduler to make use of preemption. A secondary question addressed in this chapter is: Can performance similar to that resulting from preemptive local scheduling in a single-processor system be achieved through non-preemptive scheduling in a distributed system?

The notation used in this chapter is summarized in table 4.1.

4.2. Approach

To distill the performance objective of a scheduling policy from the overall effects on performance of an implementation of that policy, we examine performance under the assumption that scheduling overhead is negligible. Under such an assumption, a policy can be implemented precisely and unambiguously; no

compromises need be made. Thus, the M/H/m-like system described in chapter 3, in which the overheads of context switching, negotiation and process transfer are ignored, provides a vantage-point for identifying performance objectives.

Our approach to identifying the objectives of distributed scheduling policies is to separately identify the objectives of local scheduling disciplines and those of load distributing strategies. We then examine the performance of scheduling policies using combinations of these components. Thus, in addition to identifying and comparing the objectives of distributed scheduling policies, this chapter provides a unique perspective on the interaction between local scheduling disciplines and load distributing strategies in determining these objectives.

To begin, we note that in an M/H/m-like system, both the load balancing (LB) and load sharing (LS) strategies are work-conserving. None of the work potential of the system is wasted by leaving nodes idle while processes wait. However, unlike LS, LB goes beyond conservation of work. By balancing the workload among nodes, each process residing in the system perceives approximately the same level of contention. If the local scheduling discipline is Processor Sharing (PS), each resident process receives service at approximately the same rate. Thus, LB in an M/H/m-like system emulates PS in an M/H/m queue, with the goal of achieving similar performance. Essentially, LB is the distributed analogue of PS. LS, on the other hand, since it is solely concerned with conservation of work, is the distributed analogue of any work-conserving M/H/m scheduling discipline.

This perspective of load distributing strategies allows us to predict the differences in performance between the LB and LS strategies in M/H/m-like systems through a less complicated study of performance in M/H/m queues, where many analytical results are available. In effect, the search for the performance objectives of these strategies is greatly simplified. These performance objectives are then validated through a study of performance in M/H/m-like systems. A second benefit of this perspective is that it allows us to identify a new load distributing strategy, which is discussed in section 4.4.3.

Unfortunately, the second prong of our approach, identifying the objectives of local scheduling disciplines, is more complicated than identifying those of load distributing strategies. Even when the system arrival process is Poisson, load distributing causes the arrival processes at individual queues to be non-Poisson. Additionally, load distributing modifies the service demand distribution observed by individual servers. Local scheduling disciplines, then, operate on G/G/1 queues. Since some of our results for M/H/m queues also apply

m	number of nodes
n	total number of resident processes
$\overline{N}$	mean number of resident processes
ρ	system load
x	service demand of a process
$\overline{X}$	mean process service demand
σ_X	standard deviation of service demand
C_X	coefficient of variation $= \sigma_X / \overline{X}$
$\overline{WT}$	mean process wait time
$\overline{WR}$	mean process wait ratio
σ_{WT}	standard deviation of wait time
σ_{WR}	standard deviation of wait ratio
$r(wait\ time, X)$	correlation between wait time and service demand
$r(wait\ ratio, X)$	correlation between wait ratio and service demand
E(Y)	expected value of random variable Y

Table 4.1 Notation

to G/G/1,[1] we are able to identify some local scheduling objectives. The remaining predictions are validated through a study of performance in M/H/m-like systems. Performance is studied both analytically and through simulation, with simulation results having less than a 3% confidence interval half-width at the 95% confidence level. As noted elsewhere in this thesis, this error level is computed using the method of independent replication [Laven83]. Unless otherwise noted, results are from simulation.

4.3. Performance of Single-Queue Scheduling Disciplines

Since both LS and LB are concerned with conservation of work, we identify analogues for LS and LB from among work-conserving M/H/m scheduling disciplines. Non-preemptive disciplines are sufficient to serve as analogues for LS, since LS is *solely* concerned with conservation of work and preemption gives no advantage in achieving this goal. When necessary, we specifically consider First-Come-First-Served (FCFS), since it is perhaps the simplest M/H/m scheduling discipline and has few goals beyond conservation of work. In their favor, work-conserving non-preemptive scheduling disciplines are non-discriminatory with respect to wait time; process wait times are independent of their service demands. However, non-preemptive scheduling has disastrous consequences for $\overline{WR}$ and σ_{WR}. In appendix B, we show that all such disciplines result in infinite $\overline{WR}$ and σ_{WR} in G/G/m or G/G/m-like systems having exponential, hyperexponential or Erlang service demand distributions. These performance indices become finite only when the processes having the highest wait ratios, those having the shortest service demands, are ignored. Since no analytic solutions exist for the mean and standard deviation of wait time ($\overline{WT}$ and σ_{WT}) resulting from non-preemptive scheduling of an M/H/m queue, we rely on simulation of FCFS scheduling. Figure 4.1, which plots $\overline{WT}$ and σ_{WT} against the coefficient of variation in process service demands (C_X),[2] shows that both $\overline{WT}$ and σ_{WT} increase with C_X for FCFS. In this plot, as well as all other plots of these indices, $\overline{WT}$ and σ_{WT} are normalized by $\overline{X}$.

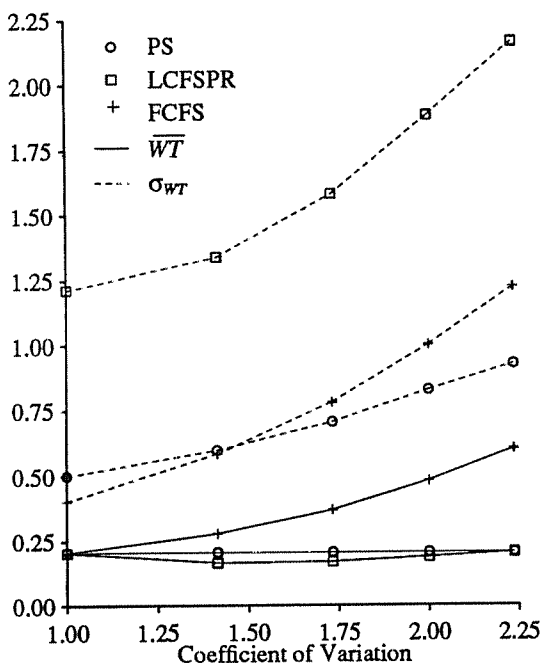

Figure 4.1 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. C_X (m=10, ρ=0.8)

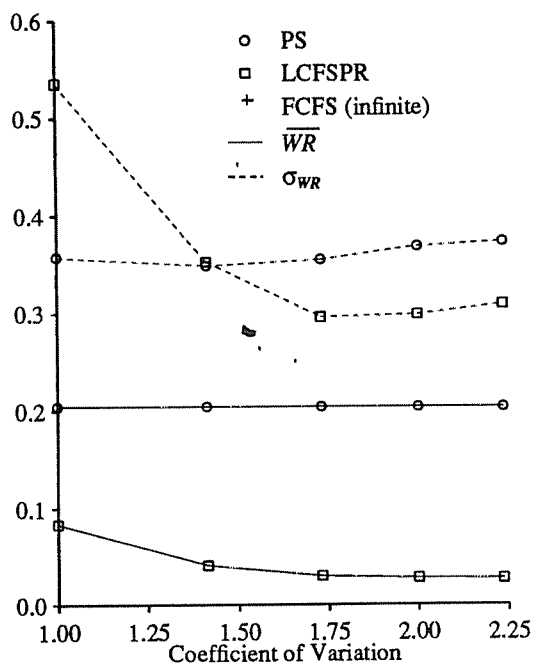

Figure 4.2 $\overline{WR}$ and σ_{WR} vs. C_X (m=10, ρ = 0.8)

1. We assume that the variances in inter-arrival time and service demand are finite.

2. When $C_X > 1$, we assume a 2-phase hyperexponential distribution, with 70% of service demands drawn from the phase having the smaller mean.

PS is the single-queue analogue of LB. Similar to FCFS, PS is a non-discriminatory scheduling discipline, but with respect to wait *ratio*, rather than wait time. This property allows the mean wait ratio ($\overline{WR}$) for PS to be easily derived:[3]

$$\overline{WT} = E[(wait\ time\ /\ x)(x)] = E[(wait\ ratio)(x)] = \overline{WR}\ \overline{X}$$

Rearranging and substituting from equations 3.1 and 3.2:

$$\overline{WR} = \overline{WT}\ /\ \overline{X} = (\overline{N}\ /\ m\rho) - 1 \qquad \text{for } \rho < 1 \tag{4.1}$$

An important property of PS is that $\overline{WR}$ and σ_{WR} are finite for stable ($\rho < 1$) M/H/m and G/G/1 queues (see appendix B). Another advantage of PS over non-preemptive disciplines is that $\overline{WT}$ is independent of C_X [Laven83]. Regardless of C_X, $\overline{WT}$ for PS is given by eq. 4.1. Thus, as shown in figure 4.1, PS results in lower $\overline{WT}$ than FCFS when $C_X > 1$, as well as lower σ_{WT} for high C_X. Since $\overline{WT}$ and $\overline{X}$ are independent of C_X, eq. 4.1 shows that $\overline{WR}$ is also independent of C_X. Simulation results displayed in figure 4.2 validate this claim, as well as showing that σ_{WR} is also independent of C_X.

Unlike FCFS and PS, simulations show Last-Come-First-Served-Preemptive-Resume (LCFSPR) to be a discriminatory scheduling policy, giving better quality of service, both in terms of wait time and wait ratio, to processes having short service demands. Figure 4.2 shows that $\overline{WR}$ and σ_{WR} *decrease* with increasing C_X, though σ_{WT} increases (figure 4.1). Based on these results, we can predict that load distributing algorithms in M/H/m-like systems that discriminate by providing better service to processes having short service demands will reduce $\overline{WR}$ with respect to those that are non-discriminatory. However, this reduction may come at the cost of increased σ_{WT}.

To summarize, conservation of work is not sufficient to provide finite $\overline{WR}$ or σ_{WR}. Even when the 1% of processes having the highest wait ratios are trimmed from the sample, simulations show that FCFS results in considerably higher $\overline{WR}$ and σ_{WR} than PS or LCFSPR. For example, when $C_X = 1$, the trimmed $\overline{WR}$ for FCFS is over 400% that for PS, while σ_{WR} is over 1300%. These values increase rapidly with C_X, to over 12000% for $\overline{WR}$ and 40000% for σ_{WR} at $C_X = 2.24$. Additionally, conservation of work is not sufficient to minimize $\overline{WT}$ when $C_X > 1$. Figure 4.1 shows that both PS and LCFSPR result in lower $\overline{WT}$ than FCFS under such a workload. In addition, PS results in lower σ_{WT} than FCFS for high C_X. Applying these results to distributed scheduling policies, the use of PS as a local scheduling discipline will result in finite $\overline{WR}$ and σ_{WR}, while the use of FCFS or any other non-preemptive discipline will result in infinite $\overline{WR}$ and σ_{WR}. We can predict that the use of PS will result in lower $\overline{WT}$ than any non-preemptive discipline when $C_X > 1$ and lower σ_{WT} for high C_X. Using the perspective of LB as the distributed analogue of PS and LS as the analogue of FCFS, we can predict that LB will result in lower $\overline{WT}$ than LS when $C_X > 1$, lower σ_{WT} for high C_X and, when PS is used as a local scheduling discipline, lower $\overline{WR}$ and σ_{WR} than LS under all conditions.

4.4. Performance of Distributed-Queue Scheduling Policies

In the previous section, we studied the performance of a representative set of single-queue scheduling disciplines. The results were used, in part, to predict the performance resulting from using each of these disciplines as the local discipline of a distributed scheduling policy. Furthermore, noting analogies between these disciplines and the LS and LB load distributing strategies, we predicted the effect on performance of using either the LS or LB strategy in a distributed scheduling policy. In this section, we continue by validating and extending the results of the previous section by comparing the performance of a set of distributed scheduling policies in a *multiple*-queue system. We consider four policies, representing the cross-product of the PS and FCFS local scheduling disciplines and the LS and LB load distributing strategies. These four policies are referred to as LS_FCFS, LS_PS, LB_FCFS and LB_PS. We begin by studying LS_FCFS and LS_PS, and continue with LB_PS and LB_FCFS.

---

3. When random variables Y and Z are independent, $E[g(Y)h(Z)] = E[g(Y)]\,E[h(Z)]$.

4.4.1. Load Sharing (LS)

Under LS, we will show that $\overline{WR}$, σ_{WR}, $r(wait\ ratio, X)$ and $r(wait\ time, X)$ are dependent on the level of heterogeneity in the rates at which processes initiate at individual nodes. Performance in terms of these indices is generally best when initiation rates are homogeneous, and worst when all processes initiate at a single node. An additional complication of LS is that all the performance indices studied in this chapter are dependent on the criteria used to select a process to migrate. We will consider the two bounding criteria: First-Come-First-Migrate (FCFM) selects the process that has least recently arrived at the node, either through initiation or migration, while Last-Come-First-Migrate (LCFM) selects the most recent arrival. By giving an advantage to processes having short service demands, we will see that LCFM captures some of the properties of LCFSPR, reducing $\overline{WR}$ with respect to that of FCFM but increasing σ_{WT}, while FCFM retains the properties of FCFS. These differences are accentuated when process initiation rates are heterogeneous.

Since LS is concerned solely with conserving work, it is naturally allied with a local scheduling discipline that shares this narrow perspective, such as FCFS. This partnership, which we refer to as LS_FCFS, results in a simple distributed scheduling policy that, in an M/H/m-like system, achieves as low $\overline{WT}$ as any other policy when the workload is constrained to $C_X = 1$. In addition, simulations show that under restricted conditions characterized by $C_X = 1$ together with either a small number of nodes ($m \leq 6$) or a high system load ($\rho > 0.8$), LS_FCFS generally results in slightly lower σ_{WT} than policies using PS. However, as predicted in the previous section and shown in figure 4.3, $\overline{WT}$ increases with increasing C_X for LS_FCFS. Though not shown, the same occurs for σ_{WT}. Also as predicted, the single-minded concern of LS_FCFS with conserving work is an insufficient perspective from which to improve $\overline{WR}$ and σ_{WR}. As shown in appendix B, these statistics are infinite, becoming finite only if the processes having the highest wait ratios, which are those having the shortest service demands, are ignored. Simulations show that, even when the shortest processes are removed from the sample, $\overline{WR}$ and σ_{WR} are considerably higher for LS_FCFS than for distributed scheduling policies using PS as a local scheduling discipline.

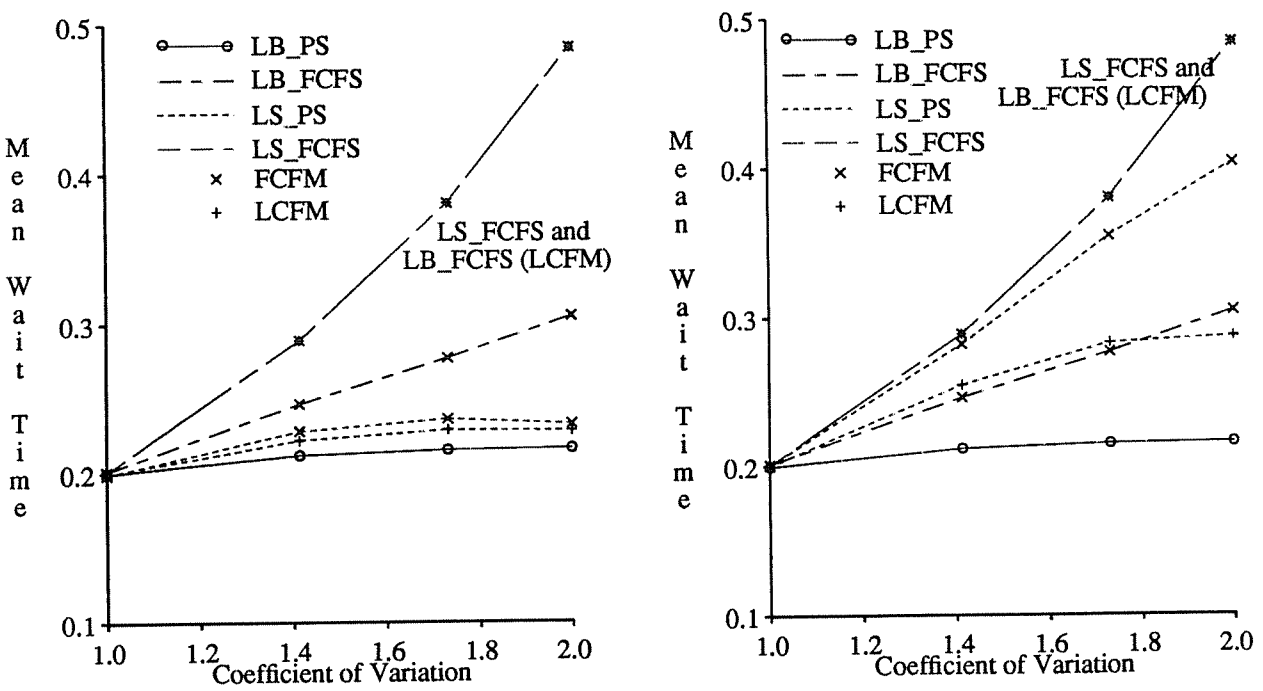

Figure 4.3 $\overline{WT} / \overline{X}$ vs. C_X assuming homogeneous initiation rates (left) or all process initiations at a single node (right) ($m = 10$, $\rho = 0.8$)

Pairing LS with PS appears incongruous, since LS is solely concerned with conservation of work, while PS has broader goals. However, studying this hybrid distributed scheduling policy provides insight into the relative effects on performance of the local scheduling discipline and the load distributing strategy. LS_PS corrects many of the weaknesses of LS_FCFS. One of the most important consequences of merging LS with PS is that $\overline{WR}$ and σ_{WR} are finite (see appendix B). In addition, LS_PS reduces the degrading effect of increasing C_X on $\overline{WT}$, as shown in figure 4.3, and on σ_{WT}. However, heterogeneity in initiation rates continues to have a degrading effect on performance, as can be seen in figures 4.3 though 4.9. In each of these figures, performance for LS-PS is shown to be worse when all processes initiate at a single node than when initiation rates are homogeneous. Figure 4.10, which plots the level of degradation for each performance index against the level of heterogeneity in initiation rates, shows that this degradation is continuous with increasing heterogeneity, rather than occurring suddenly at high levels of heterogeneity. Comparing the results for LS_PS(FCFM) with those for LS_PS(LCFM) in figures 4.3 through 4.10, performance under LS_PS cane be seen to be dependent on which migration selection criterion is chosen. LCFM provides lower $\overline{WR}$ and σ_{WR} (figures 4.4 through 4.9), while FCFM results in lower σ_{WT} (figure 4.10).

4.4.2. Load Balancing (LB)

True PS scheduling in a multiple-queue system, such as an M/H/m-like system, is not feasible, since it requires an infinite migration rate.[4] The goal of the LB strategy is to approximate PS scheduling, achieving similar performance, but with a finite migration rate.

LB is naturally allied with PS as a local scheduling discipline. In appendix C, $\overline{WR}$ and a close approximation of σ_{WR} are derived for LB_PS. In contrast to LS, an examination of equations C.2 and C.2 show that neither of these performance indices is dependent on the homogeneity of process initiation rates or the criteria used to select a process to migrate. Through simulation, we have validated these results and have shown that the remaining performance indices are independent of these factors, as well. How closely LB_PS achieves its goal of emulating PS scheduling can be seen in figure 4.11. In terms of $\overline{WT}$ and σ_{WT}, LB_PS is identical to PS scheduling under all conditions. For other performance indices, LB_PS best approximates PS at high system loads, for small numbers of nodes or for low C_X.

As shown in figures 4.4 through 4.9, LB_PS achieves lower $\overline{WR}$ and σ_{WR}, under all conditions, than any other distributed scheduling policy examined. When all processes initiate at a single node, figure 4.12 shows that LB_PS results in considerable improvement in these indices over its nearest competitor, LS_PS(*LCFM*). These improvements become increasingly pronounced with increasing numbers of nodes, system load and coefficient of variation in service demands. However, when process initiation rates are homogeneous, improvement is large only for σ_{WR}. In addition, as predicted and as shown in figure 4.3, LB_PS results in lower $\overline{WT}$ than any other policy when $C_X > 1$ and lower σ_{WT} when C_X is high. These advantages of LB_PS over LS_PS(*LCFM*) become very large in environments having both heterogeneous initiation rates and $C_X > 1$. For a system having 16 nodes, a system load of 0.8, C_X=2.24 and all processes initiating at a single node, the difference in performance between LB_PS and LS_PS(*LCFM*) is 280% for $\overline{WR}$, 575% for σ_{WR}, 30% for $\overline{WT}$ and 90% for σ_{WT}. Improvement under such conditions is important, since, as noted in chapter 3, heterogeneous initiation rates and hyperexponentially distributed service demands may be common in many general-purpose distributed systems.

Like LS_PS, pairing LB with FCFS as a local scheduling discipline results in a distributed scheduling policy with inconsistent goals. However, since FCFS is simple to implement and can be expected to result in less overhead than most local scheduling disciplines, it is interesting to examine how nearly the performance of LB_FCFS approaches that of LB_PS. In addition, studying such a hybrid policy allows us to gauge the relative effects on performance of its two components. Unfortunately, the influence of FCFS on $\overline{WR}$ and σ_{WR} is greater than that of LB. As shown in appendix B, both $\overline{WR}$ and σ_{WR} are infinite under LB_FCFS. Even when the shortest processes are ignored, simulations show that under LB_FCFS, $\overline{WR}$ and σ_{WR} are much higher than for LB_PS, though they are not as high as for LS_FCFS. Also similar to LS_FCFS and contrasting with LB_PS,

4. When the number of processes in service is greater than the number of nodes, but not a multiple of the number of nodes, some nodes serve one more process than other nodes. Since, under PS, the number of units of service each process must receive during *any* time period is not an integer in these cases, processes must migrate from high nodes to low nodes after a time period of infinitesimal duration.

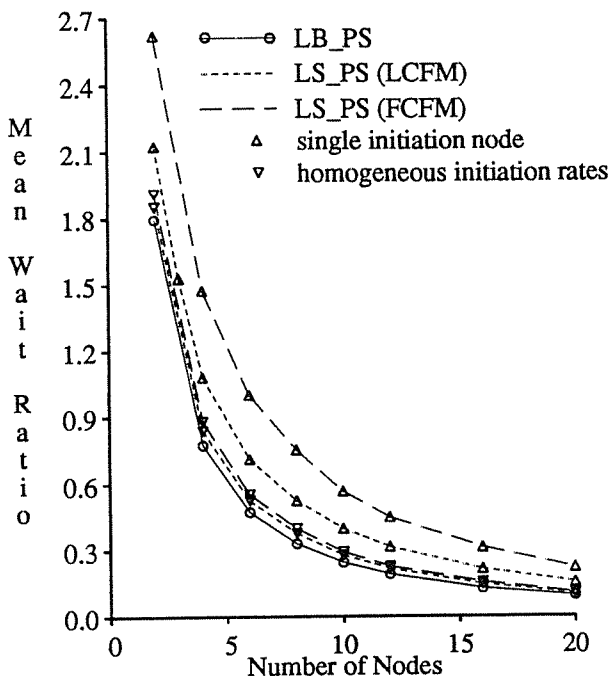

Figure 4.4 $\overline{WR}$ vs. m ($\rho = 0.8, C_x = 1$)

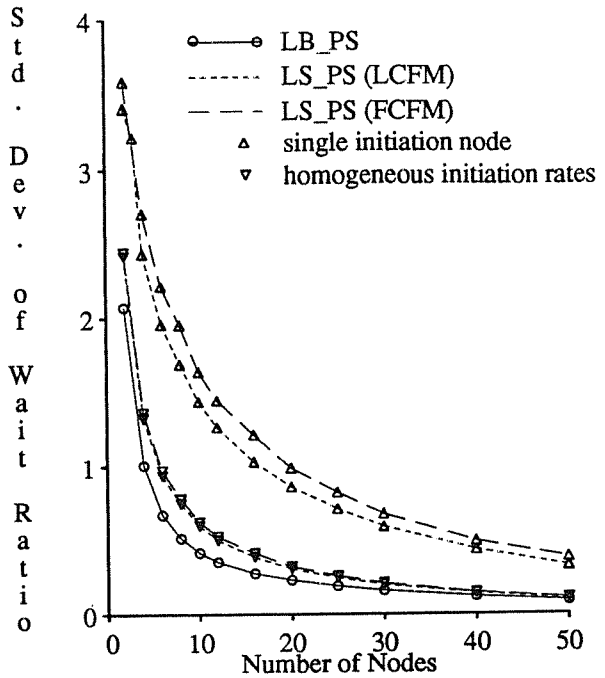

Figure 4.5 σ_{WR} vs. m ($\rho = 0.8, C_x = 1$)

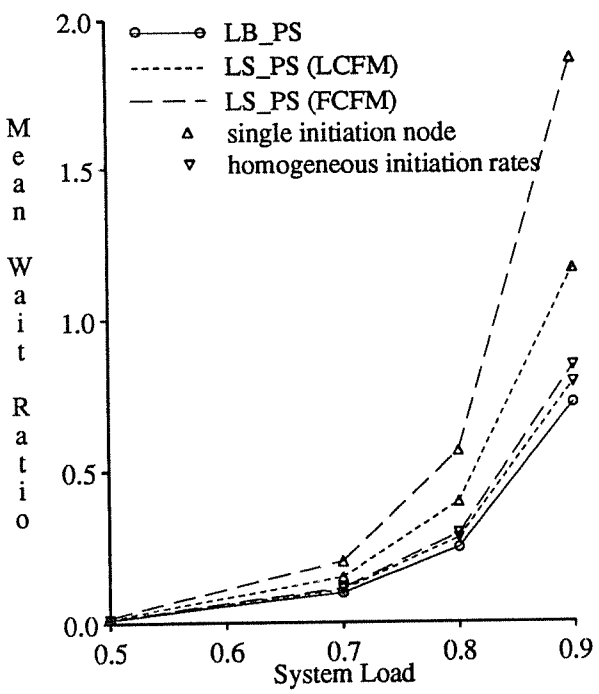

Figure 4.6 $\overline{WR}$ vs. ρ ($m = 10, C_x = 1$)

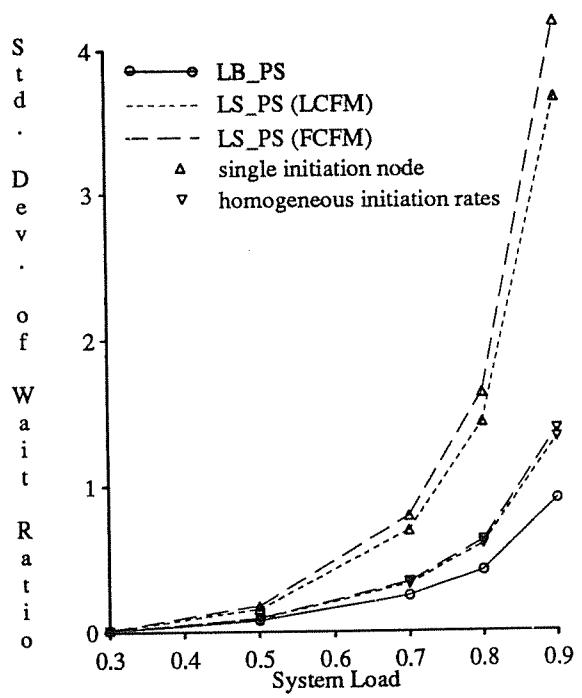

Figure 4.7 σ_{WR} vs. ρ ($m = 10, C_x = 1$)

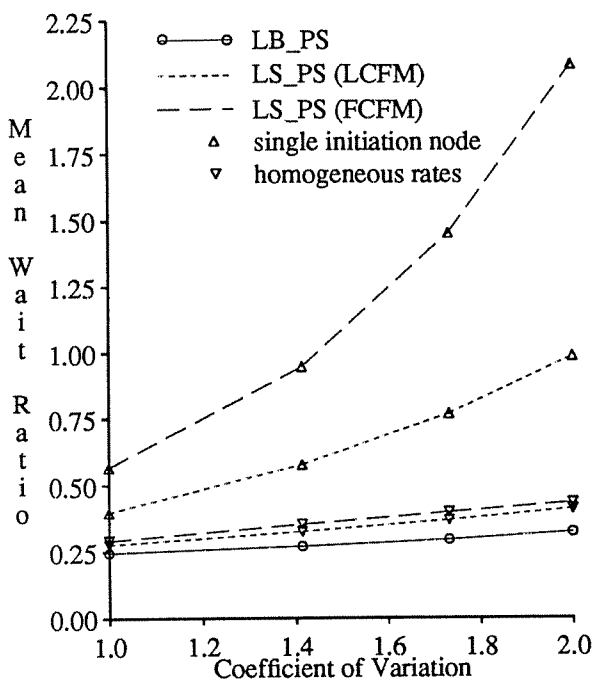

Figure 4.8 $\overline{WR}$ vs. C_X ($m = 10, \rho = 0.8$)

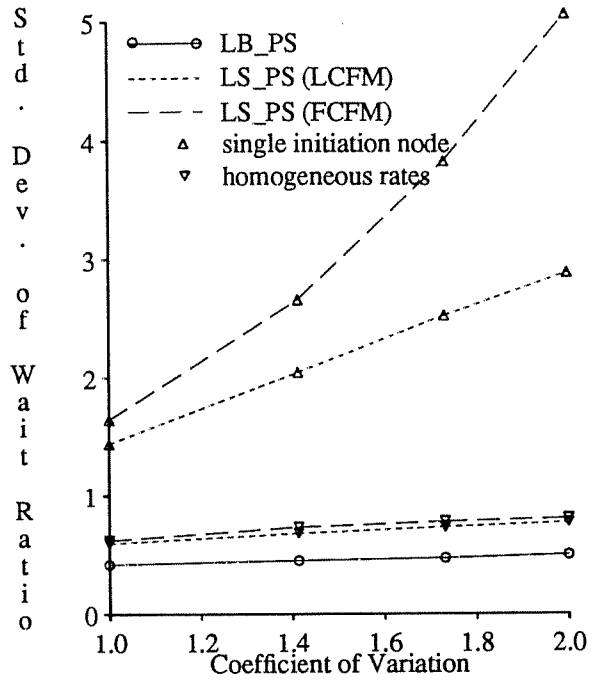

Figure 4.9 σ_{WR} vs. C_X ($m = 10, \rho = 0.8$)

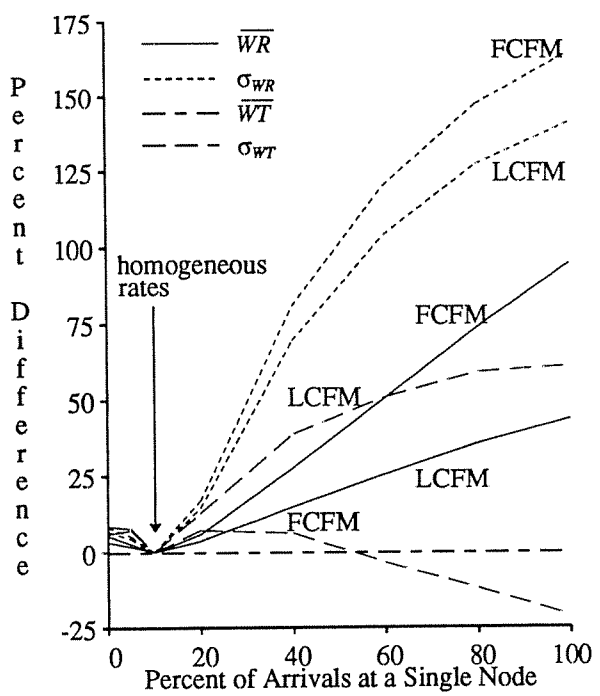

Figure 4.10 Degradation in performance of LS_PS (LCFM) caused by heterogeneous initiation rates. Percent difference in performance index (100 (heterogeneous - homogeneous) / homogeneous) vs. percent of processes that initiate at a single node, with remaining process initiations evenly divided among other nodes. ($m = 10$, $\rho = 0.8$, $C_X = 1$)

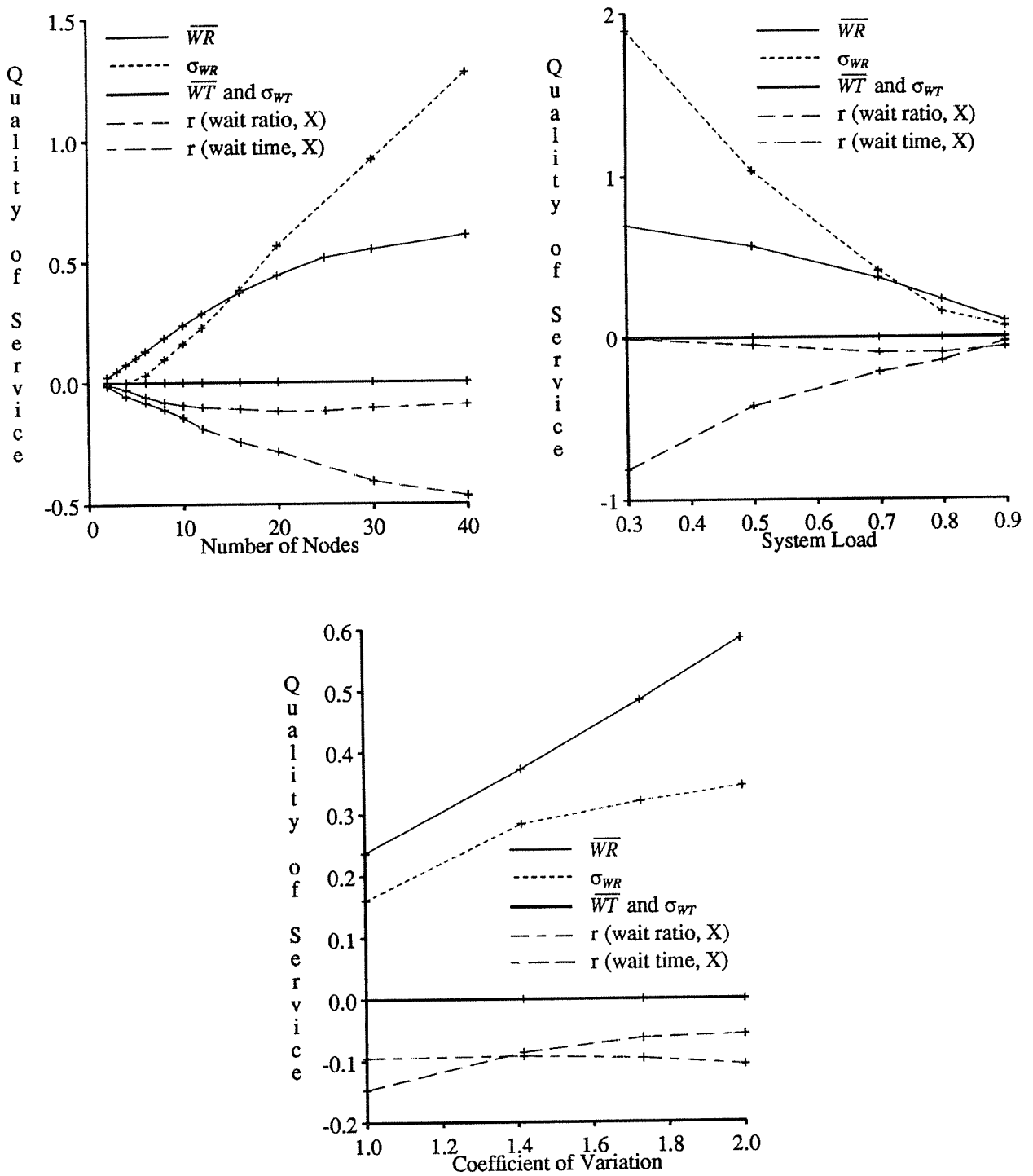

Figure 4.11 Comparison of LB_PS with PS. All indices except r (*wait ratio*, X) are plotted as (LB_PS - PS) / PS. (defaults: $m = 10$, $\rho = 0.8$, $C_x = 1$)

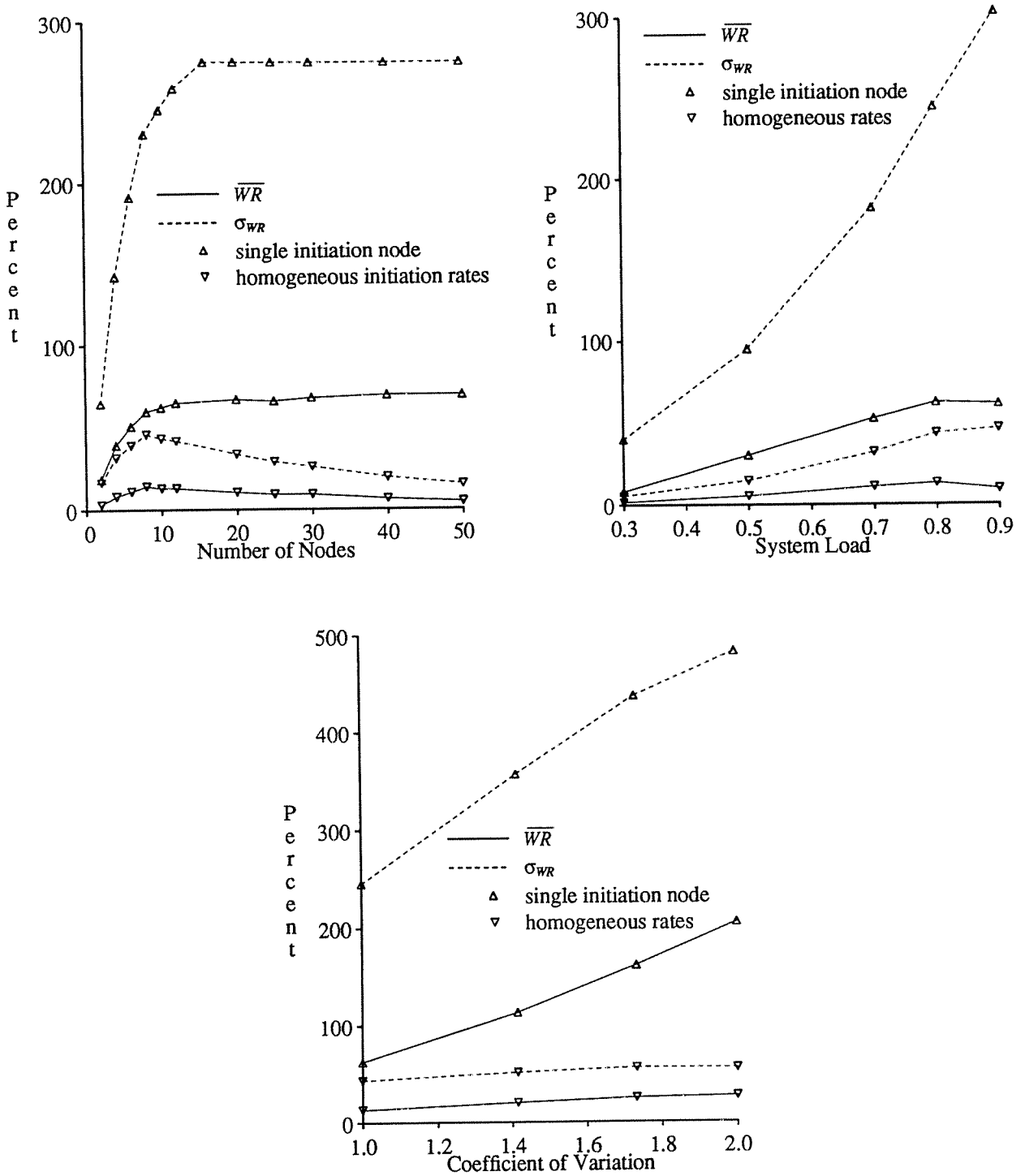

Figure 4.12 Comparison of LB_PS with LS_PS(*LCFM*): Percent difference in $\overline{WR}$ and σ_{WR} (*100(LS - LB) / LB*) (defaults: $m = 10$, $\rho = 0.8$, $C_x = 1$)

performance under LB_FCFS depends on the process selection criterion, the level of heterogeneity in initiation rates and C_X. Using FCFM as the selection criterion generally results in better performance than LCFM, since LCFM causes the same processes to be repeatedly migrated. Showing the influence of the LB strategy, figure 4.3 shows that LB_FCFS(*FCFM*) results in lower $\overline{WT}$ than LS_FCFS for $C_X > 1$. When all processes initiate at a single node, $\overline{WT}$ for LB_FCFS(*FCFM*) is also generally lower than LS_PS for $C_X > 1$. LB_FCFS(*FCFM*) exhibits another characteristic trait of LB: it is discriminatory with respect to wait time. This trait arises because LB_FCFS(*FCFM*) is a preemptive discipline; processes that have begun service may not continue to receive service immediately after migrating.

4.4.3. Load Shuffling (LSh)

LB_PS differs from PS scheduling in that not all processes receive service at the same rate when there are more processes than nodes and $n \bmod m \neq 0$. However, LB_PS can be extended to approximate PS scheduling arbitrarily closely by periodically varying the set of processes that receive service at a faster rate. We refer to these migrations between nodes differing by one in queue length as *shuffling* and to the resultant load distributing strategy as Load Shuffling (LSh). As the length of the *inter-shuffle time* approaches zero, the performance of LSh_PS more closely approximates that of PS. LCFM is not a suitable process selection criterion for LSh, since it results in the same process being repeatedly migrated. This property undermines the goal of LSh, which is to give equal service to all resident processes. The difference in performance between LB_PS in an M/H/m-like system and PS in in M/H/m queue is the potential improvement in performance that can result from LSh_PS. Figure 4.13 plots the percentage of this potential improvement that is achieved by LSh_PS against inter-shuffle time. Surprisingly, migrations between nodes that differ in load by one can significantly reduce $\overline{WR}$, σ_{WR} and $r(wait\ ratio, X)$. This improvement may be large under conditions in which LB_PS poorly approximates PS scheduling: low system load, high C_X or a large number of nodes.

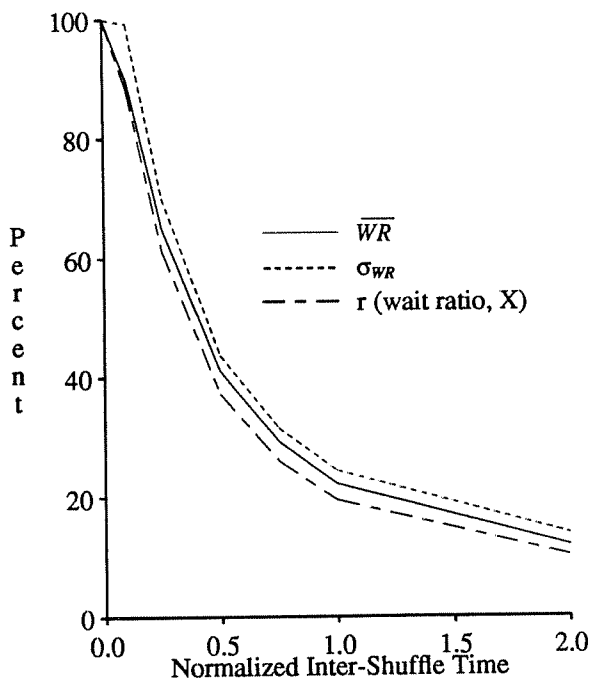

Figure 4.13 Percentage of potential improvement achieved through shuffling (100 (LB_PS - PS) / (LSh - PS)) vs. normalized intershuffle time (intershuffle time / $\overline{X}$) ($m = 10$, $\rho = 0.8$, $C_x = 1$)

4.5. Summary: Distributed Scheduling Objectives

From our study, we can identify the performance objectives of each of the components of a distributed scheduling policy in terms of a broad range of performance indices. These objectives are summarized in table 4.2.

Among local scheduling disciplines, any work-conserving discipline, such as FCFS, minimizes $\overline{WT}$ when $C_X = 1$. However, conservation of work does not minimize $\overline{WT}$ when $C_X > 1$, nor does it address the wait ratio perspective of quality-of-service. When $C_X > 1$, PS provides lower $\overline{WT}$ than any non-preemptive discipline, and lower σ_{WT} than FCFS when C_X is high. Additionally, while many work-conserving disciplines, particularly non-preemptive disciplines, result in infinite $\overline{WR}$ and σ_{WR}, PS provides finite $\overline{WR}$ and σ_{WR}, as well as minimizing $r(wait\ ratio, X)$.

Among load distributing strategies, the objective of LS is similar to that of non-preemptive local scheduling disciplines: reduce $\overline{WT}$ with respect to no load distributing, minimizing it when $C_X = 1$. However, like non-preemptive local scheduling disciplines, LS does not minimize $\overline{WT}$ when $C_X > 1$, nor does it address the wait ratio perspective of quality-of-service. The broader objective of LB is to reduce $\overline{WT}$ relative to LS when $C_X > 1$, and to reduce $\overline{WR}$ and σ_{WR} relative to LS under all conditions. Finally, the performance objective of LSh is to further reduce $\overline{WR}$ and σ_{WR} with respect to LB, and to minimize $r(wait\ ratio, X)$.

When a load distributing strategy and local scheduling discipline are chosen to have matching performance characteristics, the resultant distributed scheduling policy mirrors those characteristics. However, when a distributed scheduling policy has components with inconsistent objectives, its performance is a hybrid of the objective of each component. For such distributed scheduling policies, while the load distributing strategy has a significant effect on $\overline{WR}$ and σ_{WR}, the effect of the local scheduling discipline on these indices is more fundamental, since it determines whether they are finite. In contrast, figure 4.3 shows that the relative effect of the load distributing strategy and local scheduling discipline on $\overline{WT}$ is dependent on the workload. The local scheduling discipline has greater effect when process initiation rates are homogeneous, while the load distributing strategy is more influential when all processes initiate at a single node.

In certain situations, these hybrid policies may be useful. If $\overline{WR}$ is considered important, but σ_{WR} is not, and if $C_X \approx 1$ and process initiation rates are generally homogeneous, LS_PS may be a suitable replacement for LB_PS. Alternatively, if only $\overline{WT}$ is considered important, but $C_X > 1$ or process initiation rates are often heterogeneous, LB_FCFS may provide significantly lower $\overline{WT}$ than LS_FCFS, though not as low as results from LB_PS.

	Local Scheduling Disciplines
FCFS[5]	minimize $\overline{WT}$ when $C_X = 1$.
PS	reduce $\overline{WT}$ relative to FCFS when $C_X > 1$ finite $\overline{WR}$ and σ_{WR} minimize $r(wait\ ratio, X)$
	Load Distributing Strategies
LS	reduce $\overline{WT}$
LB	reduce $\overline{WT}$ relative to LS when $C_X > 1$ reduce $\overline{WR}$ and σ_{WR} relative to LS
LSh	reduce $\overline{WR}$ and σ_{WR} relative to LB minimize $r(wait\ ratio, X)$

Table 4.2 Performance Objectives

5. This performance objective is shared by all work-conserving non-preemptive local scheduling disciplines.

4.6. Conclusions

Much insight into the performance objectives of distributed scheduling policies can be gained by examining performance from a broader perspective than mean wait time. By considering a relatively large set of performance indices, we have shown that different scheduling policies have considerably different objectives and result in significantly different performance. However, since each index has significance only in terms of the performance expectations of the users, no single policy can be identified as being best for all systems. To choose the components of a policy suitable to a particular system, the performance expectations of the users together with the system workload characteristics must be considered.

We have shown that migrations beyond those necessary to conserve work (LS) can have a significant effect on performance. The additional migrations necessary for LB reduce $\overline{WR}$ and σ_{WR}, as well as reducing $\overline{WT}$ when $C_X > 1$. The still more extensive set of LSh migrations further reduces $\overline{WR}$ and σ_{WR} and minimizes $r(wait\ ratio, X)$.

As a load distributing strategy, LS is suitable only for those systems having narrow performance objectives and tightly constrained workloads: $C_X \approx 1$ and generally homogeneous process initiation rates. More typical workloads or broader performance objectives necessitate the use of LB. Among local scheduling disciplines, a non-preemptive discipline, such as FCFS, may be used when $\overline{WR}$ and σ_{WR} can be ignored. When the performance objective of the system includes $\overline{WR}$ and σ_{WR}, a preemptive discipline that gives immediate service to newly arriving processes, such as PS, is necessary.

With consideration for the likelihood of workloads having heterogeneous initiation rates and $C_X > 1$, together with its strong ability to improve $\overline{WR}$ and σ_{WR}, we believe that LB_PS has broad applicability to general-purpose distributed systems.

CHAPTER 5

Making Compromises

5.1. Introduction

In practical distributed systems, the overhead of load distributing can not be considered negligible. As a result, perfect load balancing and load sharing are impossible. The most fundamental of goal of these strategies, conserving work, can not be reached when load distributing itself consumes a portion of the system's capacity to perform work. Consequently, load distributing is unable to achieve a performance benefit without incurring some performance cost. For some load distributing actions, the benefit achieved may be greater than the cost, while for others, the cost is greater. A load distributing algorithm that initiates too many of these latter actions degrades performance, rather than improving it. Such an algorithm may threaten the stability of the system. To avoid actions that damage performance, a load distributing algorithm may find it necessary to back off from its chosen strategy, reducing its *degree* of load distributing by making compromises in how strictly its strategy is followed.

In this chapter, we use a combination of analysis and simulation to study three axes along which the degree of load distributing can be reduced. We begin by considering the first of these axes, which ranges from load balancing (LB), through load sharing (LS), to no load distributing (NoLD). We find that both the point on this continuum that results in the best performance and the range that maintains system stability are dependent on a variety of characteristics of the system and its workload. Complicating the choice of a suitable point on the continuum, these characteristics are subject to change over time.

The second axis depends on the observation that process transfers can be categorized into *placements* and *migrations*. Placement entails selecting a suitable node as the execution site for a process and initiating the process at that node. Later, if another node should become a better execution site, migration entails transferring the process to that node, where it continues executing. Migration generally carries more overhead than placement, since the process state, which must accompany it to its new node, becomes much more complex after execution begins. Intuitively, process migrations may be more likely than placements to incur greater performance cost than benefit. This second axis, then, ranges from the fullest use of migration necessary to implement the chosen strategy to no migration.

The final axis along which the degree of load distributing can be reduced is based on a categorization of load distributing actions as being either sender-initiated or receiver-initiated. Sender-initiated actions are originated by nodes that are potential senders of process transfers, while receiver-initiated actions are originated by potential receivers. In reducing the degree of load distributing, reduction can be applied either to sender-initiated actions or to receiver-initiated actions, with differing effects on performance. At the extremes, a sender-initiated load distributing algorithm allows *only* potential senders to initiate actions, and a receiver-initiated algorithm allows actions to be initiated only by potential receivers. Two studies [Chang86, Eager86a] have found that which of these extremes provides the best performance depends on dynamic characteristics of the system workload. This third axis, then, ranges from sender-initiated, through symmetrically-initiated, to receiver-initiated load distributing. Again, we find that the best point on this continuum and the range that maintains stability are dependent on a variety of dynamic characteristics of the system workload.

5.2. Load Balancing, Load Sharing, or No Load Distributing?

5.2.1. Introduction

While the distinction between the LB and LS strategies is clear when the resource requirements of load distributing are negligible, it blurs as this overhead becomes increasingly significant. If process transfers are instantaneous, there is no advantage, from the point of view of the LS strategy, in transferring processes to

nodes that are not idle. When overhead results in transfers requiring a significant amount of time to complete, however, a node that becomes idle is unable to immediately acquire new processes to execute even though processes wait for service at other nodes. The result is a loss of available processing power in the system. To avoid this loss, *anticipatory* transfers [Livny83] to nodes that are not idle, but are expected to soon become idle, are necessary. These anticipatory transfers increase the transfer rate of an LS algorithm, possibly making it indistinguishable from an LB algorithm. The LB strategy, then, can be considered to be a special case of the LS strategy that performs a particular level of anticipatory transfers.

Even as load distributing overhead tends to increase the transfer rate necessary to best meet the LS performance objective, this same overhead also *limits* the transfer rate that best meets either the LS or the LB objective. While a transfer can reduce contention by avoiding wasted CPU bandwidth, this benefit is reduced by the overhead of the transfer, which has the effect of raising contention. For some transfers, this performance cost may outweigh the benefit of the transfer. To most nearly meet the LS performance objective, it may be necessary to back away from the LS strategy to avoid these cost-ineffective transfers. In extreme cases, the LS performance objective may be best met by turning load distributing off entirely. The continuum between LS and no load distributing, then, can be considered to contain useful approximations of the LS strategy. Similar to LS, it may be necessary to back off from the LB strategy to best meet the LB performance objective. Possibly, the LB performance objective will be best met through the LS strategy, or possibly by turning load distributing off. Thus, LS and no load distributing can also be considered to be approximations of the LB strategy.

These two competing forces that come into play when overhead is not negligible break down the distinction between the LB and LS strategies. No longer is the LB performance objective coupled with the LB strategy, or the LS objective with the LS strategy. Instead, the strategy that best meets either objective lies somewhere on the continuum illustrated by figure 5.1, ranging from no load distributing to load balancing. In this section, we examine the effects that workload characteristics and resource availability have in determining which point on this continuum best meets either the LB or the LS performance objective. In particular, we address the question: Can an algorithm implementing a fixed point on this continuum provide reasonable performance throughout the wide range of conditions that may occur in a distributed system? We show that the answer is no.

We begin by focusing on the portion of the NoLD/LS/LB continuum that lies between LS without anticipatory transfers and LB. Using a limited analytic model, we identify characteristics of the system and its workload that affect the ability of anticipatory transfers to improve performance. This analytic study provides direction for the subsequent detailed simulation study, based on a representative load distributing algorithm. This broader study shows the effect of a variety of characteristics of the system and its workload in determining the point on the continuum that provides the best performance, and the range that results in system stability. Finally, we summarize our results and draw conclusions.

5.2.2. An Analytical Perspective on Anticipatory Transfers

In this section, we identify characteristics of the system and its workload that affect the performance cost and benefit imparted by anticipatory transfers. The goal is to identify conditions under which anticipatory transfers are most likely to improve performance, as well as conditions under which performance is most likely to be harmed. We take an analytic perspective that is independent of design details of specific load distributing

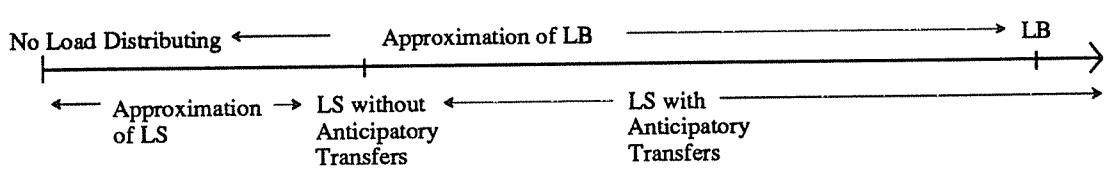

Figure 5.1 The NoLD/LS/LB continuum

algorithms.

An *unshared* state occurs when some node lies idle while processes wait elsewhere for service. Avoiding unshared states is fundamental to both the LS and LB strategies. We take a narrow perspective on performance benefit, focusing on improvement in the ability to meet this common goal. We do not consider other benefits that anticipatory transfers may impart, such as equalizing the quality of service provided to processes. The performance cost of load distributing, on the other hand, arises primarily from the resource overhead of negotiation and transfer. We focus on these actions in examining cost.

No analytic model is available that both includes the overhead of load distributing and allows system parameters, such as the level of heterogeneity in process initiation rates and the number of nodes participating in the system, to be considered. Instead, we base this study on the M/M/m-like system. Because this model does not include overhead, we are unable to examine performance directly. Instead, we examine the rates with which the actions and states that give rise to performance cost and benefit occur. An increase in negotiation and transfer rates is equated with an increase in performance cost, and a reduction in the *unshared rate* is equated with an increase in performance benefit. This unshared rate is the rate at which nodes become idle while processes wait elsewhere for service. By identifying the parameters that determine these rates, we can identify characteristics of the system and its workload that may affect the ability of anticipatory transfers to improve performance.

Using LS without anticipatory transfers as a baseline, our approach is to examine the change in performance cost and benefit that results from pursuing the level of anticipatory transfers performed by the LB strategy. As indicators of performance cost and benefit, the negotiation, transfer and unshared rates of LS and LB are derived in appendix C. These derivations show that, holding other parameters constant, the lower bound for each rate occurs when process initiation rates are homogeneous across nodes, while the upper bound is reached when all processes initiate at a single node. We refer to these extremes in the workload as the *homogeneous case* and the *single-source case*, respectively. The remaining system parameters that determine the negotiation, transfer and unshared rates in an M/M/m-like system are the number of nodes participating in the system (m), the arrival rate of processes to the system (λ) and the mean process service demand ($\bar{X}$).

Figure 5.2 plots transfer rates and unshared rates in the homogeneous and single-source cases against system load ($\rho = \lambda\bar{X}$). Since the overhead of negotiation is much less than that of a transfer, and since negotiation rates follow trends similar to the transfer rates, negotiation rates are not plotted. System load in these plots is manipulated by varying λ with fixed $\bar{X}$, though the results are similar if $\bar{X}$ is varied with fixed λ. Since LS performs a transfer only when an unshared state occurs, the transfer rate for LS is equal to its unshared rate. In contrast, the relatively higher transfer rate of LB that occurs at system loads greater than 0.5 results in an unshared rate lower than that of LS. In the homogeneous case, the increase in the transfer rate is large, though the reduction in the unshared rate is small. We can predict that when initiation rates are homogeneous, the anticipatory transfers pursued by LB are likely to be least effective at high system loads. As a result, LB is likely to perform most poorly relative to LS under such loads. In contrast, in the single-source case, the increase in transfer rate from LB results in a greatly reduced unshared rate. Increasing load is likely to improve the effectiveness of the anticipatory transfers performed by LB in the single-source case. As a result, the performance of LB relative to LS is likely to improve with increasing load in this case. In both cases, LB and LS have nearly identical rates at low system loads, and can be expected to result in the same performance under such workloads.

To illustrate the effect of the number of nodes participating in the system on performance cost and benefit, figure 5.3 plots the transfer and unshared rates against the number of nodes for the homogeneous and single-source cases. In the homogeneous case, all the rates converge as the number of nodes increases. This result is expected, since LB has the opportunity to differ from LS only when the number of processes in the system is more than one greater than the number of nodes. The probability of this occurrence approaches zero as m approaches infinity:

$$\lim_{m \to \infty} \left[\sum_{n=m+1}^{\infty} p_n \right] = 0$$

If the anticipatory transfers performed by LB give it an advantage over LS when there are few nodes in the homogeneous case, the advantage will decrease if the number of nodes increases. For a system having a large

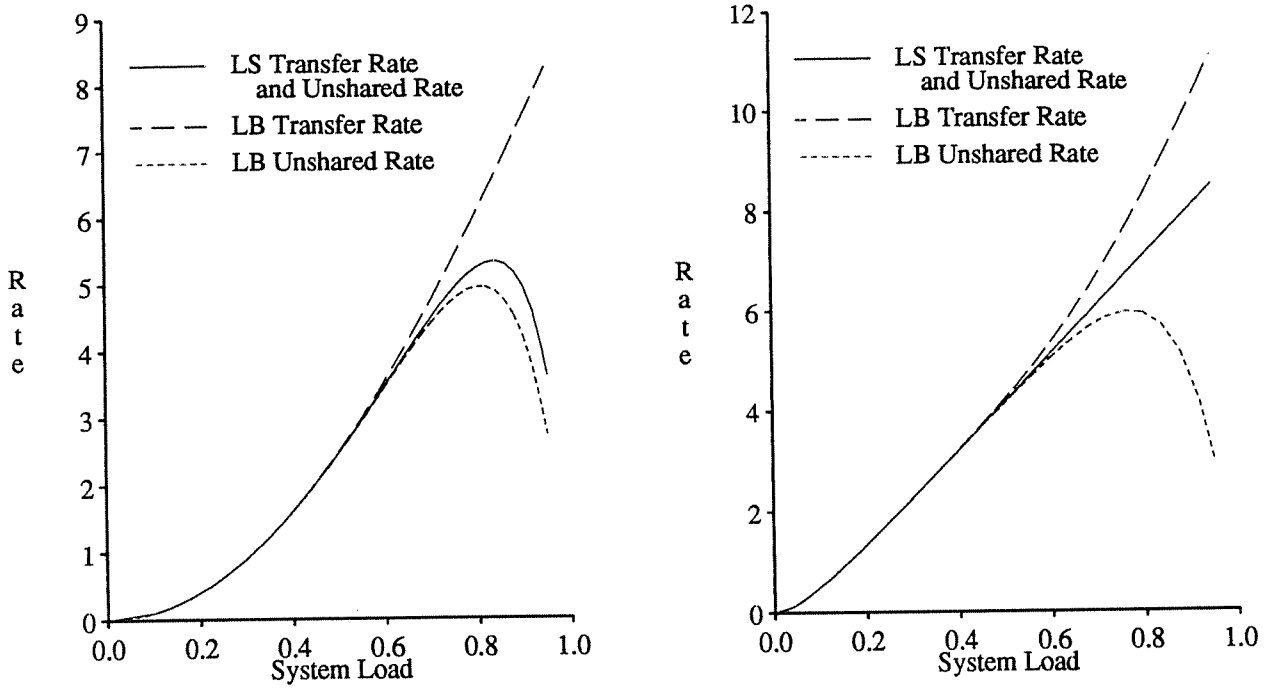

Figure 5.2 Transfer rate and unshared rate vs. system load for the homogeneous case (left) and single-source case (right) (varying λ, $\bar{X} = 1, m = 10$)

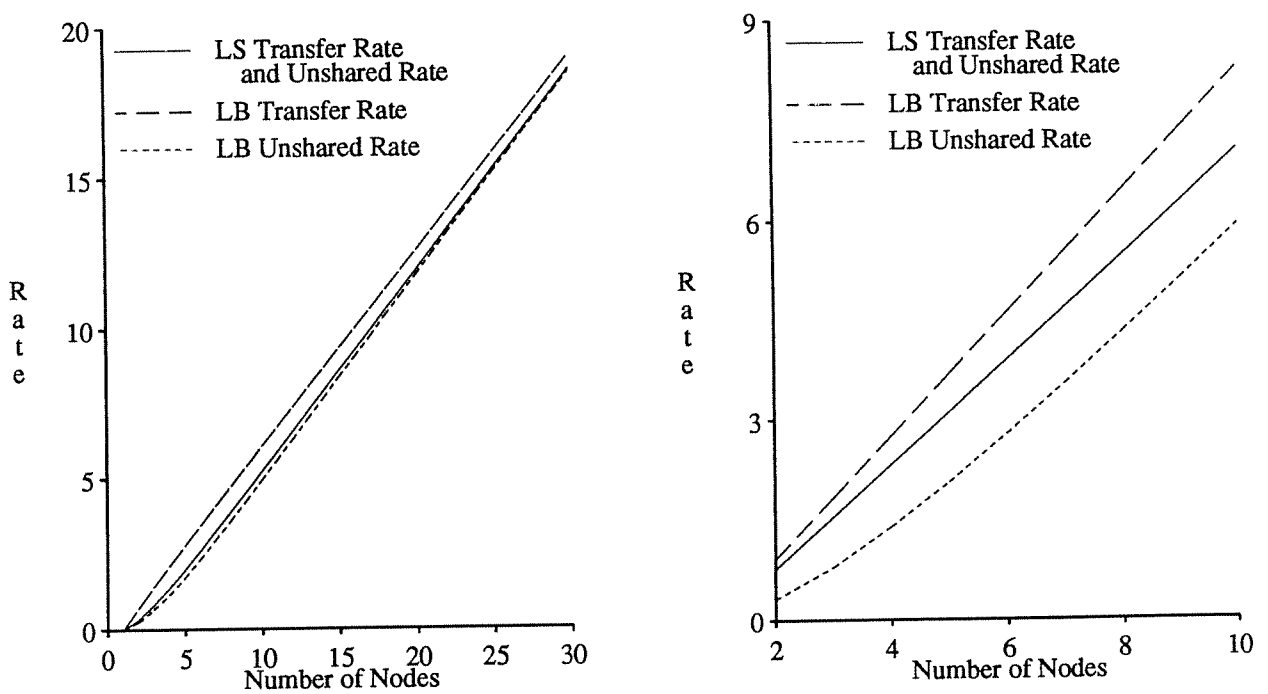

Figure 5.3 Transfer rate and unshared rate vs. number of nodes for the homogeneous case (left) and single-source case (right) ($\lambda = 0.8, \bar{X} = 1$)

number of nodes, LB and LS perform identically in the homogeneous case. In contrast, figure 5.3 shows that the rates tend to diverge in the single-source case. If LB has an advantage when there are few nodes in this case, it is likely to retain that advantage if the number of nodes increases.

In summary, this limited study shows that the ability of the level of anticipatory transfers pursued by LB to improve performance relative to LS depends on the level of heterogeneity in process initiation rates. In the homogeneous case, anticipatory transfers decrease in effectiveness with increasing number of nodes or at high system loads. For sufficiently high loads, these transfers may result in greater cost than benefit, allowing LS to perform better than LB. The opposite holds in the single-source case. Under such a workload, anticipatory transfers remain cost-effective as the number of nodes increases, and they increase in effectiveness with increasing system load. For sufficiently high loads, LB is likely to perform better than LS in this case.

5.2.3. Simulation Study

In this section, we study the effect of characteristics of the system and its workload in determining the point on the NoLD/LS/LB continuum that provides the best performance, and the range that results in stability for the system. This study relies on simulation, based on the m*(M/H/1) distributed system model. This simulation assumes that negotiation and transfer require use of CPU resources at the sending and receiving nodes, as well as use of the communication device. All simulation results presented in the section have less than 6% error at the 95% confidence level.

We begin by introducing the PollGen load distributing algorithm, on which these simulations are based. Continuing, we discuss other assumptions of the simulation study. Finally, the results are presented and discussed.

The PollGen Load Distributing Algorithm[1]

The design of a load distributing algorithm is complicated when it carries resource demands that are not negligible. For example, the performance penalty caused by the overhead of negotiating with every node in the system to find the best *transfer partner* may be prohibitive, particularly when the likelihood of finding a partner is small. To maximize performance, a load distributing algorithm may be forced to limit negotiation to a subset of nodes. As a second example, because process transfers require time to complete, the length of the CPU queue at a node is not a sufficient measure of its load. Since transfers are not instantaneous, the queue length does not change as soon as a transfer is negotiated. However, a node must not commit itself to accept so many processes that, when they finally arrive, it becomes overloaded. The load metric used by PollGen avoids this problem augmenting the node's queue length with its *reservations* [Livny83], the change in queue length expected to be induced by processes that are in transit. The number of reservations recognized by a node increases when the node agrees to accept a process transfer, and decreases when the transferred process arrives or the node becomes convinced that it will never arrive. Reservations, then, have the affect of making transfers appear atomic. Finally, when resource overhead is not negligible, sophisticated criteria for selecting a process to transfer become advantageous.

In our experiments, the process selected to transfer belongs to the set of processes that have been transferred least often among those residing at the sending node. This criterion assures that a process that meets the remaining criteria particularly well is not repeatedly migrated, disproportionally degrading its service, and thus degrading the standard deviations of response time and response ratio (σ_{WR} and σ_{WT}). A second criterion is applied only to a process that has begun executing. Such a process is eligible for transfer only if it has executed for at least F * *sender CPU time required for transfer*, where F is a parameter of the algorithm and the CPU time required for a transfer is calculated from the size of the process and the CPU overhead per transfer message at the sender. This second criterion avoids transferring very short processes, the overhead of which would severely degrade their wait ratios. In addition, since the expected residual service demand of a process increases as it accumulates processing time, given that service demands are hyperexponentially distributed, this second criterion avoids transferring those processes having the shortest expected residual service demands, which have the most transient effects on the load of the sender. So that process placement can be taken

1. A more detailed description of the PollGen algorithm can be found in appendix E.

advantage of, this second criterion is not applied to newly-arrived processes. As noted earlier, placement may incur significantly less overhead than migration. Among the processes meeting these two criteria, the process selected to transfer has the smallest *transfer size*, which includes the size of its executable image as well as any data that must accompany it.

In our experiments, we use a single generalized polling algorithm, called *PollGen*. This algorithm is similar to several that have previously been studied, including Livny and Melman's PID algorithm [Livny82] and Eager, Lazowska and Zahorjan's Thresh and Receiver algorithms [Eager86, Eager86a]. This algorithm differs from these algorithms, however, in two significant ways. First, the degree of load distributing that it pursues is easily manipulated through several parameters, and second, in its strictest form, PollGen is symmetrically initiated. Under PollGen, a node initiates negotiation with probability *SendProb* when the initiation of a process causes the load to be at least T_{Sneg}. In addition, negotiation is initiated with probability *RecvProb* whenever the completion of a process causes a node to become idle. During negotiation, identification of a suitable transfer partner depends on two thresholds: First, for all transfers, a suitable partner differs in load from the node initiating negotiation by at least T_{Tdiff}. Second, and applying only to those transfers that are sender-initiated, the load of a suitable receiver must be less than T_{Rmax}. This last parameter can control the strategy pursued by PollGen. An algorithm following an LS strategy without anticipatory transfers has $T_{Rmax} = 0$, while an algorithm implementing the LB strategy has $T_{Rmax} = \infty$.

Under the PollGen algorithm, the way in which negotiation proceeds depends on whether it is receiver-initiated or sender-initiated. An idle receiving node chooses the *first* suitable transfer partner, so that it is not idle any longer than necessary. A potential sending node, however, searches for the *best* partner: an idle node. Transfers that correct unshared states, then, take precedence over anticipatory or load balancing transfers. To negotiate, an idle node polls a randomly chosen set of *PollLimit* nodes until a suitable partner is found. If no suitable partner is found, the node remains idle. A sending node polls a random set of *PollLimit* nodes, searching for an idle node, which can then agree to accept a process. If no idle node is found, each node from this set having a load $\leq T_{Rmax}$ is polled again, beginning with the node having the lowest load when last polled, until a node at which the load remains $\leq T_{Rmax}$ is found. If no such node is found, the sending node remains overloaded.

The default values for the parameters that control the degree of load distributing pursued by PollGen are:

T_{Sneg}	2
T_{Tdiff}	2
T_{Rmax}	0
SendProb	1
RecvProb	1
PollLimit	5

Simulation Assumptions

We compare the performance of a distributed system without load distributing with three versions of PollGen that differ from the default only in T_{Rmax}. These algorithms vary from load sharing without anticipatory transfers ($T_{Rmax} = 0$), through load sharing *with* anticipatory transfers ($T_{Rmax} = 1$) to load balancing ($T_{Rmax} = \infty$). So that these versions of the PollGen algorithm are easily identified with the strategies they implement, we refer to them as NoLD, LS(0), LS(1) and LB.

The process service demand distribution assumed is derived from data collected by Leland and Ott [Lelan86] from over 9.5 million processes executed by a VAX-11/780 under UNIX[2]. Using weighted least squares regression analysis, we have found that a very good fit for the published data is the 3-phase hyperexponential distribution having the following probability density function:

2. VAX is a trademark of Digital Equipment Corporation and UNIX is a trademark of AT&T Bell Laboratories.

$$f(x) = .79(e^{-x/.31}/.31) + .192(e^{-x/2.8}/2.8) + .018(e^{-x/27}/27)$$

The mean service demand, $\bar{X}$, deriving from this density is 1.27 seconds, while the coefficient of variation, C_X, is 5.3. We feel that this distribution is more useful for our purposes than that found by Leland and Ott ($1 - F(x) \approx rx^{-c}$) because, due to our choice of performance indices, we are interested in an accurate model for the largest *number* of processes (hence our use of least squares regression), rather than for the processes accumulating the greatest portion of CPU time. Thus, in addition to providing a usable density function, our model is a significantly better fit for all but the 0.2% of processes having the longest service demands. We generalize the above density function to form a family of functions sharing C_X, but varying in $\bar{X}$:

$$f(x) = (.79/.243\bar{X})e^{-x/.243\bar{X}} + (.192/2.2\bar{X})e^{-x/2.2\bar{X}} + (.018/21.2\bar{X})e^{-x/21.2\bar{X}}$$

Informal observation of user processes executed on research computers in our department indicates that process sizes are independent of their service demands and that they can be accurately modeled by an exponential distribution. We thus assume that process *migration* sizes, which include all information that must be transmitted from sending nodes to receiving nodes as a result of migrations, are exponentially distributed and independent of all other process characteristics.

We assume that CPU service demands for negotiation and transfer messages preempt all other processing. This assumption implies that processes may not begin to receive service immediately on initiating. Appendix B shows that for such scheduling, the mean and standard deviation of wait ratio ($\overline{WR}$ and σ_{WR}) are infinite when measured over the entire population of processes. To avoid this problem, we exclude the processes having the shortest 1% of service demands when measuring these performance indices. All mean and standard deviation of wait time measurements ($\overline{WT}$ and σ_{WT}) are normalized by $\bar{X}$. The performance axis for most of the plots we present has a logarithmic scale, to allow better comparison of relative performance throughout the range examined. Default parameters are:

Number of nodes (m)	20
System load (ρ)	0.8
Mean process CPU service demand ($\bar{X}$)	1.27 seconds
Mean process size	100K bytes
CPU service demand for transfer packets	.006 seconds
Maximum packet size	4K bytes
CPU service demand for negotiation messages	.003 seconds
Negotiation message size	32 bytes
Communication device bandwidth	10 million bits/second
PollLimit	5
F	0.1

For these defaults, the CPU overhead of migrating a process of average size is 0.3 seconds, divided evenly between the sending and receiving nodes. To stress that this overhead is not as important in absolute terms as it is relative to the service demand of a process, a key parameter in our presentation is *relative migration overhead,* which is the CPU overhead of migrating a process of average size normalized by mean service demand, $\bar{X}$. This parameter, which is 0.24 for the above defaults, can be varied by varying either $\bar{X}$, the mean migration size, or the CPU service demand for transfer packets.

Finally, results presented in this section are based on the assumption that process placement carries the same overhead as migration. Our simulations show that if placement is assumed to incur less overhead than migration, none of the performance trends on which our conclusions are based are affected. We leave an exploration of the effect of variation in the difference between these overheads for the next section.

Results

We begin by examining variation in two parameters that are inaccessible to the analysis of the previous section. We consider the levels of heterogeneity in process initiation rates that lie between the homogeneous case and the single-source case, and variation in the relative overhead incurred by negotiation messages and process transfers.

To illustrate how performance is affected by the degree of heterogeneity in process initiation rates, figure 5.4 plots $\overline{WT}/\bar{X}$ against the portion of the nodes that are *arrival nodes*. The entire system workload is assumed to arrive at this subset of nodes, with an equal rate at each node, while no processes initiate at the remaining nodes. Consistent with the analysis of the previous section, the LB algorithm compares least favorably with LS(0) in the homogeneous case, and most favorably when there is a high degree of heterogeneity. No single algorithm provides the best performance throughout the range of heterogeneity. Each algorithm studied provides the best performance in some subrange, while becoming unstable at higher levels of heterogeneity. All the algorithms, however, improve performance with respect to no load distributing, which results in instability at relatively low levels of heterogeneity. While the relationships of $\overline{WR}$, σ_{WR} and σ_{WT} to the level of heterogeneity are strikingly similar to that shown in figure 5.4, figure 5.5 shows that the wait ratio indices exhibit slightly different crossover points from the wait time indices. At intermediate levels of heterogeneity, choosing the algorithm that provides the best performance depends on which indices are considered most important.

To illustrate the effect that relative migration overhead has in determining the best degree of load distributing, figure 5.6 plots $\overline{WT}/\bar{X}$ against relative migration overhead, which is manipulated by varying $\bar{X}$. Similar trends are shown when relative overhead is manipulated through the mean process size. Beginning with the homogeneous case, LB has an advantage over the LS algorithms at low levels of overhead, because LS fails to pursue some transfers that would have been cost-effective. This result is consistent with the results of chapter 4, which show that the LB strategy has a performance advantage over LS when transfer cost is negligible and

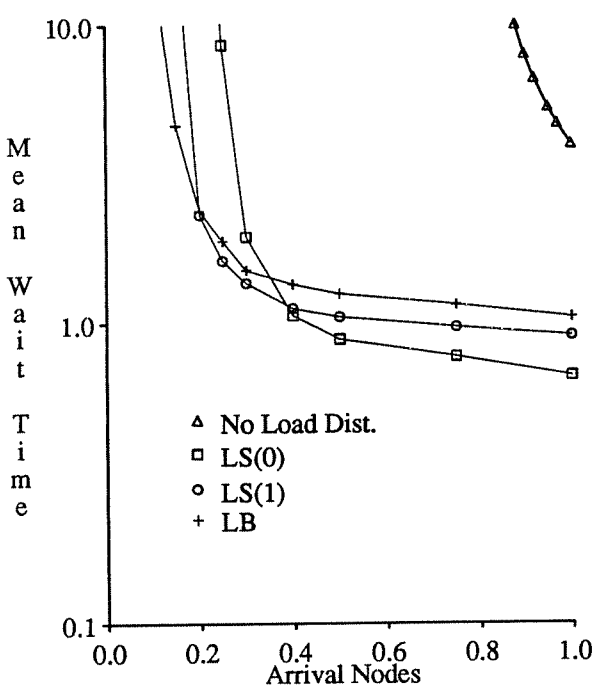

Figure 5.4 $\overline{WT}/\bar{X}$ vs. portion of nodes that are arrival nodes.

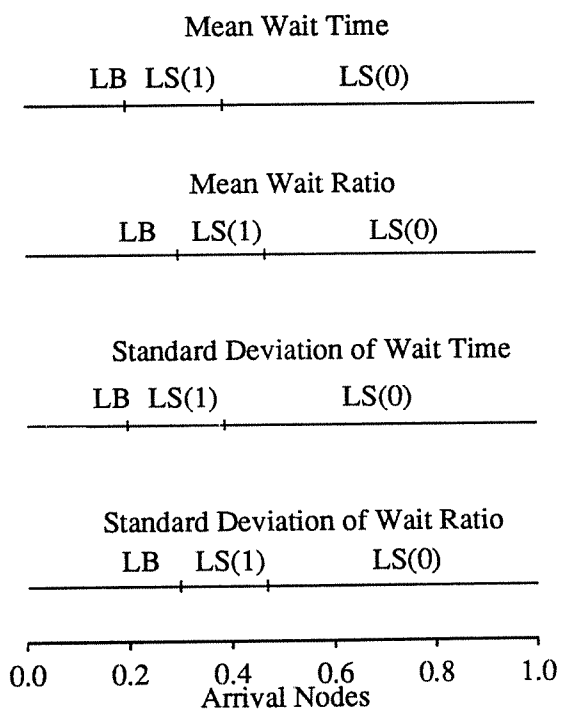

Figure 5.5 For each performance index, the range of heterogeneity in initiation rates in which each algorithm provides the best performance.

process service demands are hyperexponentially distributed. As relative overhead increases, LS(0) achieves better performance than LB by avoiding some of the transfers pursued by LB that have now become cost-ineffective. As a result, the algorithm providing the best performance varies with the level of overhead. With increasing overhead, the best performance is achieved by decreasing load distributing activity. At the extreme, when overhead is very high, LB, LS(1) and LS(0) all result in instability and the best performance results from turning load distributing off. $\overline{WR}$, σ_{WR} and $\sigma_{WT}/\overline{X}$ show trends similar to figure 5.6, though the wait time and wait ratio indices have slightly different crossover points in the homogeneous case. While the crossover point for the wait time indices occurs at relative overhead ≈ 0.05, that for the wait ratio indices occurs at ≈ 0.1.

Next, considering an intermediate level of heterogeneity, with all of the system load arriving at 25% of the nodes, figure 5.6 shows that the trends are significantly different from the homogeneous case. While increasing relative overhead worsens the performance of LB relative to LS(0) in the homogeneous case, increasing overhead improves LB relative to LS(0) in this intermediate case. However, consistent with figure 5.4, LS(1) is the best degree of load distributing at this level of heterogeneity, achieving the best performance.

In the single-source case, LS(0) is unstable, even when load distributing overhead is ignored, because it is too discriminating in its search for a suitable receiving node. Under LS(0), the single arrival node is not able to find such a node within five polls with sufficient probability to maintain stability. However, both LS(1) and LB, being less discriminating, are stable for low levels of overhead. Simulations show that the advantage of LB over LS(1) increases with increasing overhead. In contrast to the homogeneous case, reducing the level of load distributing activity as transfer cost increases is ineffective at improving performance.

To illustrate the effect of system load in determining the best degree of load distributing, figure 5.7 plots $\overline{WT}/\overline{X}$ against system load, which is manipulated by varying λ, with fixed $\overline{X}$. $\overline{WR}$, σ_{WR} and $\sigma_{WT}/\overline{X}$ are not plotted, since they again exhibit the same trends. These results are again consistent with the analysis of the previous section. At low system loads, the performance resulting from LS(0), LS(1) and LB are nearly identical. At higher system loads, the difference in performance between LB and LS(0) depends on the level of

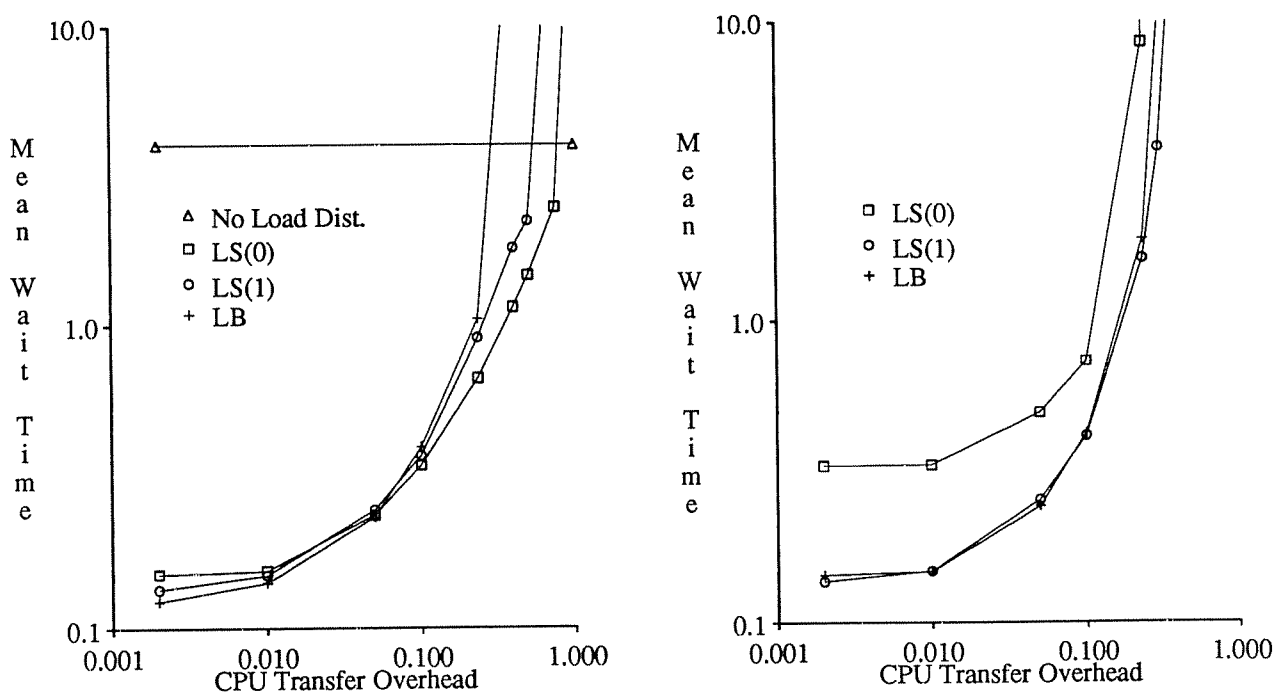

Figure 5.6 $\overline{WT}/\overline{X}$ vs. relative migration overhead (varying $\overline{X}$) for the homogeneous case (left) and assuming 25% of the system nodes are arrival nodes (right).

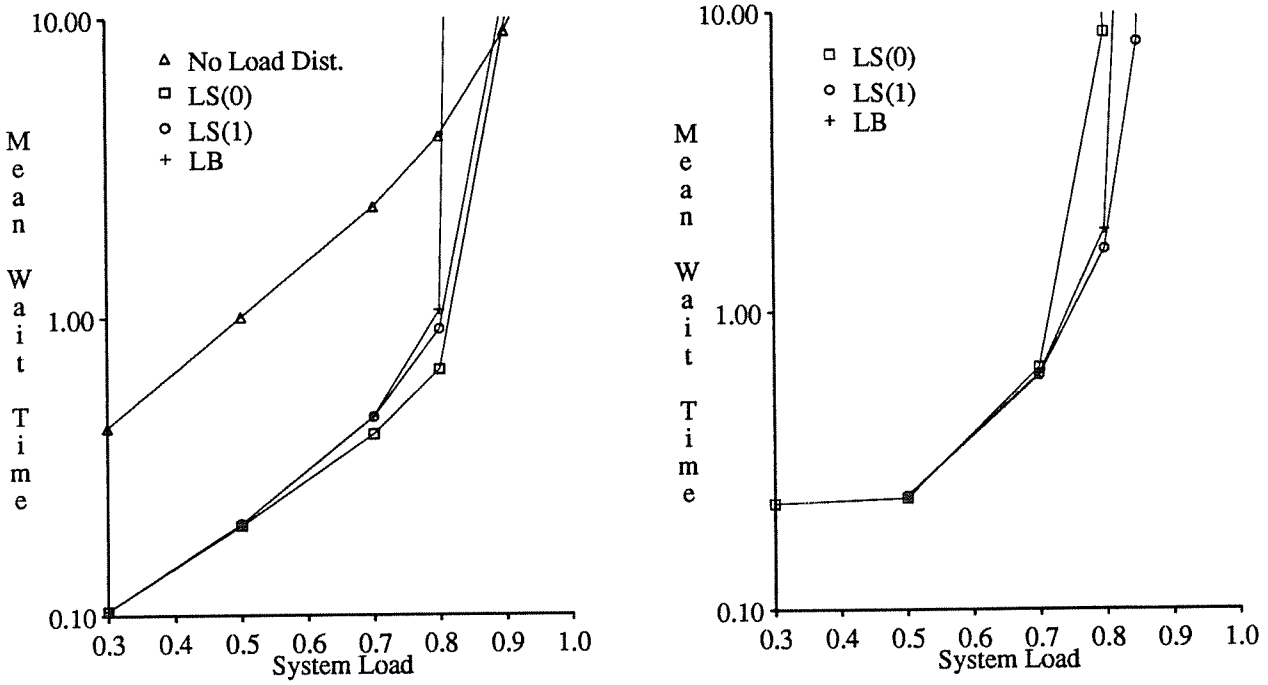

Figure 5.7 $\overline{WT}/\bar{X}$ vs. system load (varying λ with constant $\bar{X}$) for the homogeneous case (left) and assuming 25% of the system nodes are arrival nodes (right).

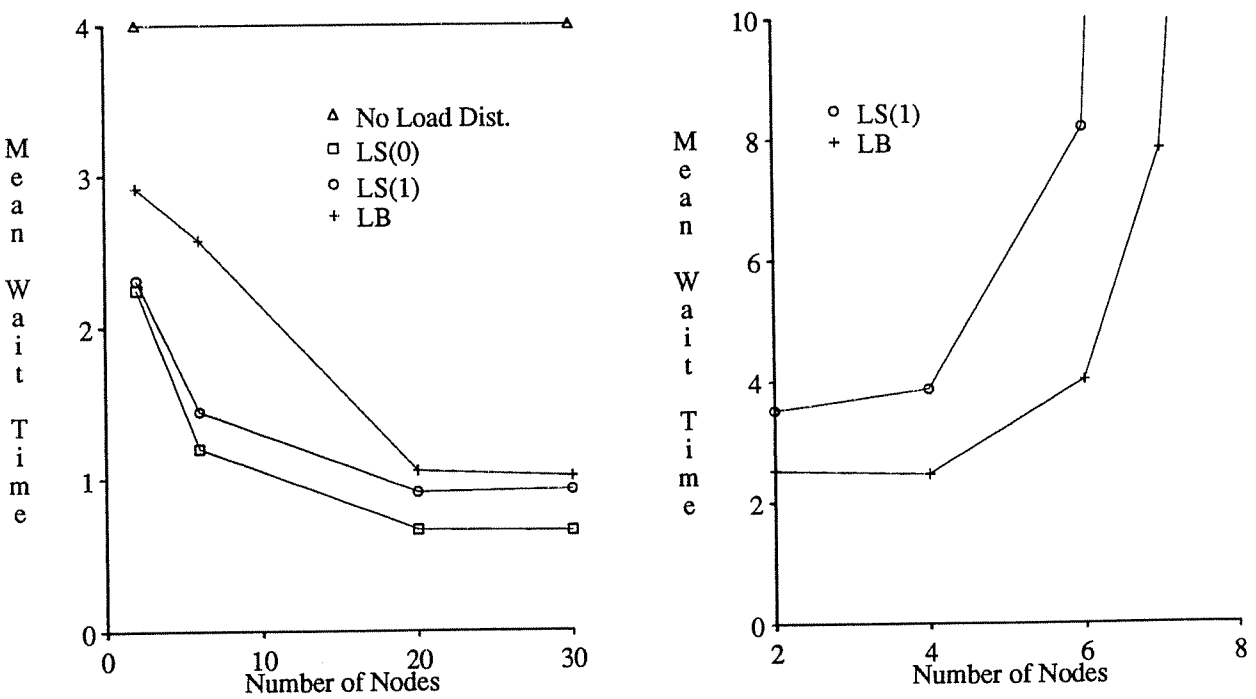

Figure 5.8 $\overline{WT}/\bar{X}$ vs. number of nodes for the homogeneous case (left) and the single-source case (right).

heterogeneity. In the homogeneous case, increasing load improves the performance of LS(0) relative to LB. The opposite occurs when the level of heterogeneity is high. When all processes arrive at 25% of the nodes, increasing load *degrades* the performance of LS(0) relative to LB. However, the intermediate algorithm, LS(1), provides the best performance at this level of heterogeneity.

Finally, figure 5.8 plots $\overline{WT}/\overline{X}$ against the number of nodes participating in the system. Again, $\overline{WR}$, σ_{WR} and σ_{WT} show the same trends. Consistent with the analysis, the difference in performance between LB and LS(0) decreases with increasing nodes in the homogeneous case. However, in the single-source case, the performance advantage that LB has over the LS algorithms *increases* with increasing nodes.

5.2.4. Summary and Conclusions

A continuum of load distributing strategies are available to improve performance in a distributed system, ranging from no load distributing, through the succession of LS without anticipatory transfers, LS with anticipatory transfers, LB and beyond. In this section, we have found that the range on this continuum that provides stability for the system, and the point within that range that results in the best performance, are determined by a wide variety of characteristics of the distributed system and its workload. Choosing the best point on this continuum is further complicated by the fact that several of these characteristics may be subject to wide variation over time:

- CPU service demands tend to be longer and the arrival rate of processes to the system lower for the (predominantly batch) jobs that run at night than for the (interactive) jobs that execute during the day.
- The system load tends to be higher during the day than at night.
- The degree of heterogeneity in arrival rates can be expected to be greater at night and during mealtimes, when most processes will be created by a small group of users.
- The number of nodes participating in the system may change over time. In addition to the transient effects of node failure and periodic maintenance, distributed systems composed of workstations may allow nodes to leave or reenter the system at will.
- The distribution of process sizes may also vary.

We conclude that to maintain stability and to effectively meet either the LB or the LS performance objective over the wide range of conditions that may occur in a distributed system, a load distributing algorithm must adapt to its environment, ranging over the NoLD/LS/LB continuum.

5.3. Are Migrations Really Necessary?

5.3.1. Introduction

Process transfers can be categorized as either being performed non-preemptively, which we refer to as process *placement*, or preemptively, referred to as process *migration*. Placement entails selecting a suitable node as the execution site for a process and initiating the process at that node. Later, if another node should become a better execution site, migration entails transferring the process to that node, where it continues executing.

Migration is more costly than placement, since the process state, which must accompany the process to its new node, becomes much more complex after execution begins. As a result, implementing and maintaining the mechanism necessary to encapsulate, transfer and resume execution from this complex state is expensive. In addition, having implemented this mechanism, it is not obvious what performance improvement might result, since its resource overhead is likely to be much greater than that for placement [Powel83, Theim85]. In light of this two-pronged expense, we address the question: Is migration worthwhile, or can most or all of the performance improvement potentially available through load distributing be achieved using placement only? Or, taking a different perspective: Can reducing the degree of load distributing by disallowing process migrations, allowing only placements, result in performance that is as good or better than that of an unrestricted algorithm?

For many computer systems, the best candidate for local scheduling is a preemptive discipline, such as Round-Robin [Klein76], that quickly provides an initial burst of service to newly-arrived processes. Such disciplines have long been used to provide service acceptable to the users of general-purpose single-processor

systems, and were shown in chapter 4 and appendix B to be necessary to provide equivalent performance for distributed systems. We assume the use of such a local scheduling discipline in this study. Under such local scheduling, process transfers that are negotiated by a receiving node (referred to as *receiver-initiated* transfers [Eager86a]) are always migrations, since it is unlikely that a receiver-node would open negotiation with a potential sender at the moment that a new process arrived at the sender. Transfers that are *sender-initiated* however, may be either placements or migrations, depending on which process the sender chooses to transfer. While receiver-initiated migration has more obvious potential for performance improvement, sender-initiated migration may also improve performance. A load distributing algorithm that is free to choose any process to transfer, rather than being restricted to a newly-arrived process, may choose some other process that can be identified as likely to result in a larger improvement in performance. Though sender-initiated migration holds promise for future study, this section focuses on receiver-initiated migration.

Ignoring, for a moment, the overhead incurred by load distributing, we can predict the conditions under which the addition of receiver-initiated migration will be most useful. Sender-initiated placement can improve performance whenever a process arrives at a busy node, which can potentially occur even when only a single process resides in the system. In contrast, the addition of receiver-initiated migration can improve performance only when a process completes at a time when the system contains more processes than nodes, so that the initial placement of processes resulted in some nodes servicing more than one process. For an M/M/m system composed of 10 nodes that is 65% utilized, the probability of this latter occurrence is only 0.1 [Laven83]. This probability falls to 0.03 if the number of nodes increases to 20, though it rises to 0.5 if the utilization is then increased to 90%. From this simple analysis, the addition of receiver-initiated migration can be predicted to be useful only at high system loads or for systems containing few nodes. However, this result does not address the complications that arise in practical distributed systems, in which load distributing overhead is significant. This section investigates the effect of this overhead on the ability of migration to improve performance.

Though the LB strategy was shown in chapter 4 to have the potential to provide better performance than LS when the overhead of load distributing is ignored, there are at least two reasons why an LS algorithm may prove advantageous for practical systems. First, as shown in the previous section, when overhead is not ignored, an LS algorithm may provide better performance by avoiding process transfers that are not *cost-effective*, since they do not provide enough improvement in performance to justify their overhead. Second, particularly for distributed systems composed of workstations, ownership rights of individual nodes or groups of nodes may preclude LB [Nicho87, Litzk88]. To expose dependencies between improvement in performance and the particular strategy chosen, we consider both the LB and LS strategies.

In the following pages, we show that while placement can indeed achieve much of the performance improvement available through load distributing, considerable additional improvement can be gained through migration. Under many conditions, this additional improvement would be obvious to the users of the system. We conclude that, while a migration facility is not as essential as placement, it will often a worthwhile investment.

5.3.2. Simulation Study

In this section, we compare the improvement in performance resulting from sender-initiated placement with the *additional* improvement that results from augmenting placement with receiver-initiated migration. We study performance through simulation, since no analytic models are available that allow hyperexponentially distributed process service demands, heterogeneous process initiation rates, or the number of nodes participating in the system to be considered. These simulations are based on the assumption that negotiation and transfer require use of CPU resources at the sending and receiving nodes, as well as use of the communication device. All simulation results presented have less than 10% error at the 90% confidence level.

Simulation Assumptions

Assuming, for simplicity, that the input and output of a process reside on disk, the way migration and placement are modeled depends, in part, on the file system structure of the distributed system. If nodes have no local secondary storage, but rely on a shared disk server, placement can be accomplished simply by sending a message to another node specifying the program, input and output files. For such a system, migration carries much greater overhead than placement, since the process state considerably increases the size and complexity

of the data that must be transferred. At the opposite end of the spectrum, if each node has local disk storage and no files are replicated, placement entails transferring the program and its input to the new node, and transferring any output back to the originating node when the program completes. Migration, for such a system, is not greatly more expensive than placement, since the process state does not greatly increase the size or complexity of the transfer. Between these endpoints lie systems having local secondary storage but some replication of files, and systems having 'minimal' local disks used only for swapping. We model process transfers for this spectrum of distributed systems as occurring in two logical parts. A message having constant size for all transfers (128 bytes) is followed by a message having a size corresponding to the migration size of the process being transferred. This size includes its executable image, its state description, and any input or output. For placement, the size of the second message is smaller than the migration size by a factor *PlaceFactor*. A smaller *PlaceFactor* models a system having no local disk storage, while *PlaceFactor* is larger when nodes have local disks and no files are replicated.

Other features of the system may also influence the difference in overhead between placement and migration. For example, this difference may be affected by the ability or inability of a process to dynamically allocate and deallocate primary storage, or by design details of the compiler and loader that determine when storage for static variables is allocated. PlaceFactor, in this study, models differences in overhead between placement and migration that arise for any reason.

The remainder of the simulation assumptions for this section are identical to those of the previous section, except that the following default parameters are assumed:

Number of nodes (m)	20
System load (ρ)	0.85
Mean process CPU service demand ($\bar{X}$)	1.27 seconds
Mean process migration size	100K bytes
CPU service demand for transfer packets	.004 seconds
Maximum packet size	4K bytes
CPU service demand for negotiation messages	.002 seconds
Negotiation message size	32 bytes
Process initiation rates	Homogeneous
Communication device bandwidth	10 Mbits/sec.
PollLimit	5
F	0.1

For these defaults the relative migration overhead is 0.16. In addition, because process migration sizes are exponentially distributed, the performance of PollGen is improved if an upper limit is placed on the sizes of processes that are migrated. In this section, a process is chosen to migrate only if its migration size is less than *MaxMigSize*, which is set to 100K bytes.

We compare the performance of the following five PollGen variants, ranging in strategy from no load distributing (NoLD), through sender-initiated LS and LB (SenderLS and SenderLB), to symmetrically-initiated LS and LB (SymLS and SymLB):

	SendProb	*RecvProb*	T_{Rmax}
NoLD	0	0	0
SenderLS	1	0	0
SenderLB	1	0	∞
SymLS	1	1	0
SymLB	1	1	∞

Results

To compare the improvement in performance resulting from placement with that of migration, we compare the percent *reductions* in each of the performance indices that result from placement:

$$placement\ improvement\ =\ 100\left[1-(Sender\ /\ NoLD)\right]$$

with the *additional* reductions that result from augmenting placement with receiver-initiated migration:

$$additional\ migration\ improvement\ =\ 100\left[1-(Sym\ /\ Sender)\right]$$

Placement improvement for LS, then, measures the performance of SenderLS relative to NoLD, while migration improvement for LS measures the improvement in performance resulting from SymLS relative to SenderLS. Similar relationships hold for SenderLB, SymLB and NoLD.

To begin, we examine the interplay between PlaceFactor and the ability of PollGen to improve performance. Figures 5.9 and 5.10, which plot percent improvement against *PlaceFactor*, show that in spite of the large improvement resulting from placement alone, the addition of migration can provide performance that is significantly better. Even when migration is *greatly* more expensive than placement, at low values of *PlaceFactor*, it can significantly improve performance.

Not surprisingly, while an increase in *PlaceFactor* reduces the ability of placement to improve performance, it generally *increases* the ability of migration to additionally improve performance. Placement becomes less able to improve performance as it becomes increasingly expensive. In contrast, migration becomes better able to additionally improve performance as its overhead relative to placement decreases. We can conclude that load distributing algorithms relying solely on placement are best applied to distributed systems modeled by a small *PlaceFactor*, while migration is most valuable for systems modeled by a larger *PlaceFactor*.

While placement improvement from load balancing is often larger than that from load sharing, we observe the opposite for migration, where improvement is greater for LS than for LB. A trend that is related is that a decrease in *PlaceFactor* is more advantageous for SenderLB than SenderLS, and less harmful to SymLB than SymLS. Both these phenomena have the same cause: the LB strategy is more reliant on sender-initiated transfer, while LS is more reliant on receiver-initiated migration. Intuitively, since the sender-initiated component of an LB algorithm is less restricted in its search for a suitable destination node, placement carries a heavier load distributing burden for LB than for LS. Conversely, receiver-initiated migrations are more necessary to LS. This reliance of LB algorithms on placement and LS on migration can be quantified by calculating and comparing the rates of sender-initiated and receiver-initiated transfers for these two strategies. Derivations of these rates under assumptions of negligible load distributing overhead and exponentially distributed service demands (see appendix D) show that, holding other parameters constant, the lower bound for each rate occurs when process initiation rates are homogeneous, while the upper bound is reached in the *single-source case*, when all processes initiate at a single node. Figure 5.11, which plots the transfer rates against system load, shows that the rate of sender-initiated transfers is higher for LB, while the receiver-initiated transfer rate is *lower* for LB, particularly when process initiation rates are heterogeneous. If migration is a great deal more costly than placement, LB may carry less overhead than LS, even though it has a higher overall transfer rate.

An important feature of figures 5.9 and 5.10 is that, for both placement and migration, improvement in mean response time ($\overline{RT}$) is greater than that in mean response ratio ($\overline{RR}$). Intuitively, since a given reduction in response time reduces the response *ratio* of a long-running process less than that of a short process, this difference in improvement implies that response time improvements have been more heavily 'allocated' to long processes. Alternatively, since a given response ratio improvement reduces the response *time* of a long process more than that of a short process, this difference implies that the response ratio improvements have also been more heavily allocated to long processes. The reason for this bias is that the processes that are potentially most helped by a transfer (the transferred process and the processes left behind at the sender) must execute for some period before they 'recover' from the delay imposed by the overhead of the transfer. The amount that the response time or response ratio of a process is reduced as the result of a transfer increases with increasing residual service demand. Since the residual service demand of a long process is likely to be longer than that of a short process (regardless of the service demand distribution), a transfer that incurs a given level of overhead generally improves the performance of a long process more than that of a short process. This mechanism is

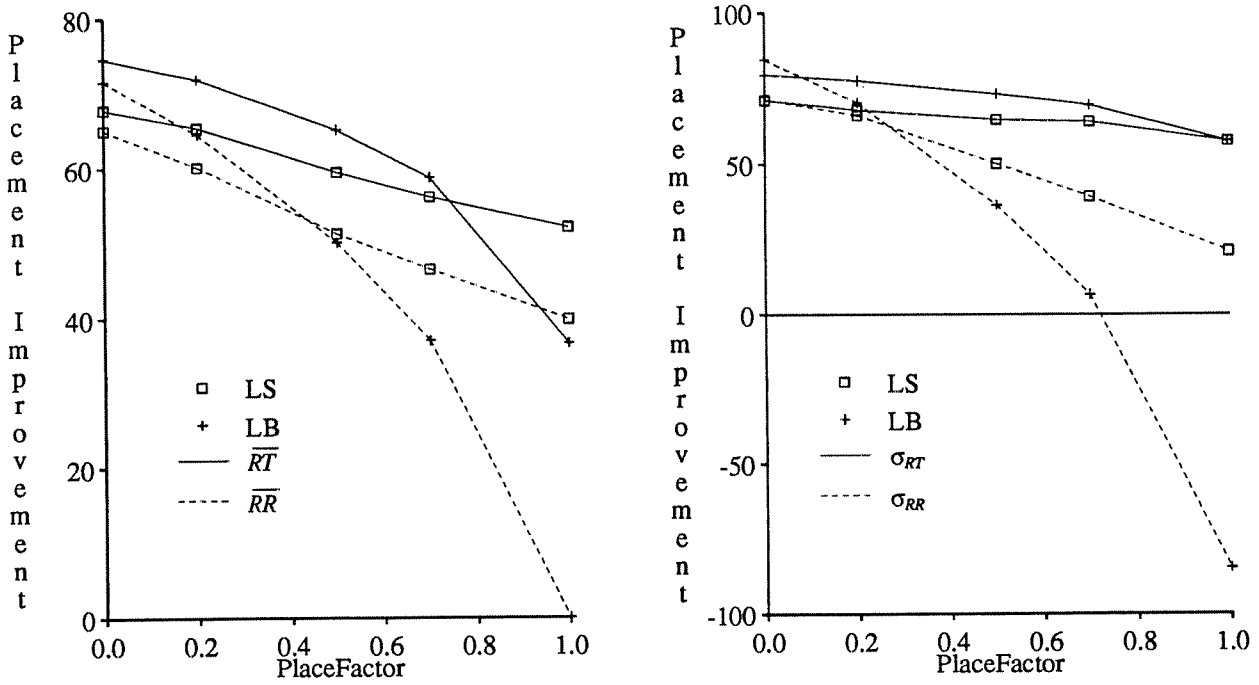

Figure 5.9 Percent placement improvement vs. *PlaceFactor*

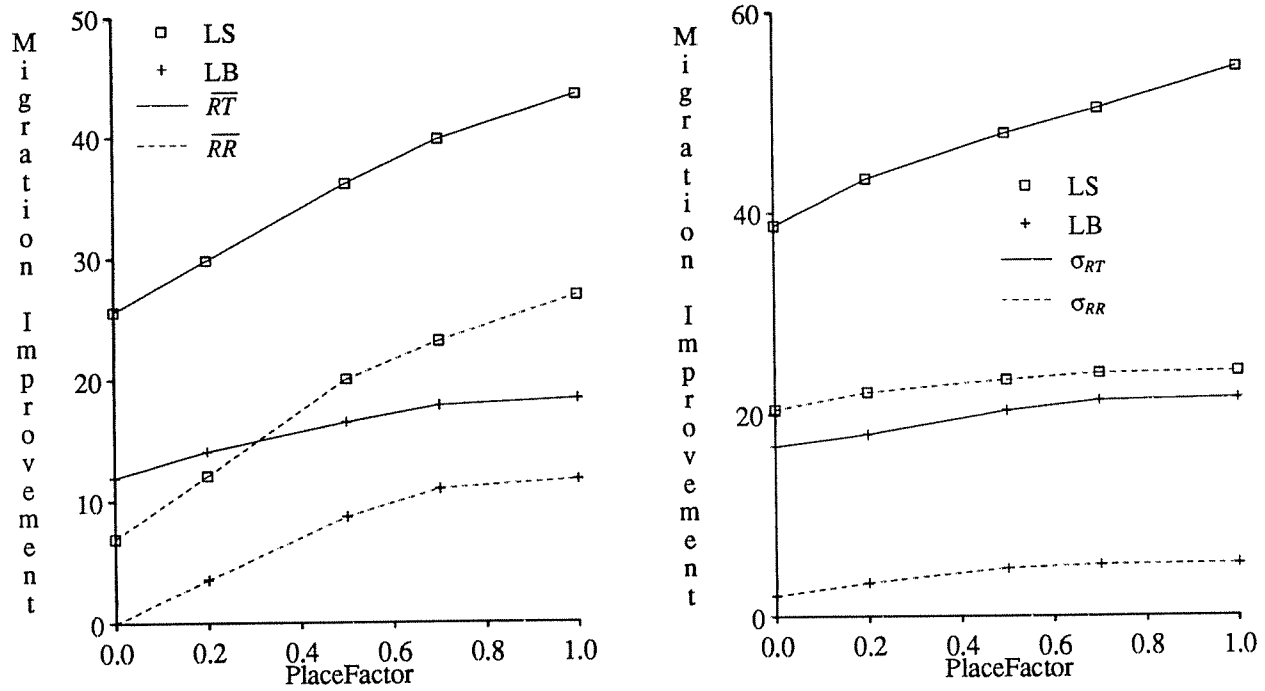

Figure 5.10 Percent migration improvement vs. *PlaceFactor*

more clearly illustrated by figures 5.12 and 5.13, which present results of experiments in which mean process service demand ($\overline{X}$) is varied. The resulting data are plotted in terms of relative migration overhead, with longer $\overline{X}$ corresponding to smaller relative overhead. The results are similar if, instead of varying $\overline{X}$, the mean migration size or the CPU overhead per packet are varied. Figure 5.12 shows that, for placement, the bias in favor of long-running processes (as measured by the difference between $\overline{RT}$ and $\overline{RR}$) decreases with relative overhead, disappearing when overhead becomes negligible. Correspondingly, figure 5.9 shows that this bias decreases with *PlaceFactor*, nearly disappearing at *PlaceFactor* = 0, when load distributing overhead is solely from negotiation. However, figure 5.13 shows that the migration-induced bias follows a different trend, being significant even when overhead is negligible. The reason for this difference is that a bias in favor of long-running processes is *inherent* to migration, rather than arising solely as the result of overhead. Even when overhead is negligible, all processes that are immediately helped by a transfer have executed for some period of time before the transfer, so the mean service demand of these processes is greater than the overall mean. Thus, migration is more heavily biased toward long-running processes than placement.

Figure 5.13 shows that migration can provide significant improvement in performance over a broad range of relative overhead. In no case examined did the addition of migration harm performance. This improvement is largest for SymLS at high levels of overhead. Because an LS algorithm would likely be chosen in preference to an LB algorithm at high overhead *regardless* of whether a migration facility was included, this improvement is more 'important' than the negligible improvement shown by SymLB under such circumstances. The reason for this preference for LS algorithms is that, as shown in the previous section, the additional transfers performed by an LB algorithm beyond those of LS are too costly to be worthwhile under such conditions. As a result, both of our LS algorithms provide better performance than either LB algorithm when relative overhead is high. For sufficiently high overhead, figure 5.12 shows that placement by itself results in performance that is *degraded*, rather than improved. However, the large migration improvement achieved by SymLS under such conditions counteracts this degradation, significantly broadening the range of overheads for which performance is better than without load distributing. For example, SenderLS provides lower $\overline{RT}$ and σ_{RT} than NoLD only

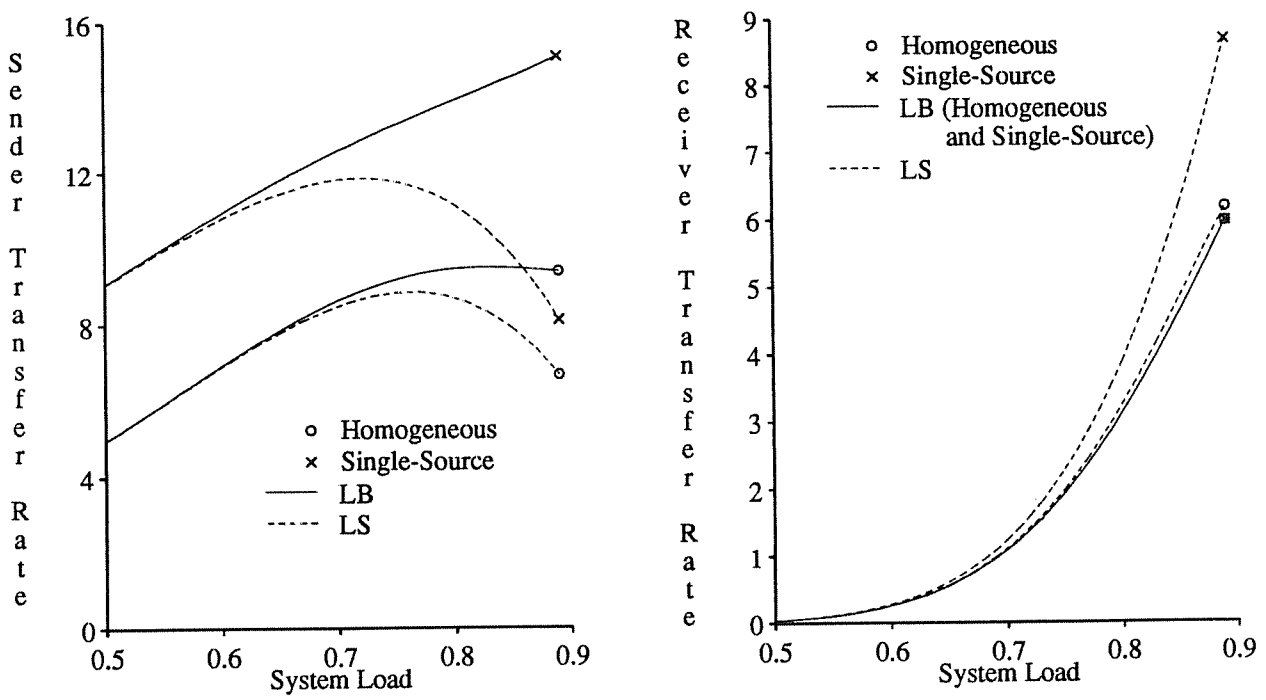

Figure 5.11 Sender-initiated (left) and receiver-initiated (right) transfer rates vs. system load

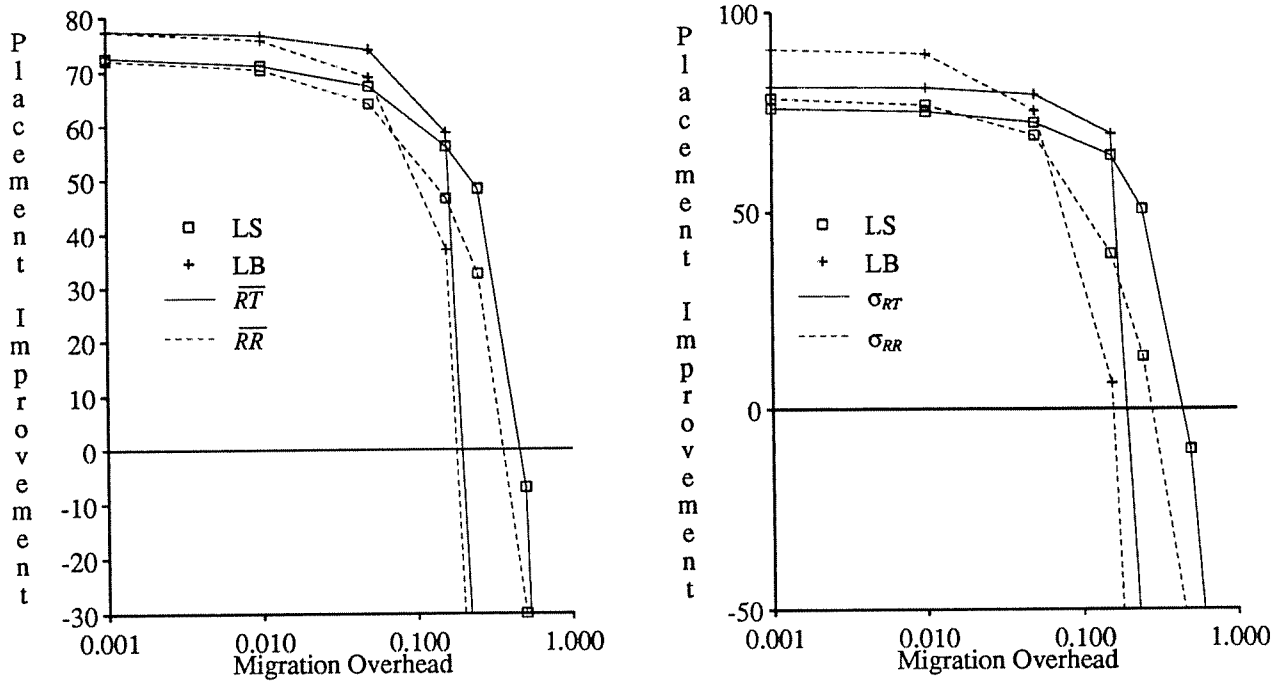

Figure 5.12 Percent placement improvement vs. relative migration overhead (*PlaceFactor* = 0.7)

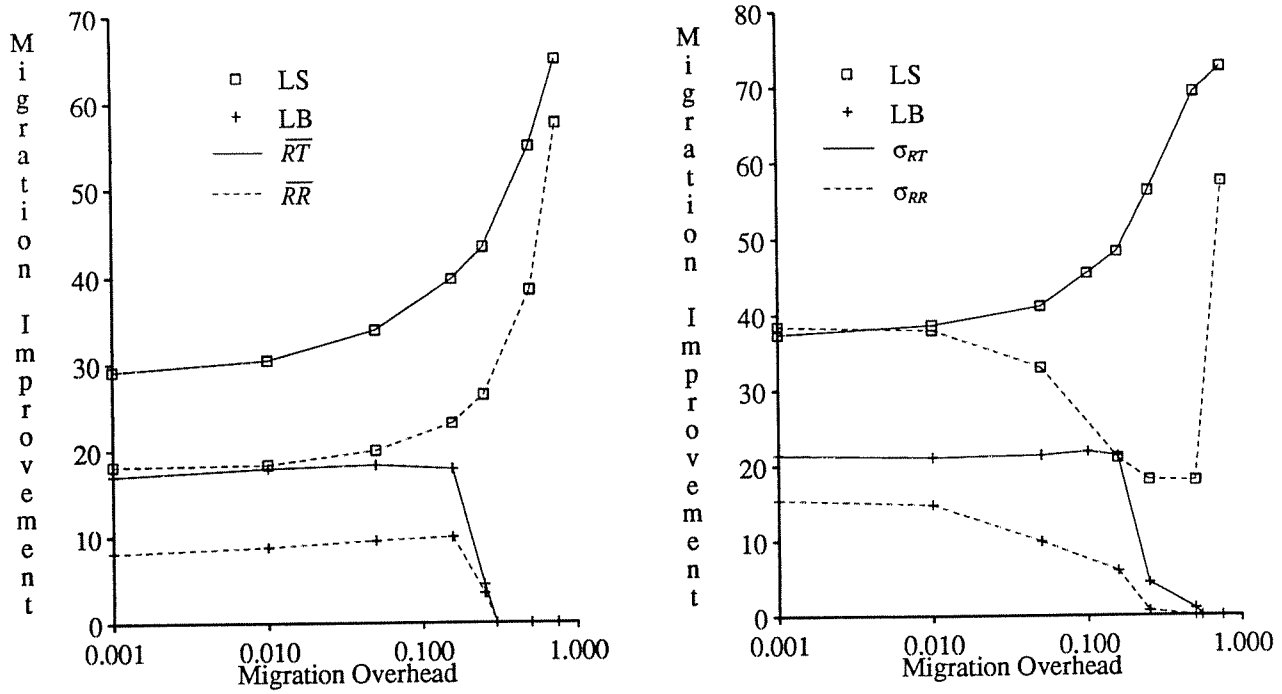

Figure 5.13 Percent migration improvement vs. relative migration overhead (*PlaceFactor* = 0.7)

when relative overhead < 0.5, but SymLS extends this range to 0.8. This extension of the useful range of load distributing makes a migration facility particularly attractive for systems expected to perform well under a broad range of workloads, resulting in widely varying relative overhead.

The large increase in migration improvement resulting from SymLS at high levels of relative overhead, as well as the rapid decrease for SymLB, can be better understood by examining figure 5.14, which plots migration improvement against system load. The trends at high system load are similar to those at high levels of overhead, because high overhead has the effect of increasing the *actual* system load. The increasing migration improvement from SymLS mirrors the results of Eager, Lazowska and Zahorjan [Eager86a], who showed that, while the effectiveness of sender-initiated LS drops off at high loads, receiver-initiated LS becomes increasingly effective with increasing system load. On the other hand, the drop-off in migration improvement from SymLB at high system loads occurs because its placement component allows less variation in load among nodes than that of SymLS. As a result, these migrations are not sufficiently advantageous to be worth the significantly increased CPU contention resulting from their overhead at high system loads. Similar to the case at high levels of relative overhead, the large improvement from SymLS at high system load is more important than the negligible improvement for SymLB. LS algorithms typically outperform LB at sufficiently high system load, where the additional transfers performed by LB increase CPU contention too much to be worthwhile. Following this trend, our simulations show LS to be the strategy of choice under such workloads, with both LS algorithms outperforming either LB algorithm.

Figure 5.14 shows that receiver-initiated migration is particularly useful for systems that, at times, operate at high levels of utilization. Conversely, due to the rapid decline in migration improvement that occurs with decreasing system load for most of the performance indices, migration may be unnecessary for systems that are always lightly utilized.

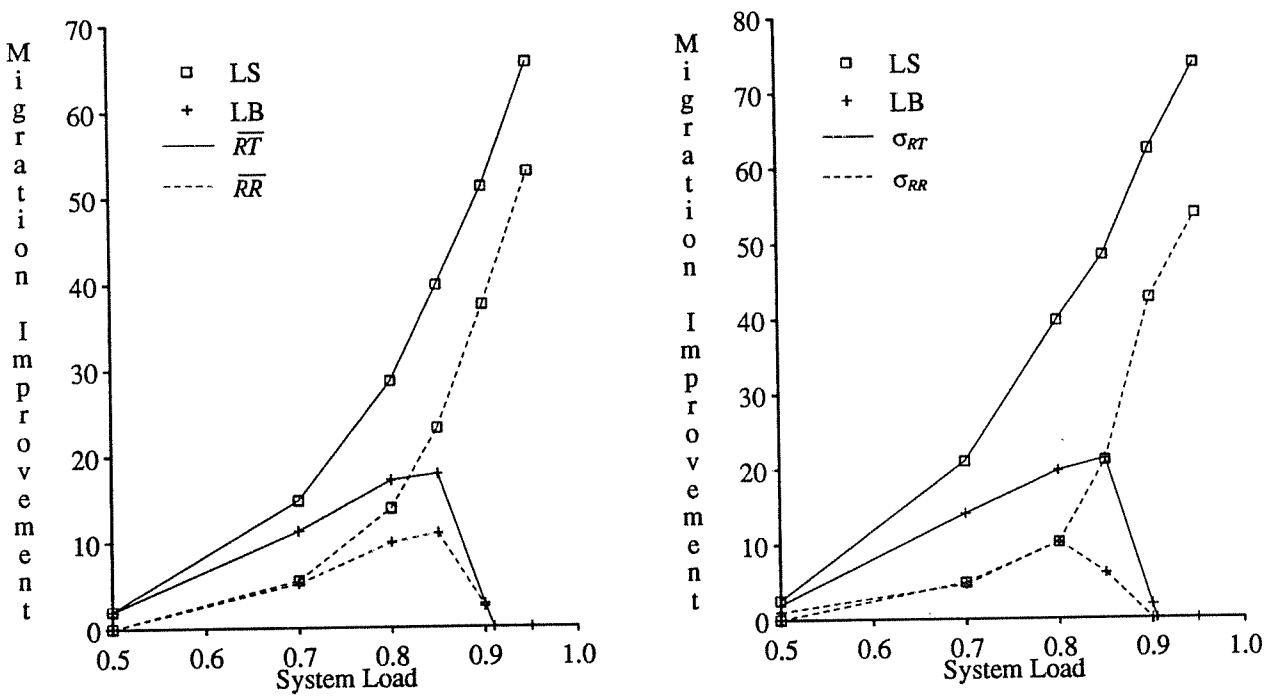

Figure 5.14 Percent migration improvement vs. system load (varying arrival rate with fixed $\overline{X}$, *PlaceFactor* = 0.7)

Still another environment in which receiver-initiated migration provides significant improvement in performance is one in which process initiation rates are heterogeneous. Figure 5.15 plots $\overline{RR}$ (the trends are similar for the other performance indices) against the level of heterogeneity in process initiation rates. The level of heterogeneity is manipulated by varying the portion of the system nodes that are *arrival nodes*. As might be found in a workstation environment, the entire system workload is assumed to initiate at this subset of nodes, with an equal rate at each node, while no processes initiate at the remaining nodes. As predicted in chapter 4, the level of heterogeneity has little effect on the performance of the LB algorithms, though the LS algorithms are strongly affected. For all workloads having sufficiently high levels of heterogeneity, both LB algorithms outperform either LS algorithm, and are thus preferable. However, particularly for distributed systems composed of workstations, ownership rights of individual nodes or groups of nodes may preclude LB, and LS may be necessary. While the performance of the LS algorithms becomes poorer with increasing heterogeneity, the difference in performance between SenderLS and SymLS shows that the addition of a migration component moderates this trend. The increasing migration improvement is more clearly shown by figure 5.16, which plots migration improvement against the portion of nodes that are arrival nodes. These results show that when process initiation rates are heterogeneous, the users of a distributed system using an LS algorithm would perceive a large improvement in performance as a result of a receiver-initiated migration facility.

5.3.3. Comparison with a Similar Study by Eager, Lazowska and Zahorjan

In a recent study, Eager, Lazowska and Zahorjan [Eager88] consider the same problem we have addressed in this section, but predict different results. Through a clever analysis, Eager, et al. predict an upper bound on additional migration improvement in $\overline{RT}$ of $(3 - \sqrt{5}) / 2$, or about 38%. This bound is predicted to be approached when the number of nodes is large (greater than 100) and both the coefficients of variation in process inter-arrival time (C_A) and in process service demand (C_X) are large. In this section, however, we have found migration improvement in excess of 65% (figures 5.13 and 5.14) for a system having only 20 nodes and a workload in which C_A is only 1 and $C_X = 5.32$.

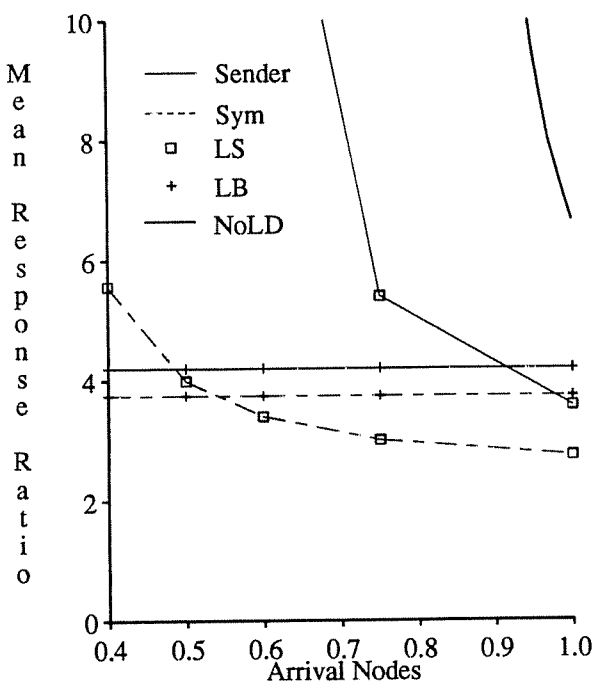

Figure 5.15 $\overline{RR}$ vs. portion of nodes that are arrival nodes (*PlaceFactor* = 0.7)

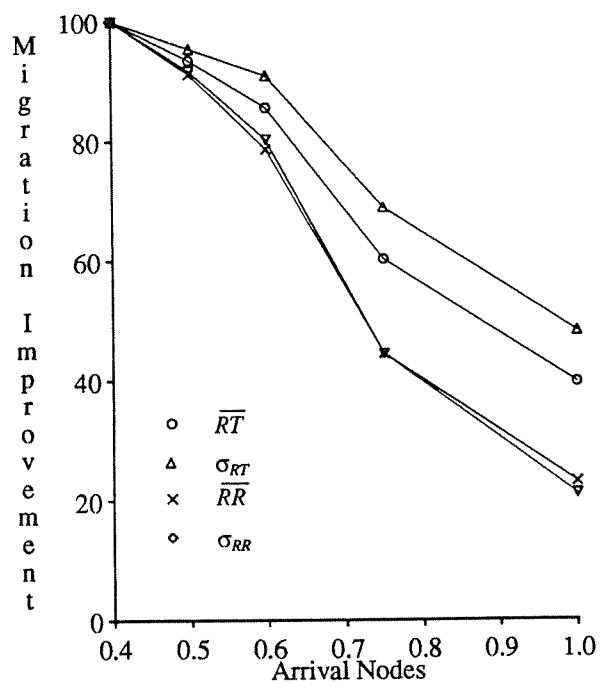

Figure 5.16 Percent migration improvement for SymLS vs. portion of nodes that are arrival nodes (*PlaceFactor* = 0.7)

The difference between the results of these two studies is accounted for by the greater amount of detail included in our simulation study, relative to the analytic study. Two such details, in particular, largely account for the difference. First, while Eager, et al. consider the effect of migration overhead, placement overhead is assumed to be negligible. As a result, the optimal placement strategy is always load balancing. In contrast, the study in this section considers non-negligible placement overhead. Because not all placements may be cost-effective under such overhead, load balancing may not always be optimal. Consequently, we consider algorithms using load sharing for placements (SenderLS and SymLS), as well as those using load balancing (SenderLB and SymLB). Second, the study by Eager, et al. assumes negotiation overhead to be negligible. As a result, a load distributing algorithm is able to make decisions based on perfectly accurate information from every node in the system. In contrast, the study described in this section includes negotiation overhead. Consequently, state information gathered from other nodes may be obsolete. Furthermore, recognizing that gathering state information from every node in the system may not be cost-effective, only a subset of the nodes are polled.

To illustrate the effect of this greater level of detail, we compare the performance of algorithms that poll all nodes with algorithms that poll only a subset of nodes, and load balancing algorithms with load sharing algorithms. These algorithms are compared over a wide range of negotiation and transfer overhead. The algorithms considered are identical to those studied earlier in this section, except that *PollLimit* is varied:

	SendProb	*RecvProb*	T_{Rmax}	*PollLimit*
NoLD	0	0	0	0
SenderLS-20	1	0	0	20 (all nodes)
SenderLB-20	1	0	∞	20 (all nodes)
SymLS-20	1	1	0	20 (all nodes)
SymLB-20	1	1	∞	20 (all nodes)
SenderLS-5	1	0	0	5
SenderLB-5	1	0	∞	5
SymLS-5	1	1	0	5
SymLB-5	1	1	∞	5

Figure 5.17 plots $\overline{RT}$ for each of these algorithms against relative migration overhead, manipulated via $\overline{X}$. Both axes in this plot are logarithmic. The remaining performance indices follow trends similar to $\overline{RT}$. Because relative migration overhead is manipulated via X for this figure, an increase in relative overhead indicates an increase in the significance of negotiation overhead as well as transfer overhead.

Among the algorithms described above, SenderLB-20 and SymLB-20 are most similar to those studied by Eager, et al. These algorithms are shown by figure 5.17, however, to be viable in only a narrow region of relative migration overhead. The remaining algorithms, implementing reduced degrees of load distributing, provide better performance under most conditions. First, the algorithms that poll only a subset of nodes generally provide better performance than those than poll all nodes in the system. These latter algorithms have an advantage only when relative overhead is very low. Second, the LS algorithms often perform better than the LB algorithms. Consistent with section 5.2, the LB algorithms have an advantage only at low levels of overhead.

Figure 5.18 shows that these reductions in the degree of load distributing, together with improving performance under many conditions, also increase the ability of migration to additionally improve performance. In this figure, migration improvement in $\overline{RT}$ is plotted against relative migration overhead. The migration improvement that is achieved under assumptions similar to those of Eager, et al. is represented by the line for LB-20. Consistent with their predictions, improvement is small when overhead is negligible and decreases slightly with increasing overhead. Replacing LB with LS, however, substantially increases the magnitude of additional migration improvement, as shown by the results for LS-20. In addition, contrasting with LB-20, improvement increases with increasing overhead. As discussed earlier in this section, these two effects result

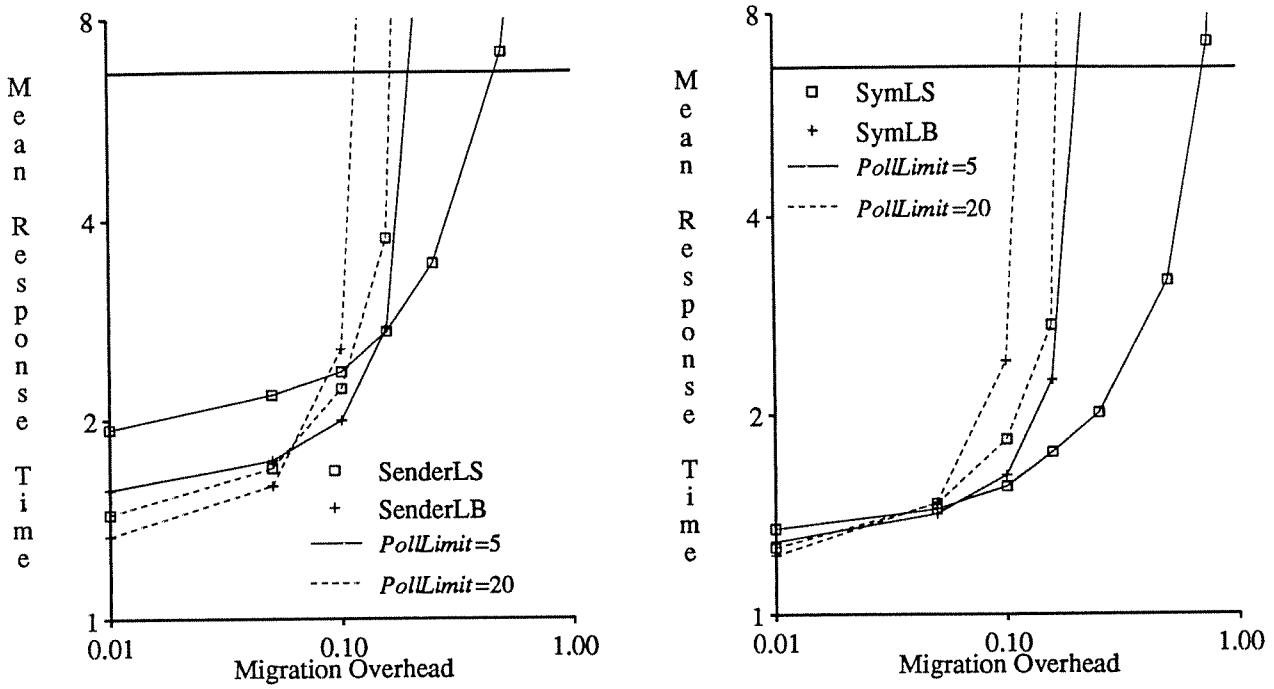

Figure 5.17 $\overline{RT}$ vs. relative migration overhead for sender-initiated algorithms (left) and symmetrically-initiated algorithms (right) ($PlaceFactor = 0.7$)

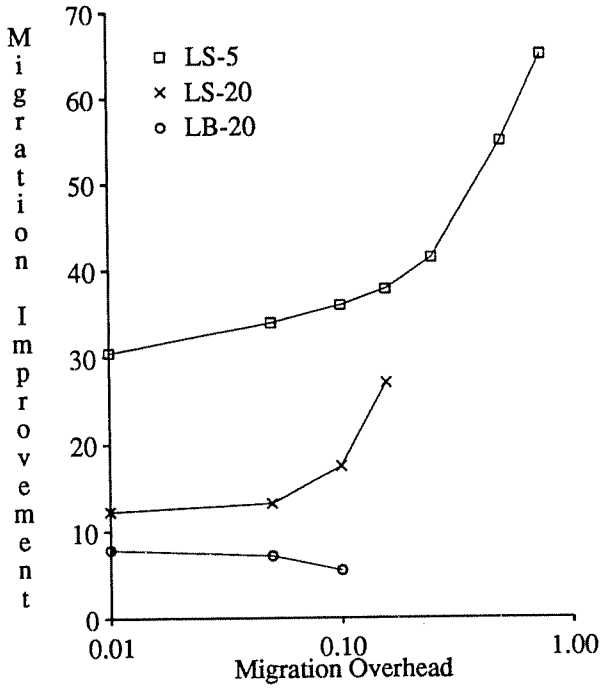

Figure 5.18 Percent migration improvement in $\overline{RT}$ vs. relative migration overhead

from the greater reliance of LS on migration, relative to LB, and on the decreasing effectiveness of the additional transfers pursued by LB with increasing relative overhead. Finally, the results for LS-5 show that polling a subset of nodes instead of all nodes, again, considerably increases migration improvement (these results are the same as those shown for LS in figure 5.13). This increase occurs because a reduction in the amount of information available to load distributing increases the likelihood of a placement 'mistake', such as failing to locate a suitable receiving node, even though one exists. Migration, then, has greater opportunity to correct these mistakes. Because this combination of polling only a subset of nodes and using LS in place of LB provides the best performance for all except the lowest levels of relative overhead (figure 5.17), this large additional migration improvement shown for LS-5 is significant over a wide range of conditions.

In summary, rather than contradicting the results of Eager, et al., our study validates and extends their results. By including greater detail in our study, we are able to consider a wider range of distributed system. We find that, for those systems in which negotiation and transfer overhead are significant, as is generally the case, the algorithms studied by Eager, et al. are often not viable. When the algorithms that provide the best performance for these systems are considered, the addition of a migration facility may allow substantial performance improvement beyond that available through placement.

5.3.4. Summary and Conclusions

In this section, we have considered whether the addition of a process migration facility to a distributed scheduler already capable of process placement is a worthwhile investment. This question is particularly important to distributed operating system developers, because implementation of a migration facility is likely to be much more expensive than a placement mechanism. One must question whether this investment can provide significant gains in performance. Since, perhaps, the most obvious use for migration is to allow receiver-initiated process transfers, this section has focused on the performance gains available through such use.

We have found that, while placement alone is capable of large improvement in performance, the addition of receiver-initiated migration, in many cases, achieves considerable additional improvement. This improvement is particularly important, because it broadens the range of workload conditions under which load distributing improves, rather than degrades, performance. The magnitude of this improvement depends on the workload characteristics of the system, as well as on the overhead of migration relative to placement, which is influenced by a variety of design features of the system. The key characteristic of a distributed system that can profit from the addition of a migration facility is a high level of utilization, including the overhead of load distributing. Although most systems are not specifically designed to operate at high utilization over the long term, many are exposed to such workloads for short periods of time. While these periods are short with respect to the lifetime of the system, they may be long enough to significantly affect the performance perceived by the users and should, therefore, not be ignored. In addition, performance gains may increase when process initiation rates are heterogeneous. Again, even for systems in which heterogeneity is intermittent, it may persist long enough to have a strong impact on user performance. It is important, then, to understand the effects of such conditions and to develop distributed scheduling algorithms that perform well, even under 'abnormal' circumstances which may otherwise threaten the stability of the system.

We have shown that receiver-initiated migration has an inherent bias toward reducing the wait times of long-running processes. As a consequence, receiver-initiated migration is more capable of improving $\overline{RT}$ than $\overline{RR}$, and improves σ_{RT} more than σ_{RR}, though large improvements in these latter indices may also be achieved. Also, we have shown that migration improves the performance of load sharing algorithms more than that of load balancing algorithms, though again, significant improvement may also be achieved for load balancing.

In summary, a wide range of system characteristics and workload conditions exist under which the additional improvement provided by receiver-initiated migration would be obvious to the users of the system. In addition, no conditions have been discovered in which the addition of receiver-initiated migration harms performance. While not as essential to load distributing as placement, a migration facility is potentially a worthwhile investment.

5.4. Sender, Receiver or Symmetrically-Initiated Load Distributing?

The previous section noted that many load distributing actions can be categorized according to the goals of the nodes that initiate them. An action is referred to as *sender-initiated* if it is originated by a node that is the

potential source of a process transfer. For example, a negotiation session is sender-initiated if it is begun by a node seeking to transfer a process to another node. Conversely, a *receiver-initiated* action is originated by a node seeking a process that can be transferred to it. A method of reducing the degree of load distributing pursued by an algorithm is to reduce only the rates of sender-initiated actions, or only receiver-initiated actions. At the extremes, a sender-initiated algorithm is restricted to performing *only* sender-initiated actions, while a receiver-initiated algorithm performs only receiver-initiated actions. Load distributing algorithms, then, fall on a continuum ranging from receiver-initiated, through symmetrically-initiated, to sender-initiated algorithms.

The effect on performance of reducing the degree of load distributing along this continuum differs depending on whether the reduction is aimed at sender-initiated or receiver-initiated actions. Two studies that compare sender-initiated with receiver-initiated algorithms [Chang86, Eager86a] show that which of these extremes provides the best performance depends on dynamic characteristics of the workload. Neither of these studies, however, examines the performance of these extremes relative to the center of the continuum, symmetrically-initiated load distributing. If only performance is considered, the previous section indicates that the first of these extremes, sender-initiated load distributing, has little to recommend it. Even when migration carries much greater overhead than placement, the symmetrically-initiated algorithm studied performs better than the sender-initiated algorithm over all system environments examined. On the basis of that evidence, we turn our attention from the portion of the continuum lying between sender and symmetrically-initiated load distributing, to that lying between symmetrically and receiver-initiated load distributing. Because of the relationships between receiver-initiated actions and process migration, and between sender-initiated actions and migration, we, in effect, switch our scrutiny from process migration to process placement. The goal is to illustrate the effect of dynamic characteristics of the workload in determining the point on the sender/symmetrical/receiver-initiated continuum that results in the best performance, and the range that maintains system stability.

The simulation assumptions for this section are identical to those of the previous section. Because, under PollGen, receiver-initiated actions are performed only by idle nodes, receiver-initiated load balancing is not allowed. Therefore, we examine the performance of only a single receiver-initiated version of PollGen, in addition to SymLS, SymLB and NoLD from the previous section:

	SendProb	*RecvProb*	T_{Rmax}
NoLD	0	0	0
SymLS	1	1	0
SymLB	1	1	∞
Recv	0	1	-

To maximize our skepticism for Recv, which relies solely on migration, the results presented assume that PlaceFactor is near zero. Placement is assumed to require only a single packet to be sent to the receiving node.

To illustrate the effect of the offered system load in determining the point on the continuum that results in the best performance, figure 5.19 plots $\overline{WT}/\overline{X}$ against offered system load, which is manipulated by varying λ with fixed $\overline{X}$. To avoid redundancy, $\sigma_{WT}/\overline{X}$, $\overline{WR}$ and σ_{WR} are not plotted, as they show similar trends. At low system loads, a potential sending node has little trouble finding a suitable receiver, since many nodes are idle. A potential receiver, however, is unlikely to find a suitable sender, since few nodes are overloaded. As a result, SymLS and SymLB perform better than Recv in this region. Reducing the degree of load distributing by avoiding sender-initiated actions is not advantageous at such loads. In contrast, at high system loads, a receiving node can easily find a suitable sender, since many nodes are overloaded. A potential sender under SymLS, however, is unlikely to find a suitable receiver, since few nodes are idle. The overhead of the consequent high rate of failed negotiations hurts performance. Under SymLB, potential senders are able to find suitable receivers, but the resulting transfers are often not cost-effective. As a result, even though migration carries much greater overhead than placement, Recv performs better than either of the symmetrically-initiated algorithms in this region. Under such workloads, reducing the degree of load distributing by avoiding sender-initiated actions can substantially improve performance.

Figure 5.20, which plots $\overline{WT}/\bar{X}$ against relative migration overhead, which is varied through $\bar{X}$, shows the effect of $\bar{X}$ in determining the best point on the continuum. Again, $\sigma_{WT}/\bar{X}$, $\overline{WR}$ and σ_{WR} show similar trends. The trends in this plot are similar to those of figure 5.19, and for the same reasons, because an increase in migration overhead has the effect of increasing the *actual* system load. While SymLS and SymLB provide significant improvement in performance relative to Recv at low relative overhead, they result in instability for relative overhead > 0.75. Recv, however, continues to improve performance at high relative overheads and maintains stability beyond relative overhead $= 10$.

In summary, even when the overhead of placement is much less than that of migration, reducing the degree of load distributing by avoiding sender-initiated actions can substantially improve performance under many workload conditions. In contrast, the previous section found no workload under which avoiding receiver-initiated actions improves performance. We conclude that sender-initiated actions can pose a greater threat to performance and to system stability than receiver-initiated actions. Similar to the NoLD/LS/LB continuum, the point on the continuum between symmetrically-initiated and receiver-initiated load distributing that provides the best performance and the range that results in stability are dependent on characteristics of the workload that are subject to change over time.

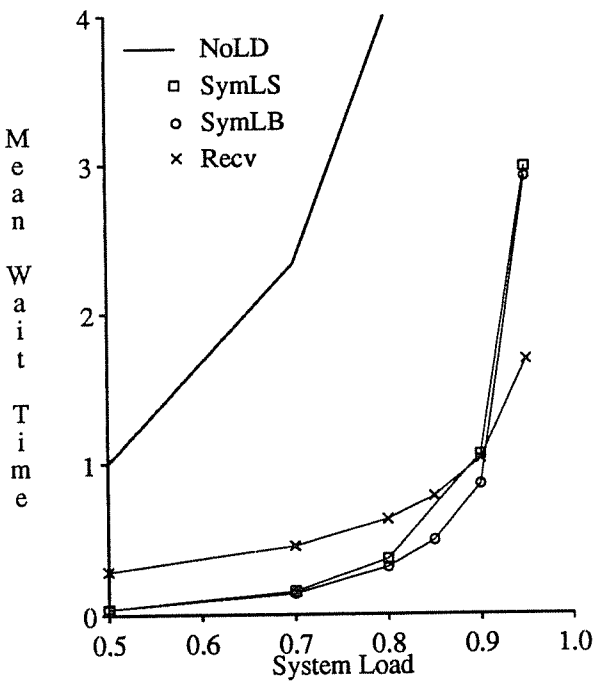

Figure 5.19 $\overline{WT}/\bar{X}$ vs. offered system load (varying λ with constant $\bar{X}$)

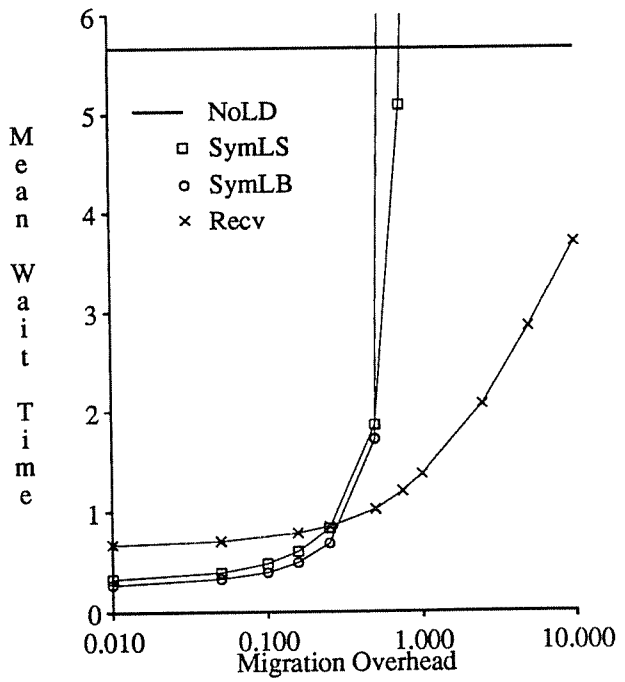

Figure 5.20 $\overline{WT}/\bar{X}$ vs. relative migration overhead (varying $\bar{X}$)

5.5. Summary and Conclusions

In practical distributed systems, the overhead of load distributing can not be considered negligible. As a result, the performance cost incurred by some load distributing actions may exceed the benefit. If too many of these ineffective actions are initiated, performance is degraded, rather than improved, and the stability of the system is threatened. To avoid such actions, a load distributing algorithm may find it necessary to modify its strategy, reducing its degree of load distributing by making compromises in how strictly its strategy is followed.

In this chapter, we have considered three axes along which the degree of load distributing can potentially be reduced. The first of these axes forms a continuum ranging from load balancing, through load sharing, to no load distributing. We have found that no algorithm implementing a single point on this continuum can guarantee stability for the system, much less provide the best performance, over the wide range of conditions that may occur in a distributed system over the course of a day.

The second axis is based on the observation that process placement may carry less overhead than migration. Focusing a reduction in the degree of load distributing on migration, rather than placement, has the largest effect on the overhead of load distributing. This second axis, then, forms a continuum ranging from the fullest use of migration required by the chosen load distributing strategy to no migration. Assuming that local scheduling is preemptive, providing a burst of service to newly-arrived processes, loose relationships exist between placement and sender-initiated load distributing, and between migration and receiver-initiated load distributing. Placements must be sender-initiated, though migrations may be either sender or receiver-initiated. Our study of this continuum focused on two points: the no-migration extreme, relying solely on sender-initiated placement, and an intermediate point, augmenting placement with those migrations that are receiver-initiated. No system environment was identified in which these migrations are a liability to performance. Many environments were identified, however, in which migration can make substantial improvements in performance. We conclude that, while reduction in degree along this axis may in the future be shown to hold some advantage, avoiding migration entirely holds little promise for improving performance.

The third axis depends on a categorization of load distributing actions into those that are sender-initiated and those that are receiver-initiated. Because of the relationships between receiver-initiated transfer and process migration, and between sender-initiated transfer and process placement, this axis is not orthogonal to the second axis, though neither is it an extension of the second axis. This section compared the performance of algorithms at the center of this continuum, implementing symmetrically-initiated load distributing, with algorithms at the two extremes: sender-initiated and receiver-initiated load distributing. Though the symmetrically-initiated algorithms were shown to provide significantly better performance under some conditions than the other types of algorithms, the receiver-initiated algorithm provides the best performance under other conditions. In particular, the receiver-initiated algorithm was found to improve performance under some conditions that result in instability for the other algorithms. No system environment was identified, however, that allows the sender-initiated algorithms to hold an advantage over either of the other two algorithm types. We conclude that, even if migration carries much greater overhead than placement, sender-initiated load distributing actions pose a greater threat to performance and to system stability than receiver-initiated actions. Focusing a reduction in the degree of load distributing on the portion of this continuum ranging from symmetrically-initiated to receiver-initiated load distributing holds the greatest promise for improving performance. Similar to the first continuum, however, the point on this continuum that provides the best performance and the range that results in stability are dependent on dynamic characteristics of the workload.

In summary, two continua have been identified along which reducing the degree of load distributing may result in substantially improved performance. The point on the plane formed by these continua that provides the best performance, and the area that maintains stability, are dependent on a variety of characteristics of the system and its workload that are subject to change over time. We conclude that an algorithm implementing a static point on this plane is insufficient to provide acceptable performance over the wide range of conditions that may occur in a distributed system.

CHAPTER 6

Adaptive Load Distributing

6.1. Introduction

In chapter 5, we found that the degree of load distributing that provides the best performance, and the range in degree that maintains stability for the system, are dependent on many characteristics of the system and its workload that are subject to change over time. The goal of this chapter is to develop a framework for constructing load distributing algorithms that adapt in degree, allowing them to be robust in the face of the wide range of conditions that may occur in a distributed system over the course of a day. Rather than design a single load distributing algorithm that features such adaptivity, our approach is to identify and build the tools necessary to add adaptivity to a broad range of algorithms.

We begin by describing a framework for adding adaptivity to load distributing algorithms. As an example, we then use this framework to extend the PollGen algorithm to include adaptivity. The ability of this enhanced algorithm to adapt to the system environment is evaluated through simulation. As a second example, we add adaptivity to an algorithm bearing little resemblance to PollGen: the Random algorithm described be Eager, Lazowska and Zahorjan [Eager86]. Again, the resulting adaptive algorithm is compared with the original algorithm through simulation. Finally, we summarize our results and draw conclusions.

6.2. A Framework for Constructing Adaptive Load Distributing Algorithms

In this section, we describe a framework for constructing load distributing algorithms that are adaptive to the distributed system environment. This framework is sufficiently general to be applicable to many load distributing algorithms. We begin by clarifying the notion of the degree of load distributing performed by an algorithm by defining the *tolerance of an algorithm*, and by identifying several methods for manipulating the tolerance. Continuing, we derive a set of cost-benefit estimates for load distributing actions that take into account the system environment. These estimates are not tied to a specific load distributing algorithm, but are applicable across a broad range of algorithms. Using these estimates to discern effective from ineffective actions, the tolerance can be manipulated so that the actions that are cost-effective in the existing system environment are allowed, while ineffective actions are not. By doing so, the resulting tolerance adapts to changes in the system environment.

6.2.1. The Tolerance of a Load Distributing Algorithm

Because load distributing actions, such as negotiation and process transfer, require time to complete, no load distributing algorithm can avoid deviation from its chosen strategy. No load sharing algorithm can completely avoid periods in which the system is *unshared*, with some node lying idle while processes wait for service. Nor can a load balancing algorithm avoid *unbalanced* periods in which some pair of nodes differs in load by more than one. We refer to the degree to which such states are allowed by a load distributing algorithm as its *tolerance*. A tolerance may be specified in terms of the following:

Magnitude
: How far the system may deviate from the chosen ideal (such as a shared or balanced state) before corrective action is taken. Such corrective action includes initiation of negotiation or a process transfer.

Time
: How long the system may deviate from its ideal before corrective action is taken.

The *inherent tolerance* is the strictest tolerance that can be attained, given the bandwidths and levels of contention for the resources required for load distributing. Physical characteristics of the system, such as CPU and communication device bandwidths, together with workload characteristics, such as system load and the level of interprocess communication, set this bound on how strict a tolerance can be maintained. Identifying this tolerance and the algorithm that achieves it would certainly be difficult, and may be intractable in the general case. However, achieving this tolerance is not a suitable goal, because it generally does not optimize performance. Some load distributing actions, while helping to maintain a strict tolerance by keeping the system in a balanced or shared state, may damage performance, possibly threatening the stability of the system. Such actions are not *cost-effective,* since the performance *cost* they incur is greater than the *benefit* they impart. To achieve acceptable performance and to maintain system stability, a load distributing algorithm may find it advantageous to hold itself to a standard more lax than the inherent tolerance. Relaxing the tolerance reduces the transfer rate or negotiation rate, reducing the cost of load distributing by decreasing contention for resources shared with user processes. Under conditions that would result in the overall cost of load distributing being greater than its benefit if the strictest tolerance were adhered to, a relaxed tolerance may cause greater reduction in cost than benefit. As a result, the load distributing algorithm may improve, rather than degrade, performance.

Several features of the design of a load distributing algorithm can be identified as contributing to its tolerance. These features include:

- Identification of nodes that initiate negotiation or transfer. For example, the strictest tolerance for load balancing requires a node to initiate negotiation whenever a transfer might be necessary, which is whenever a process completes or whenever a process arrives at a busy node. A relaxed tolerance may require the process completion to result in the node becoming idle, or the process arrival to be to a node that already has some backlog of work. Relaxing the tolerance in this way neglects those negotiations that are likely to lead to the least beneficial transfers, while allowing those that will likely be most beneficial.
- Identification of a suitable transfer partner. For example, the strictest tolerance for load balancing requires only that the number of executing processes at the sender and receiver differ by at least two. A relaxed tolerance may require these loads to differ by at least three. Relaxing the tolerance in this way neglects those transfers that have the least benefit, while continuing to perform those resulting in the greatest benefit.
- Temporal relationships between negotiation sessions, process transfers or the exchange of information may be specified. For example, the strictest tolerance for load balancing requires that negotiation begin immediately whenever a transfer might be necessary. A relaxed tolerance may require a minimum period to elapse between negotiation sessions. Alternatively, a relaxed tolerance may allow only potential sending nodes to initiate negotiation.
- Negotiation, transfer or remote information maintenance may be probabilistic. For example, when all other criteria are met, a load distributing algorithm having a relaxed tolerance may initiate negotiation with probability $P < 1$.
- The size of the subset of nodes that exchange information or negotiate process transfers with a given node may be specified. For example, the strictest tolerance may require negotiation to include every node in the system, if no suitable transfer partner is found earlier. A relaxed tolerance may allow negotiation to quit after some limited set of nodes has been considered, if no suitable node can be found.

These design features provide a means for manipulating the tolerance of a load distributing algorithm. As such, they provide a first step toward an *adaptive* tolerance. If an algorithm is designed so that some subset of these features can be modified dynamically, and if suitable decision-making procedures can be found to control this modification, an adaptive tolerance can be implemented.

6.2.2. Estimating Cost-Effectiveness

The tolerance of a load distributing algorithm is best suited to its system environment if it allows only those actions that are cost-effective to be performed. The basis for an adaptive tolerance, then, is the ability to discern actions that are cost-effective from those that are ineffective in any system environment. Our approach

to identifying cost-effective actions is to estimate their effects on performance before they are undertaken. These estimates can then be used to manipulate the tolerance in such a way that cost-effective actions are performed, while ineffective actions are not.

In this section, we develop estimates for the performance cost and benefit of load distributing actions. We will pursue estimates that are not overly tailored to a specific load distributing algorithm, so that they can be used across a broad range of algorithms. These estimates must be sufficiently accurate to effectively control an adaptive tolerance. However, this need for accuracy must be balanced with the need for *efficiency*, since the overhead of calculating these estimates will further add to the performance cost of load distributing. This need for efficiency leads us to develop estimates that are particularly simplistic, at the cost of accuracy. Second-order effects on performance are generally ignored. However, we show in the next section that these estimates are sufficiently accurate to control an adaptive tolerance.

We begin our search for suitable estimates by categorizing the short term, or first order, effects that load distributing has on performance. Long term effects are often dependent on the specific load distributing algorithm used and are of smaller magnitude than short term effects, so will not be considered.

Load distributing can decrease the amount of time user processes must wait for allocations of CPU bandwidth. For example, by moving a process from a node where it must wait for service to an idle node that can provide immediate service, CPU queuing delays are reduced for both the process moved and the processes left behind at the sender. This reduction in queuing delays we term the *benefit* of load distributing. However, in accomplishing this reduction, load distributing itself requires the use of resources. These resources include CPU and communication device bandwidth, which are shared with user processes. Because of possible contention for these resources, load distributing can increase the amount of time user processes must wait for resource allocations. This increase in the queuing delays of user processes is the *cost* of load distributing. Both benefit and cost, then, are changes in user process queuing delays caused by load distributing. The overall effect that load distributing has on performance, which we refer to as load distributing *advantage*, is the difference between its benefit and cost.

CPU cost is the increase in CPU queue delays experienced by users processes as a result of load distributing resource use. CPU cost can be divided into *negotiation cost* and *transfer cost*. For load distributing algorithms having negotiation or remote information maintenance components, CPU negotiation cost is the increase in CPU queue delays at the nodes sending and receiving negotiation and remote information messages, caused by the CPU overhead of processing these messages. CPU transfer cost, on the other hand, consists of some or all of the following:

- A process being transferred is delayed during some portion of the time required to complete the transfer. This delay can be considered a CPU queuing delay, as the process is awaiting access to a CPU resource.
- The processes at the sending and receiving nodes of a process transfer are delayed, due to the CPU service demand of process selection and transfer.
- The processes already residing at the receiving node are delayed, due to the addition of the transferred process to the CPU queue.

Communication cost is the other primary component of load distributing cost. This cost is the increase in queuing delays at the communication device caused by the message traffic generated in the course of load distributing.

The **benefit** of load distributing is the sum of the benefits imparted by individual process transfers. The benefit resulting from a specific transfer consists of:

- When a process is transferred, the reduction in CPU queue length at the sending node results in decreased CPU queue delays for the processes left behind.
- CPU queue delays for the transferred process are reduced if it joins a shorter CPU queue at the receiving node.

6.2.2.1. CPU Transfer Cost, Benefit and Advantage

In this section, we develop measures that can be efficiently calculated for the CPU cost and benefit of performing a specific process transfer. CPU cost and benefit will be measured in terms of expected *future* wait time and wait ratio, which are the portions of a process' wait time and wait ratio that are experienced in the future. For the wait ratio estimates, we assume that the local scheduling discipline is Processor Sharing.

While all processes residing in the system, as well as many future processes, may be affected by a process transfer, three groups of processes are primarily and immediately affected: the process actually being transferred, the processes left behind at the sending node, and the processes already residing on the receiving node. Each of these groups has a unique perspective on the cost and benefit of a transfer.

In deriving estimates for CPU transfer cost, benefit and advantage, we use the following notation:

r_{com}	Response time of a packet on the communication device
μ_{com}	Service rate of the communication device
ρ_{com}	Utilization of the communication device
$\overline{S}_p$	Mean packet size
s	Amount of data that must be transmitted to accomplish a transfer, in packets
s_a	Amount of data remaining to be transferred *after* transferring process has blocked
OV_d	CPU service demand for disconnecting a process
OV_c	CPU service demand for reconnecting a process
OV_s	CPU service demand for sending a packet on the communication device
OV_r	CPU service demand for receiving a packet from the communication device
t_b	Time during which transferred process is blocked
n_s	CPU queue length at the sending node before the transfer is performed
n_r	CPU queue length at the receiving node before the transfer is performed
x'_t	Residual service demand of the transferred process
x'_s	Residual service demand of a process left behind at the sending node
x'_r	Residual service demand of a process residing at the receiving node before the transfer
M	(Defined by equation 6.4)
$I\%$	Percent of processes included in the wait ratio cost estimates (equation 6.5)

The Transferred Process

The transferred process is delayed waiting for CPU bandwidth while it is disconnected at the sending node, sent to the receiving node via the communication device[1] and reconnected at the receiving node. Disconnection and reconnection involve such actions as notifying servers and cooperating processes of the new location of the transferred process. The period of time during which the transferred process is blocked while the transfer is performed, then, is:

$$t_b = OV_d + s\,OV_s + s_a\,r_{com} + OV_c \tag{6.1}$$

The purpose of s_a in this expression is to allow for distributed operating systems, such as the V-System [Theim85], that allow a transferring process to continue executing while some portion of it is sent to the receiving node. While r_{com} for each packet is unknown in advance, it can be approximated by assuming that message interarrival times and communication device service demands are independently exponentially distributed, and that scheduling of the communication device is work-conserving [Klein76]:

$$E(r_{com}) \approx \frac{1}{\mu_{com}(1-\rho_{com})}$$

where μ_{com} = *bandwidth of communication device* / $\bar{S}_p$. Using this approximation together with equation 6.1 provides an expression for transfer cost, in terms of wait time and from the perspective of the transferred process. The increase in expected future wait time is equal to the expected time during which a process is blocked while it is transferred:

$$TC_{t,wt} = E(t_b) = OV_d + s\,OV_s + \frac{s_a}{\mu_{com}(1-\rho_{com})} + OV_c \tag{6.2}$$

Since the time during which a process is blocked while it is transferred can be assumed to be independent of its residual service demand, the expected increase in the wait *ratio* of a transferred process is:

$$E(t_b / x'_t) = E(t_b)\,E(1/x'_t)$$

If process service demands are assumed to follow an exponential distribution, the distribution of residual service demands is identical to the overall service demand distribution, so:

$$= E(t_b)\,E(1/x) \tag{6.3}$$

This assumption of exponentially distributed service demands may evoke some protest. As noted in chapter 3, hyperexponential distributions have been found to model process service demands more accurately than exponential distributions. However, this exponential assumption greatly simplifies our estimates, and we are willing to sacrifice some accuracy for the sake of efficiency. Unfortunately, this assumption does not bring our problems to an end, since the latter term of equation 6.3, $E(1/x)$, is shown in appendix B to be infinite for exponential, hyperexponential or Erlang distributions. However, if the processes having the shortest service demands are trimmed, the *conditional* expectation, $E(1/x \mid x > C)$, is finite. For example, to trim the shortest 1% of processes from an exponential distribution, $C \approx .01\bar{X}$, and $E(1/x \mid x > .01) \approx 4/\bar{X}$. To generalize our estimates, we use the following notation:

$$M = \bar{X}\,E(1/x \mid x > C) \tag{6.4}$$

Several values for M can be found in table B.1 in appendix B. Also, we refer to the percent of processes included in those cost estimates that are in units of wait ratio as:

1. We assume that the processing time incurred by the destination CPU in receiving the packets containing the process transfer is concurrent with the processing time incurred by the source CPU in sending them.

$$I\% = 100\left[1 - (1/\bar{X})\int_{C}^{\infty} e^{-x/\bar{X}}\,dx\right] \tag{6.5}$$

Based on the above and equation 6.3, our measure of transfer cost, in units of wait ratio and from the perspective of the transferred process, is:

$$TC_{t,wr} = (M / \bar{X})\, TC_{t,wt} \tag{6.6}$$

The cost of the transfer must be weighed against its benefit. From the perspective of the transferred process, this benefit is the decrease in wait time or wait ratio that results if the CPU queue at the receiving node is shorter than that at the sender. An approximate expression for the difference between the future wait *ratio* that the process would have perceived had it remained at the sending node, and that enjoyed at the receiving node once the transfer has completed, is:

$$B_{t,wr} = n_s - 1 - n_r \tag{6.7}$$

and the difference in wait *time* for this process is:

$$B_{t,wt} = \bar{X}\, B_{t,wr} \tag{6.8}$$

Both of the above equations make the simplistic assumptions that the processes at the receiver execute at least as long as the transferred process, that the processes at the sender would have as well, had the process not been transferred, and that no new processes arrive. These assumptions exemplify the level of detail that is omitted to make calculation of these estimates efficient.

Weighing the cost against the benefit, the overall advantage in pursuing the transfer, from the point of view of the transferred process, is:

$$A_t = B_t - TC_t \tag{6.9}$$

We note that the process at the sending node for which transfer is the most advantageous is the one for which s and s_a are smallest.

Processes at the Sending Node

Processes left behind at the sending node are delayed while the CPU disconnects and transmits the transferred process to the receiving node. Making the assumption that the processes at the sender continue to execute at least until the transfer is complete, the increase in future wait time caused by the transfer, from the perspective of the processes at the sender, is:

$$TC_{s,wt} = s\, OV_s + OV_d \tag{6.10}$$

Since the overhead of disconnecting and transmitting the transferred process can be assumed to be independent of the residual service demands of the processes left behind at the sender, the increase in expected future wait *ratio* of these processes is $E(1/x'_s)\, TC_{s,wt}$. Approximating residual service demands based on the exponential assumption, the increase in wait ratio caused by the transfer for the $I\%$ processes is:

$$TC_{s,wr} = (M / \bar{X})\, TC_{s,wt} \tag{6.11}$$

Assuming that the processes at the sender continue to execute at least until the transfer is complete, the benefit that results from the transfer, as perceived by the processes remaining at the sending node and in units of wait ratio, is due to the transferred process being removed from the source CPU queue:

$$B_{s,wr} = 1 \tag{6.12}$$

Further assuming that the processes at the sender would have executed at least as long as the transferred process, had it remained, the decrease in expected future wait time of these processes is equal to the residual

service demand of the transferred process, x'_t. Again, making use of the exponential assumption in estimating residual service demands:

$$B_{s,wt} = \bar{X} \tag{6.13}$$

The advantage, then, from the vantage point of the processes remaining at the sender, is the difference between the benefit and the cost of performing the transfer:

$$A_s = B_s - TC_s \tag{6.14}$$

On the basis of this estimate, the process whose transfer results in the maximum advantage for the processes left behind at the sender is, again, the one having the smallest size.

Processes at the Receiving Node

Processes already residing at the receiving node are delayed by the CPU overhead required to load and reconnect the transferred process and by the increase the transferred process causes in the length of the CPU queue. The increase this delay causes in the expected future wait times of these processes is $s\ OV_r + OV_c + x'_t$. Again using the exponential assumption, we have:

$$TC_{r,wt} = s\ OV_r + OV_c + \bar{X} \tag{6.15}$$

Noting that the overhead of receiving and disconnecting the transferred process can be assumed to be independent of the residual service demands of processes at the receiver, and making the exponential assumption, the increase in expected future wait *ratio* caused by the transfer for the I % processes is:

$$TC_{r,wr} = 1 + (M / \bar{X})\,(s\ OV_r + OV_c) \tag{6.16}$$

Again, each of the above expressions assumes that the processes at the receiver continue to execute at least until the transfer is complete.

Unfortunately, there is no benefit in performing this transfer from the point of view of the processes at the receiver:

$$B_r = 0 \tag{6.17}$$

So, the advantage, or in this case disadvantage, of performing the transfer from the point of view of the processes at the receiving node, is the negation of the transfer cost:

$$A_r = -TC_r \tag{6.18}$$

Once again, the process for which transfer is the most advantageous (least disadvantageous) to the processes at the receiving node is the one having the smallest size.

The Transfer Decision

The overall advantage, or disadvantage, of performing a particular process transfer is the expected difference, resulting from the transfer, in the future wait times or wait ratios of the processes at the sending and receiving nodes:

$$A = \frac{(n_s - 1)A_s + n_r A_r + A_t}{n_s + n_r} \tag{6.19}$$

To find the maximum advantage of transferring a process from a node, it is necessary to calculate A for only the smallest process residing at the node. Since this process results in the largest values for A_m, A_s and A_d, A is maximized as well.

Ignoring for a moment the error in our estimates, as well as costs other than CPU Transfer Cost, the mean wait time or wait ratio of all processes is reduced if $A > 0$. While this criterion may also result in a reduced standard deviations of wait time or wait ratio, the standard deviations are always reduced if $A_t > 0$ and $A_s > 0$,

as well. Therefore, to meet a performance objective that considers only the mean, only A need be examined, though to meet an objective containing the standard deviation, A_T and A_S should be examined as well.

A potential sending node may wish to calculate the maximum load at a receiving node that allows these criteria to be met. In units of wait time, the maximum receiver load for which $A > 0$ is:

$$MRL_{1,wt} = \left\lfloor \frac{(n_s - 1)(A_{s,wt} + 1) - (t_b / \bar{X})}{1 - A_{r,wt}} \right\rfloor$$

or, in units of wait ratio:

$$MRL_{1,wr} = \left\lfloor \frac{(n_s - 1)(A_{s,wr} + 1) - (Mt_b / \bar{X})}{1 - A_{r,wr}} \right\rfloor$$

Similarly, the maximum receiver load for which $A_T > 0$, in units of wait time, is:

$$MRL_{2,wt} = \left\lfloor n_s - 1 - (t_b / \bar{X}) \right\rfloor$$

or, in units of wait ratio:

$$MRL_{2,wr} = \left\lfloor n_s - 1 - (M\ t_b / \bar{X}) \right\rfloor$$

Since A_s is independent of the load at the receiver, we set $MRL_3 = \infty$ if $A_s > 0$ and $MRL_3 = -1$ if $A_s \leq 0$. If only mean wait time or wait ratio is considered important, the maximum receiver load is MRL_1, though if the standard deviation is also a part of the performance objective, the maximum receiver load is:

$$MaxRecvLoad = \min(MRL_1, MRL_2, MRL_3) \tag{6.20}$$

Conversely, a potential receiving node may wish to calculate the minimum load at a sending node that allows these criteria to be met. In units of wait time, the minimum sender load for which $A > 0$ is:

$$MSL_{1,wt} = \left\lceil \frac{(A_{s,wt} - n_r A_{r,wt}) + \bar{X}(n_r + 1) + t_b}{A_{s,wt} + \bar{X}} \right\rceil$$

or, in units of wait ratio:

$$MSL_{1,wr} = \left\lceil \frac{(A_{s,wr} - n_r A_{r,wr}) + n_r + 1 + (M\ t_b / \bar{X})}{A_{s,wr} + 1} \right\rceil$$

Similarly, the minimum sender load for which $A_T > 0$, in units of wait time, is:

$$MSL_{1,wt} = \left\lceil n_r + 1 + (t_b / \bar{X}) \right\rceil$$

or, in units of wait ratio:

$$MSL_{2,wr} = \left\lceil n_r + 1 + (M\ t_b / \bar{X}) \right\rceil$$

Since A_s is independent of the load at the sender, we set $MSL_3 = 0$ if $A > 0$ and $MSL_3 = \infty$ if $A_s \leq 0$. Again, if only mean wait time or wait ratio is considered important, the minimum sender load is MSL_1, though if the standard deviation is also a part of the performance objective, the maximum receiver load is:

$$MinSendLoad = \max(MSL_1, MSL_2, MSL_3) \tag{6.21}$$

6.2.2.2. CPU Negotiation Cost, Benefit and Advantage

In estimating CPU negotiation cost, benefit and advantage, we use the following notation in addition to that used in the previous subsection:

k	Number of negotiation messages in a given negotiation session
$n_{r,i}$	CPU queue length at the node receiving negotiation message i

When a negotiation message is sent and responded to, each process at the sender is delayed by the overhead of sending the message and receiving its response, and each process at the receiver is delayed while the message is received and a response is sent. The cost, then, of a single negotiation message, in units of wait time, is $(n_s + n_r)OV_p$. Our estimate of negotiation cost is the average delay experienced as a result of a negotiation session comprised of k messages and responses. In units of wait time, this cost is:

$$NC_{wt} = \frac{OV_p \left[k\, n_s + \sum_{i=1}^{k} n_{r,i} \right]}{n_s + \sum_{i=1}^{k} n_{r,i}} \tag{6.22}$$

Making the exponential assumption, the negotiation cost in units of wait ratio for the I % processes is:

$$NC_{wr} = (M / \bar{X})\, NC_{wt} \tag{6.23}$$

The benefit imparted by a negotiation session is the transfer that may result. The average advantage experienced over all processes affected by the session is:

$$A_{neg} = \frac{(n_s - 1)\, A_s + n_r\, A_r + A_t - NC}{n_s + \sum_{i=1}^{k} n_{r,i}} \tag{6.24}$$

where A_s, A_r and $A_t = 0$ if no transfer results from the session.

6.2.2.3. Communication Device Cost

In estimating communication cost, we use the following notation:

r_{non}	Response time of a message on the communication device when load distributing is not active
r_{LD}	Response time of a message on the communication device when load distributing *is* active
μ_{com}	Communication device service rate
ρ_{non}	Utilization of the communication device other than for load distributing
ρ_{LD}	Utilization of the communication device for load distributing
ρ_{tot}	Total communication device utilization $= \rho_{non} + \rho_{LD}$

Our estimate for communication cost is the increase in packet response time caused by use of the communication device for load distributing:

$$CC = \frac{E(r_{LD} - r_{non})}{E(r_{non})} = \frac{E(r_{LD}) - E(r_{non})}{E(r_{non})}$$

If we make the simplifying assumptions that the interarrival times and communication device service demands of packets are exponentially distributed, and scheduling for the communication device is work-conserving, then:

$$E(r_{non}) = \frac{1}{\mu_{com}(1-\rho_{non})} \qquad \text{and} \qquad E(r_{LD}) = \frac{1}{\mu_{com}(1-(\rho_{LD}+\rho_{non}))}$$

So:

$$CC = \frac{\rho_{LD}}{1-(\rho_{non}+\rho_{LD})} = \frac{\rho_{LD}}{1-\rho_{tot}} \tag{6.25}$$

6.2.3. Summary

A framework has been described for constructing adaptive load distributing algorithms. This framework is sufficiently general to be applicable to a variety of load distributing algorithm designs having a variety of performance objectives. The observation underlying this framework is that performance is most improved if only those load distributing actions that improve performance (*effective* actions) are allowed to occur, while those that carry so much overhead compared with their potential benefit that they degrade performance (*ineffective* actions) are disallowed. From this perspective, the prerequisites for an adaptive algorithm are the ability to discern effective actions from those that are ineffective, and the ability to discriminate on that basis in determining which actions are performed. Beginning with the latter of these points, our framework includes a formalization of the degree of load distributing performed by an algorithm as its tolerance, and a catalog of mechanisms for manipulating the tolerance of an algorithm. These mechanisms can be used to determine which actions are performed. Completing the framework is a set of estimates for the performance cost and benefit of load distributing actions that take the system environment, as well as the node state, into account. Using these estimates to discern effective from ineffective actions, the tolerance can be adjusted so that ineffective actions are not performed. The goal of the estimates, then, is to be sufficiently accurate to effectively control an adaptive tolerance, yet be efficient, so that their overhead does not significantly reduce the improvement in performance available through load distributing. In seeking a balance between accuracy and efficiency, we have developed estimates that omit many details about the system state, with some cost in accuracy. What remains is to test this balance, as well as the general applicability of the framework, by applying it to existing load distributing algorithms, extending them to include an adaptive tolerance and measuring the effects on performance.

6.3. Adding an Adaptive Tolerance to the PollGen Algorithm

In this section, as an example, we apply the framework discussed in the previous section to construct an adaptive tolerance for the PollGen algorithm. Since the CPU cost-effectiveness estimates derived in the previous section are slightly different for wait time than for wait ratio, we begin by constructing an adaptive tolerance that maintains acceptable performance in terms of the wait time indices. We then examine the effect on performance if the tolerance instead focuses on wait ratio. Our description of the adaptive tolerance for PollGen is spread across three subsections, each of which addresses a portion of the tolerance that solves a specific problem.

We begin by constructing a tolerance that adapts within the *no-load-distributing/load sharing/load balancing* continuum discussed in section 5.2. For the PollGen algorithm, the difference between load sharing and load balancing is apparent only for sender-initiated transfers. Potential receiving nodes negotiate only on becoming idle, regardless of the load distributing strategy pursued. To focus on adapting within this continuum, section 6.3.1 addresses the construction of an adaptive tolerance for a PollGen algorithm constrained to performing only sender-initiated placement.

Section 5.3 showed that augmenting placement with process migration can improve performance. Making the most effective use of migration is explored in section 6.3.2. The goal is to adapt within a continuum ranging from no migration to the fullest use of migration by performing only those migrations that are cost-effective.

In the course of sections 6.3.1 and 6.3.2, we find that an adaptive tolerance that controls only process transfers is not sufficient to provide acceptable performance in certain system environments. Even if ineffective transfers can be avoided, the overhead of negotiations that fail to locate cost-effective transfers can significantly degrade performance. Section 6.3.3 explores *adaptive negotiation*. The goal of this section is to construct an adaptive tolerance that reduces the rate of failed negotiations, while not significantly reducing the rate of successful negotiations. Such a tolerance adapts within a continuum ranging from receiver-initiated, through symmetrically-initiated, to sender-initiated negotiation.

Section 6.3.4 examines the differences in performance that result from using cost-benefit estimates that are in units of wait ratio, rather than wait time. Finally, the results of this section are summarized in section 6.3.5.

6.3.1. Adaptive Placement

Using the framework described in section 6.2, an adaptive tolerance can easily be added to a PollGen algorithm constrained to sender-initiated placement. We begin by considering CPU Transfer Advantage. The tolerance we describe adapts by manipulating the criteria for identifying nodes that initiate negotiation (T_{Sneg}) and for identifying suitable transfer partners (T_{Rmax}). Before negotiation begins, *MaxRecvLoad* is calculated from equation 6.20. If *MaxRecvLoad* < 0, negotiation need not be initiated, since no receiving node will result in a cost-effective transfer. If $0 \leq$ *MaxRecvLoad* < 1, PollGen proceeds with the load sharing phase of negotiation, but does not continue into the balancing phase, since no node that is not idle will result in a cost-effective transfer. Finally, if *MaxRecvLoad* ≥ 1, it is included in negotiation messages during the balancing phase to specify the maximum load allowed for a suitable receiving node. A node receiving such a negotiation message agrees to accept a transferred process only if its load is less than or equal to this value.

Continuing, we broaden the adaptive tolerance to consider Communication Cost. This portion of the tolerance adapts by manipulating the criterion for identifying nodes that initiate negotiation (T_{Sneg}). Every *ThreshChangePeriod* seconds, Communication Cost, CC, is calculated from equation 6.25. If, due to the communication traffic of load distributing, CC becomes greater than $MaxCC$, T_{Sneg} is increased by *ThreshChangeUnit*. Later, if CC drops below $MinCC$, indicating that the communication device is not sufficiently utilized, T_{Sneg} is reduced. While T_{Sneg} has no upper limit, it is never reduced below 2, which is the minimum threshold required for either load sharing or load balancing. In our simulations, the following parameters are assumed for this portion of the tolerance:

ThreshChangePeriod	1 second
ThreshChangeUnit	0.1
MaxCC	1.8
MinCC	1.4

To calculate *MaxRecvLoad* and CC, the utilization of the communication device, ρ_{tot}, must be estimated. In addition, calculation of CC requires estimation of the portion of the communication device utilization that is the result of load distributing, ρ_{LD}. To estimate these parameters, we assume the existence of a device, either software or hardware, capable of measuring the utilization of the communication device. While this study does not address the construction of such a device, a hardware implementation does not appear difficult [Livny85]. Relying on single measurements of communication device utilization can result in oscillatory behavior. Low utilization over a measurement interval makes process transfers appear more advantageous, causing more to occur during the next interval. This increased transfer rate increases communication device utilization, causing the subsequent measurement of utilization to be high. In light of this measurement, transfers appear less advantageous, reducing the transfer rate and the communication device utilization over the next interval. To avoid this self-perpetuating oscillatory behavior, the utilization of the communication device is estimated as the mean of the most recent 10 measurements, taken at regular intervals. An upper limit of $1 - 10^{-7}$ is placed on the estimate of ρ_{tot}, avoiding division by zero in equation 6.25.

Calculation of *MaxRecvLoad* also requires the mean packet size, $\bar{S}_p$, to be estimated. Rather than sample individual packets, we pessimistically estimate that $\bar{S}_p$ is equal to the maximum packet size. Since $\bar{S}_p$ affects

only the transfer cost perceived by the process being transferred, it holds a minor role in the calculation of *MaxRecvLoad*. The increased accuracy available from sampling packets would likely not be worth the overhead incurred by sampling.

Several additional parameters are required to calculate *MaxRecvLoad*. We assume that the CPU overhead for sending and receiving a packet, OV_s and OV_r, can be accurately estimated. For a distributed system having a token-ring communication device, these parameters are likely to be nearly constant across all packets. For placement, since the transferred process has not begun execution, the CPU overheads incurred by disconnecting it at the sender and reconnecting it at the receiver, OV_d and OV_c, are zero. The mean process service demand, $\bar{X}$, is estimated at each node by the sample mean of the most recent 30 processes to have completed at that node. The loads of the sending and receiving nodes, as defined by the PollGen algorithm (section 5.2.3), are used to estimate n_s and n_r. Finally, consistent with our assumptions that processes execute independently and perform I/O to random nodes, the amount of data that must be sent between nodes as the result of a process transfer, s, is calculated simply as the physical size of the process.

Simulation

To test the ability of the tolerance described to adapt to a varying system environment, we compare the performance of PollGen with adaptive placement, which we denote Adapt-P, with that of three versions of PollGen having static tolerances: LS, LB, and the null algorithm, NoLD.

	SendProb	*RecvProb*	T_{Rmax}	T_{Sneg}
NoLD	0	0	0	2
LS	1	0	0	2
LB	1	0	∞	2
Adapt-P	1	0	adaptive	adaptive

Performance that is better than that resulting from NoLD is considered improved, while poorer performance is considered degraded. The default parameters are:

Number of nodes (m)	20
System load (ρ)	0.8
Mean process migration size ($\bar{S}$)	100K bytes
CPU service demand for transfer packets	.004 seconds
Maximum packet size	4K bytes
CPU service demand for negotiation messages	.002 seconds
Negotiation message size	32 bytes
Process initiation rates	Homogeneous
Communication device bandwidth	10 Mbits/sec.
PlaceFactor	1
PollLimit	5
F	0.1

A CPU overhead of 300 microseconds for each calculation of *MaxRecvLoad* has been included in the simulation. Simulation results show that this is the overhead that would be expected of a processor having the power of a VAX 11/780 that is 80% utilized. The overhead of calculating ρ_{tot}, ρ_{LD} and the average process service demand over the processes that have recently completed at a node are not included. Implementations of these calculations have shown them to carry minimal overhead.

We begin by examining the effects of mean process size ($\bar{S}$) on performance. Figure 6.1 plots the normalized mean and standard deviation of wait time ($\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$) against relative migration overhead, which is manipulated by varying $\bar{S}$, with $\bar{X}$ = 1.27 seconds. As shown in chapter 5, at low levels of overhead, LB performs better than LS, because LS fails to pursue some transfers that would have been cost-effective. Under such a workload, Adapt-P mimics LB, with nearly identical performance. At higher levels of overhead,

LS achieves better performance than LB by avoiding some of the ineffective transfers pursued by LB. In this range, Adapt-P performs better than either LS or LB by carefully selecting the most advantageous transfers. When overhead is higher still, both LS and LB result in instability, and NoLD performs better than either. However, Adapt-P continues to improve performance by choosing only the few transfers that are advantageous. At very high levels of overhead, such transfers become rare, and Adapt-P mimics NoLD, achieving similar performance. In summary, across a wide range of overheads, Adapt-P performs as well or better than any of the versions of PollGen having static tolerances. When possible, performance is improved, and when it can not be, at least it is not degraded. Most important, Adapt-P does not threaten the stability of the system.

Continuing, we study the ability of Adapt-P to adapt to the level of utilization of the communication device for purposes *other* than load distributing. This utilization can be expected to vary in a distributed system, depending, for example, on whether the processes that are executing tend to be I/O or CPU-bound, or on how many nodes are participating in the system. Figure 6.2 plots $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against utilization of the communication device other than for load distributing, with $\overline{X} = 2$ seconds. To simulate this communication, each process sends a message containing its result to a random node on completion. This type of communication might be typical in a system having a distributed file server. The results are similar to figure 6.1. LB performs better than LS at low utilization, while LS performs better at higher utilization. At sufficiently high utilization, though it is not shown in this figure, both LB and LS degrade performance. At all levels of utilization, however, Adapt-P performs as well or better than any of the PollGen algorithms using a static tolerance. As contention for the communication device increases, Adapt-P reduces its transfer rate by increasing T_{Snag}, limiting itself to only the most effective transfers.

To illustrate the ability of Adapt-P to adapt to the degree of heterogeneity in process initiation rates, figure 6.3 plots $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against the portion of nodes that are *arrival* nodes, with $\overline{X} = 1$ second. As in chapters 4 and 5, the entire system workload is assumed to arrive at this subset of nodes, with an equal rate at each node, while no processes initiate at the remaining nodes. A workload in which process initiation rates are heterogeneous strikingly demonstrates the ability of load distributing to improve performance. As shown by

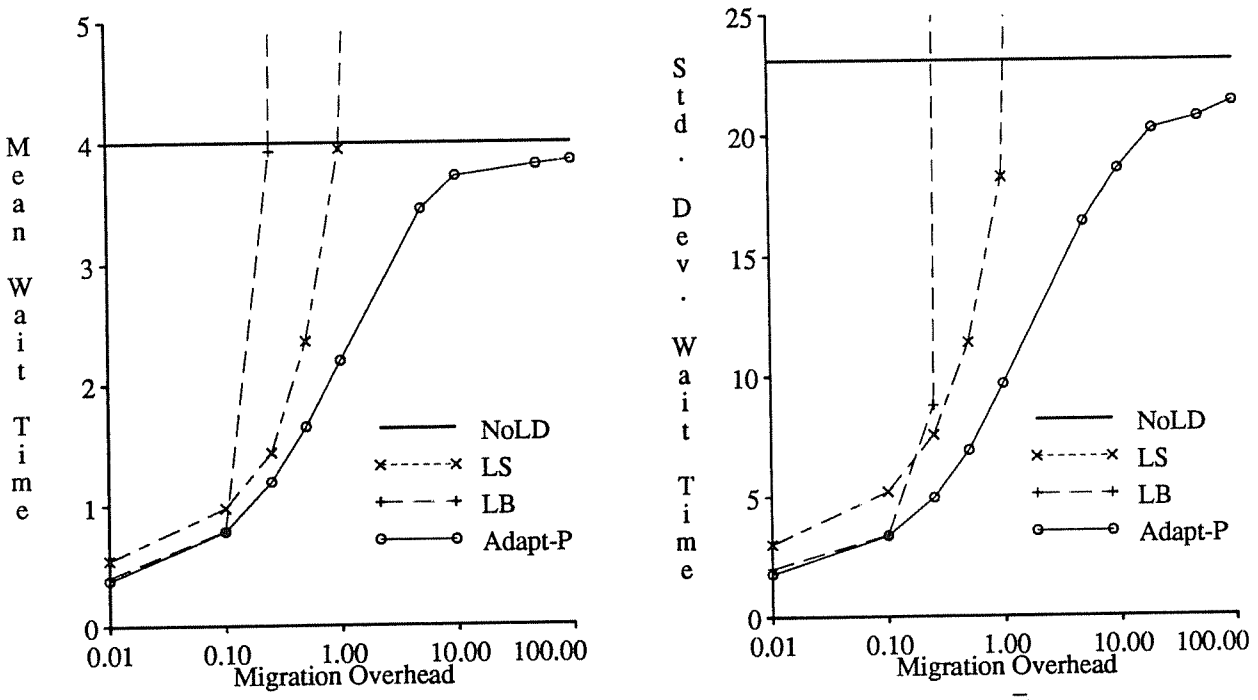

Figure 6.1 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. relative migration overhead (varying $\overline{S}$)

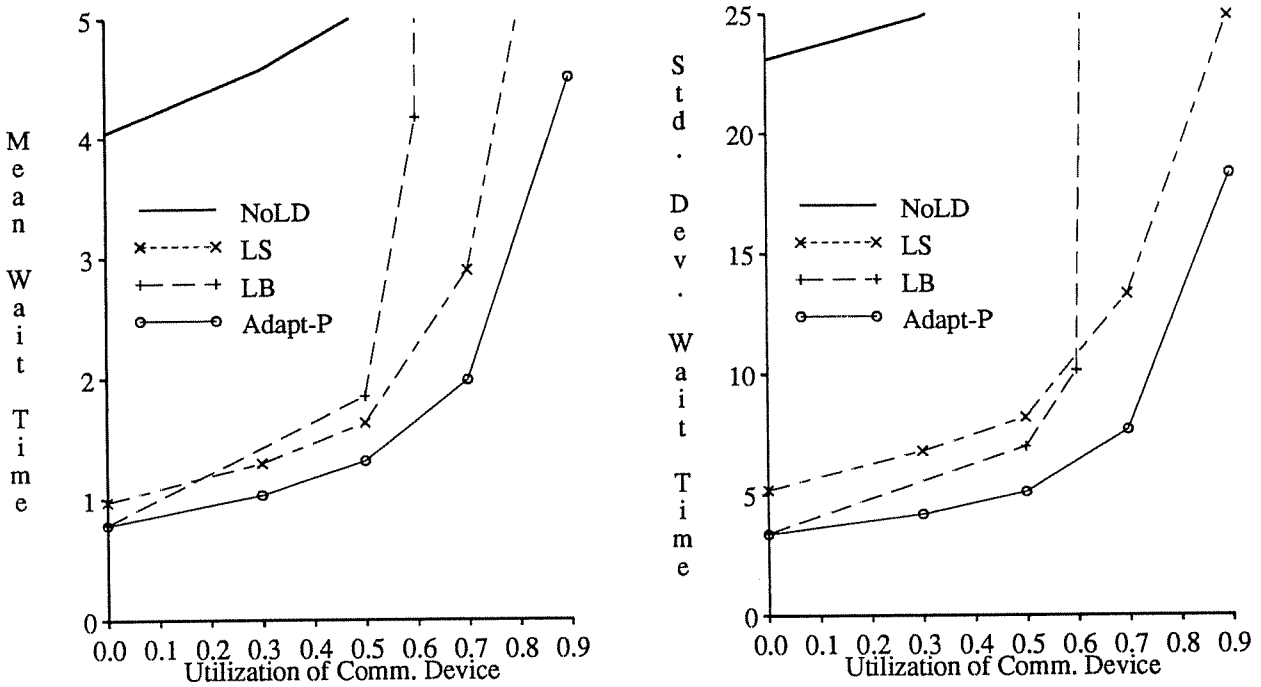

Figure 6.2 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. utilization of the communication device other than for load distributing

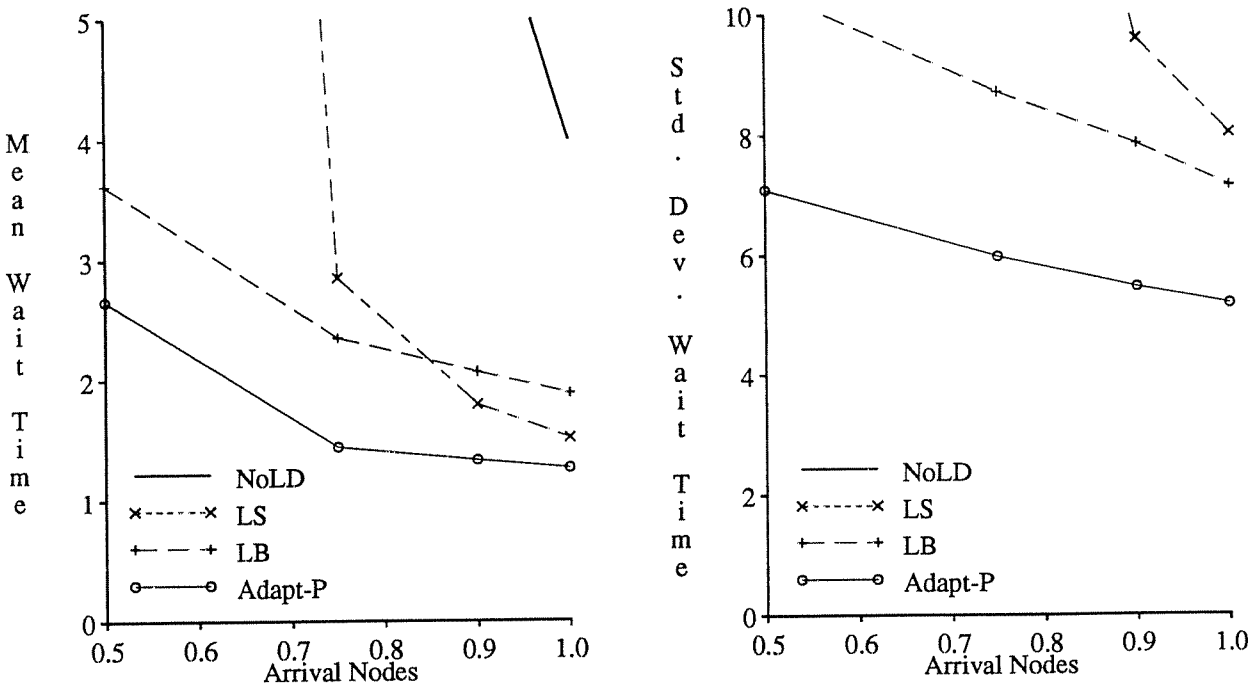

Figure 6.3 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. portion of the system nodes that are arrival nodes

figure 6.3, the performance resulting from NoLD rapidly worsens with increasing heterogeneity, until the system becomes unstable when the portion of arrival nodes equals the system load. When initiation rates are homogeneous (portion of arrival nodes near 1), LS performs better than LB, since it better avoids ineffective transfers. As heterogeneity increases (decreasing portion of arrival nodes), the performance of LB becomes better than LS, as LS fails to perform anticipatory transfers that would have been cost-effective. Throughout this range of heterogeneity, Adapt-P performs as well or better than LS, LB or NoLD.

The ability of Adapt-P to adapt to the system load is shown by figure 6.4, which plots $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against offered system load, which is manipulated by varying the process arrival rate, with $\overline{X} = 2$ seconds. At low to intermediate loads, all of the algorithms perform equally. At such system loads, since an idle receiving node can generally be found, LB rarely has an opportunity to perform a transfer that would not be performed by LS. At higher system loads, LB is able to assert its difference from LS, allowing it to achieve better performance. Under such workloads, Adapt-P mimics LB, achieving the same performance. At very high system loads, LS gains the advantage over LB, because the higher transfer rate of LB requires more CPU bandwidth than is available, resulting in an unstable system. Adapt-P, however, chooses a transfer rate between that of LS and LB, achieving better performance than either. Across the spectrum of system load, Adapt-P achieves performance that is as good or better than any of the PollGen algorithms having static tolerances.

Finally, the ability of Adapt-P to adapt to the distribution of process service demands is illustrated by figure 6.5, which plots $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against relative migration overhead, which is varied by varying $\overline{X}$. Most of the trends of figure 6.1 hold for 6.5 as well, and for the same reasons. The most notable difference is the performance of PollGen at high levels of relative overhead. At such levels, though Adapt-P is able to avoid ineffective transfers, it continues to pursue negotiations that inevitably fail. Because process service demands are so short, the overhead from failed negotiation has a significant effect on performance, and performance is degraded with respect to NoLD. Thus, placement adaptivity, by itself, is not sufficient to avoid degradation in performance under all workloads. To maintain acceptable performance over this range of $\overline{X}$, the rate of failed negotiation must be better controlled.

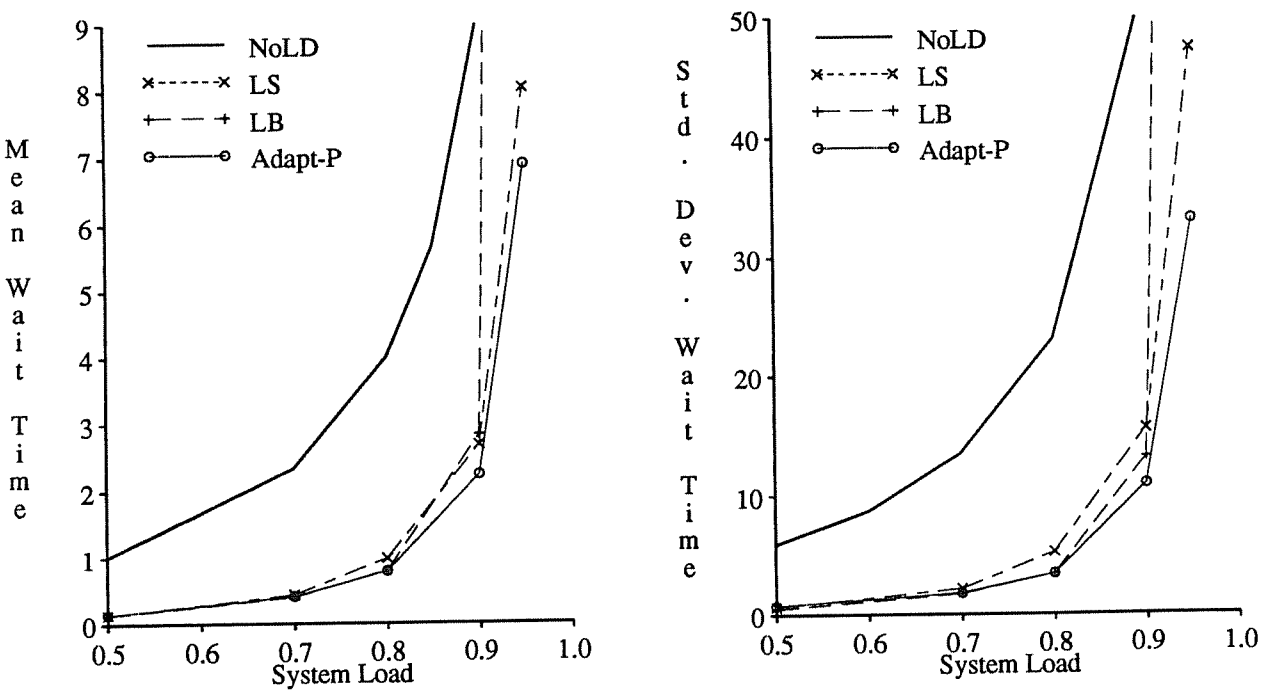

Figure 6.4 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. offered system load

In summary, the adaptive tolerance described in this subsection successfully ranges the continuum from no load distributing, through load sharing, to load balancing. Across a wide range of system environments, adaptive placement chooses the point on this continuum that achieves as good or better performance than any of the algorithms studied having static tolerances. This adaptive tolerance significantly broadens the range of environments in which performance can be improved by PollGen, and in which stability is maintained.

6.3.2. Adaptive Migration

Only a minor modification to the PollGen algorithm is required to augment adaptive placement with adaptive migration. Similar to the tolerance for adaptive placement, this portion of the tolerance adapts by manipulating the criteria for identifying nodes that initiate negotiation (T_{Sneg}) and for identifying suitable transfer partners (T_{Rmax}). To add sender-initiated migration, instead of calculating *MaxRecvLoad* based on the newly-arrived process, it is calculated for the process meeting the PollGen process selection criteria (section 5.2.3):

Among the processes at the sending node that have transferred least often,
and are either newly-arrived or have executed for at least F * *CPU transfer overhead*,
choose the process having the smallest transfer size.

Negotiation then proceeds as for adaptive placement. The effect of including *MaxRecvLoad* in the decision to initiate negotiation is to strengthen the last of the process selection criteria:

choose the process having the smallest transfer size if A , A_s and $A_t > 0$.

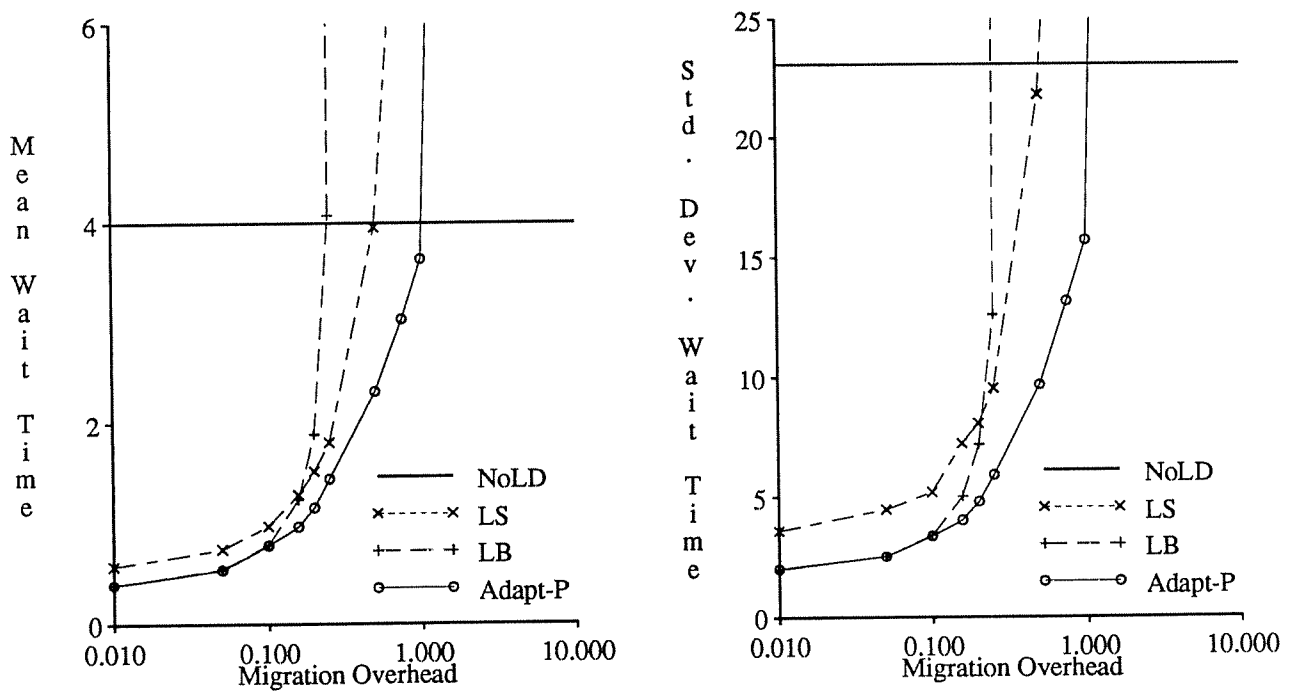

Figure 6.5 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. relative migration overhead (varying $\overline{X}$)

The addition of receiver-initiated migration is similarly easy. On receiving a negotiation message from a potential receiver, instead of using the standard PollGen criteria to select a process to transfer, the strengthened criteria are used. If no process meets these criteria, the negotiation is rejected.

Simulation

We wish to study the ability of adaptive migration to improve performance. To do so, we compare the performance of three adaptive versions of PollGen, as well as that of the null algorithm, NoLD. AdaptSend is identical to Adapt-P of the previous subsection, performing only adaptive placement. AdaptSym is the algorithm described above, augmenting placement with both sender-initiated and receiver-initiated adaptive migration. Finally, to view placement with the same level of skepticism as migration, AdaptRecv is a constrained version of AdaptSym, performing only receiver-initiated adaptive migration.

	SendProb	*RecvProb*
NoLD	0	0
AdaptRecv	0	1
AdaptSend	1	0
AdaptSym	1	1

The simulation assumptions for this section are identical to those for adaptive placement, except that $\bar{X}$ is 1.27 seconds, and PlaceFactors other than 1 are considered. In calculating *MaxRecvLoad*, both OV_d and OV_c are assumed to be zero. Thus, the difference in overhead between placement and migration is modeled as a difference in the amount of data that must be sent to a receiving node in the course of transferring a process. As in section 5.3, this difference in size is referred to as *PlaceFactor*, which is the ratio of the placement size of a process to its migration size.

We begin by studying the effect of the difference in overhead between placement and migration on performance. Figure 6.6 plots $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ against PlaceFactor, which is manipulated by varying the overhead of placement, with the overhead of migration held constant. Agreeing with intuition, the difference in performance between AdaptSend and AdaptSym shows that the ability of migration to improve performance increases with the overhead of placement, being greatest when placement carries the same overhead as migration (PlaceFactor = 1). Even when the overhead of migration is *much* greater than that of placement, adaptive migration is able to improve performance. To take a skeptical point of view, the remainder of this section assumes this worst-case for migration; placement is assumed to require only a single packet to be sent to the receiving node.

Interestingly, figure 6.6 shows that at high values of PlaceFactor, sender-initiated load distributing causes more harm than good, and AdaptRecv performs better than either AdaptSend or AdaptSym. The decreasing effectiveness of sender-initiated load distributing is caused by the increase in average overhead of sender-initiated transfers that occurs with increasing PlaceFactor. As a result, the *actual* load of the system increases, increasing the number of potential sending nodes while making suitable receiving nodes increasingly difficult to find. Many negotiations fail, adding their overhead to the already bloated actual system load. As can be seen, avoiding ineffective transfers is not sufficient to achieve the best performance in every system environment. To meet this goal, ineffective sender-initiated negotiation must also be avoided.

Taking a more obvious look at the effect of system load on performance, figure 6.7 plots $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ against offered system load. Comparing AdaptSend with AdaptSym, the improvement in performance resulting from adaptive migration increases with increasing system load. Even in this worst case for migration, the performance of AdaptSym is as good or better than that of AdaptSend across the spectrum of system load.

A second look at figure 6.7 shows that at low system loads, AdaptSend performs better than AdaptRecv. Since many nodes are idle under such a workload, a potential sending node has little trouble finding a suitable receiver, though a potential receiver is unlikely to find a suitable sender. At such loads, AdaptSym duplicates the performance of AdaptSend, indicating that the high rate of failed receiver-initiated negotiations performed by AdaptSym under such a load has little effect on performance. At higher system loads, both potential sending

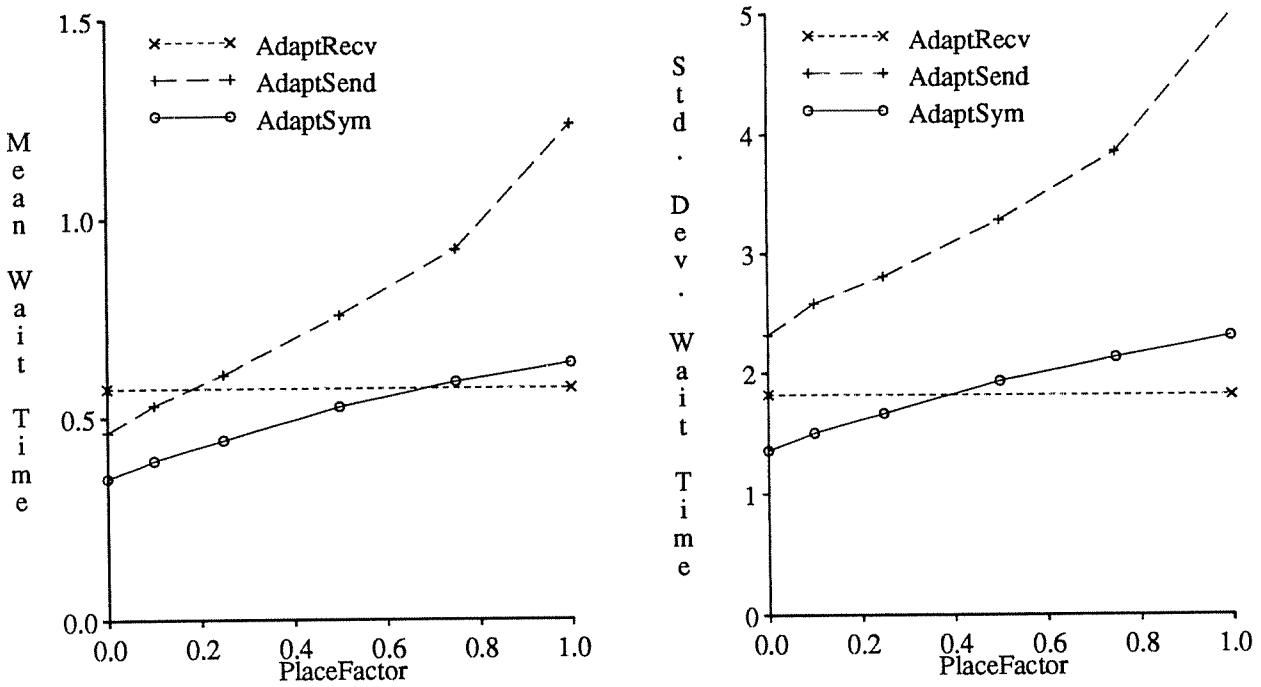

Figure 6.6 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. PlaceFactor

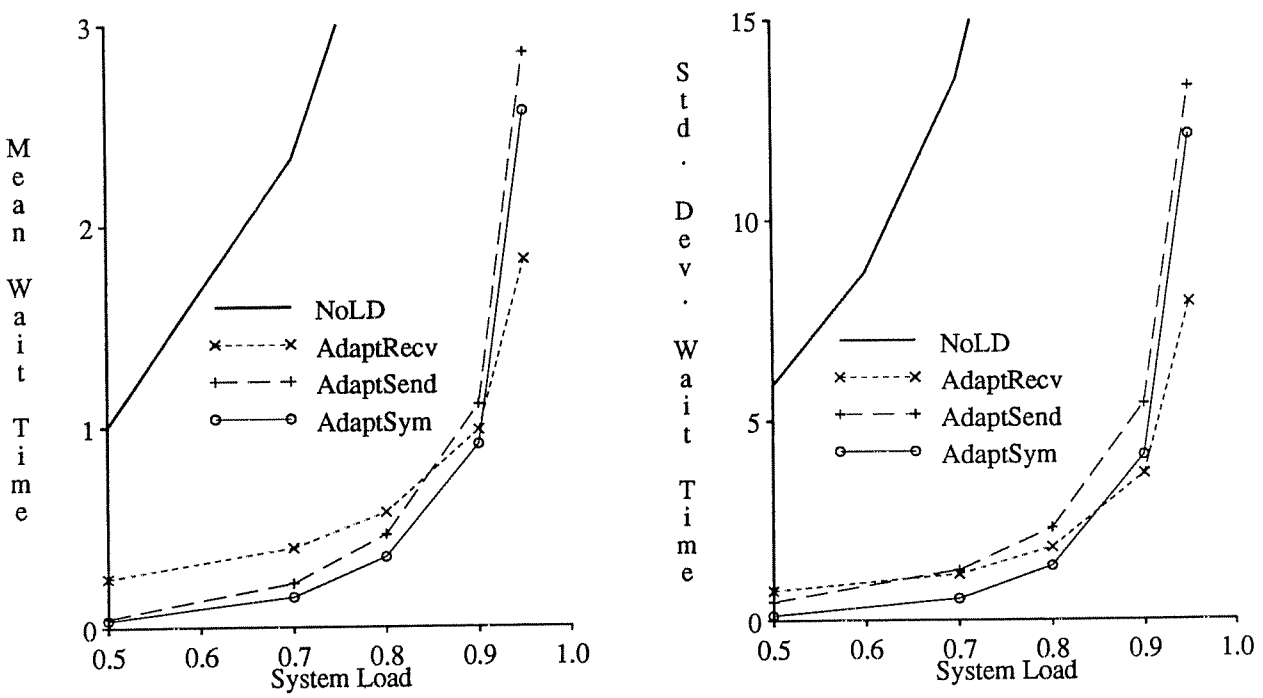

Figure 6.7 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. offered system load

nodes and potential receivers are able to find suitable transfer partners, and AdaptSym performs better than either AdaptSend or AdaptRecv. At high system loads, however, sender-initiated negotiation becomes a serious liability. The performance of both AdaptSend and AdaptSym falls off quickly, and AdaptRecv performs best. Again, avoiding ineffective transfers is not sufficient to best improve performance; ineffective *sender-initiated* negotiations must also be avoided.

The ability of adaptive migration to adapt to $\bar{X}$ is illustrated by figure 6.8, which plots $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ against relative migration overhead, which is manipulated through $\bar{X}$. Comparing AdaptSend with AdaptSym, the ability of adaptive migration to improve performance decreases with increasing relative overhead (decreasing $\bar{X}$). As $\bar{X}$ becomes smaller, fewer migrations are found to be cost-effective, thus fewer are performed. At high relative overhead, cost-effective migrations are rare and AdaptSym mimics AdaptSend, achieving similar performance.

Similar to figure 6.7, figure 6.8 shows that sender-initiated negotiation is a serious liability in some system environments. As $\bar{X}$ decreases (increasing relative overhead), the transfer rate increases, increasing the actual system load. Again, under such circumstances, a potential sender has difficulty finding a suitable receiver. As a result, the rate of failed sender-initiated negotiations is high, further increasing the already high actual system load. Under such a workload, AdaptSend and AdaptSym result in instability, and AdaptRecv achieves better performance than either by avoiding ineffective sender-initiated negotiations.

In summary, even in the worst case for migration, when migration is *much* more expensive than placement, the adaptive tolerance described in this section successfully ranges the continuum from no migration to the fullest use of migration allowed by PollGen. When cost-effective migrations are possible, they are performed and performance is improved. When no migrations are cost-effective, none are performed. The results of this section also show, however, that avoiding ineffective placements and migrations is not sufficient to achieve the best performance or to avoid instability. While ineffective receiver-initiated negotiations have little negative effect on performance, ineffective sender-initiated negotiations are capable of seriously degrading performance. To best improve performance and to avoid instability, these ineffective negotiations must be avoided.

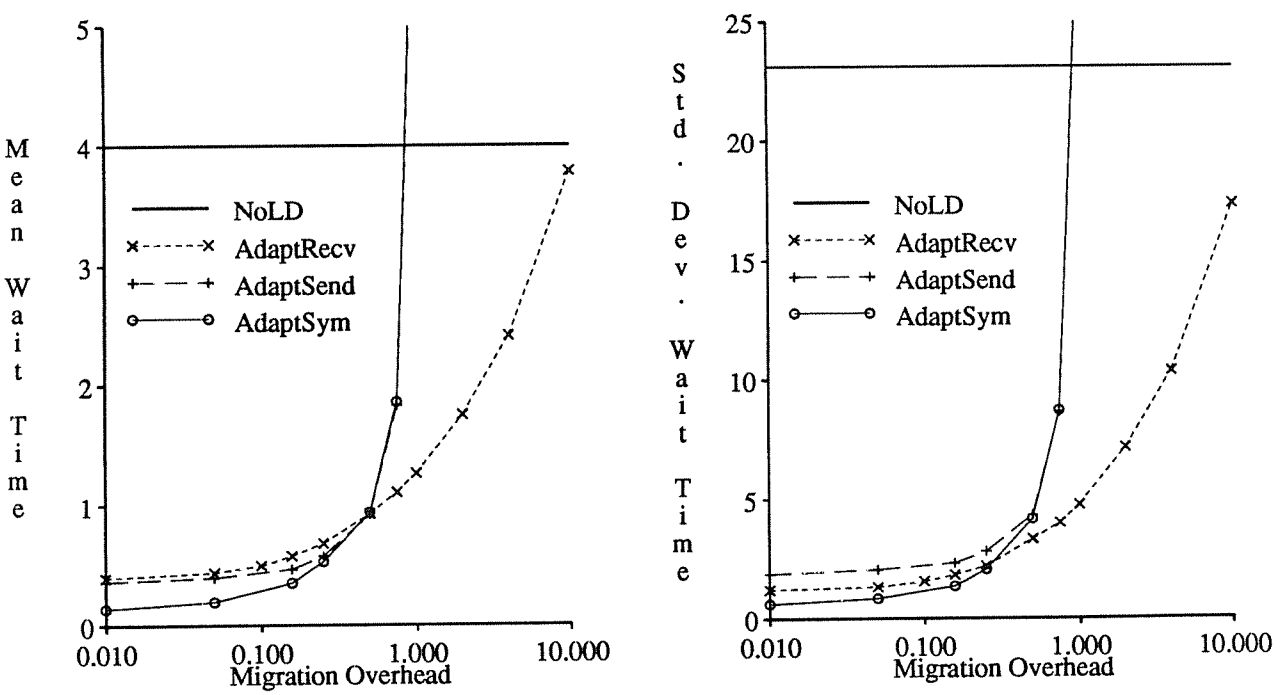

Figure 6.8 $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ vs. relative migration overhead (varying $\bar{X}$)

6.3.3. Adaptive Negotiation

The resources used in the course of a negotiation session are wasted if the session does not terminate in a cost-effective process transfer. We refer to such sessions as failed negotiation. As shown in the previous two subsections, when a significant portion of negotiations fail, overall performance can be adversely affected even though ineffective transfers are avoided. Under some conditions, the overhead of these failed negotiations cause load distributing to degrade performance and to threaten the stability of the system.

A load distributing algorithm is best able to improve performance if it does not initiate negotiations that will fail. Unfortunately, as we saw in the previous subsections, such negotiations are difficult to identify because the types of negotiation that are likely to fail depend on dynamic workload characteristics that affect the actual system load, such as $\bar{X}$ and the offered system load. The goal of adaptive negotiation is to identify the negotiations that are most likely to fail, given the current system environment, and to adjust the tolerance so that those negotiations are not initiated. Negotiations that are likely to succeed, however, should be allowed to proceed. Because high failure rates of receiver-initiated negotiation tend to have little effect on performance, while high sender-initiated failure rates tend to have more serious consequences, our focus is on controlling sender-initiated negotiation. The techniques described, however, could be used separately to control receiver-initiated negotiation.

A tolerance that provides adaptive negotiation is not as simple as that for adaptive transfer. The tolerance described here is manipulated by controlling the probability with which potential sender-initiated negotiation sessions are initiated (*SendProb*). The algorithm is distributed by maintaining this probability separately at each node. Following a negotiation-and-transfer session, the initiating node determines whether the session was successful by calculating A_{neg} from equation 6.24. This estimate of the level of success weighs the cost incurred by the negotiation and transfer against the benefit imparted by the transfer. While a negotiation session that fails to achieve a transfer is always disadvantageous, the inclusion of negotiation cost may cause some sessions that do end in transfers also to be disadvantageous. If a session is found to have been advantageous ($A_{neg} > 0$), the probability of initiating future negotiations is increased, while it is decreased if the session was disadvantageous. A decrease in *SendProb* is accomplished by multiplying it by *MigProbFactor*, and an increase is effected by dividing it by this factor. So that *SendProb* is not reduced to a level that would require a huge number of successful sessions to resurrect it to a significant level, a lower bound is set at *MinNegProb*.

This method of manipulating the tolerance has the drawback of giving equal weight to every negotiation-and-transfer session. For example, sessions that produce only slight advantages are able to counteract the effects on the tolerance of sessions that are *very* disadvantageous. As a result, performance may be degraded. At least two possible extensions to this method allow greater weight to be given to sessions having larger effects on performance. First, *SendProb* can be modified by an amount that is a function of the magnitude of the advantage. The drawback of this method is that the distribution of advantages must be well understood. If the function relating an advantage to its effect on *SendProb* is poorly chosen, sessions may typically have large effects on the probability. As a result, *SendProb* will vary wildly, possibly causing instability in the algorithm. In any case, this method allows very rare occurrences, those sessions resulting in extraordinarily large (dis)advantages, to have the longest-lasting effects. A preferable extension, which is used in this study, is to batch advantages. When a complete batch has been collected, the sum of the batched advantages determines whether *SendProb* is increased or decreased by a fixed *NegProbFactor*, and collection of a new batch begins. This extension gives greater weight to sessions that result in larger (dis)advantages within a batch, while not suffering from the drawbacks of the first proposed extension.

The method described so far for manipulating the tolerance still has a fault. In a given system environment, not all potential negotiations are equally likely to fail. The likelihood that a node will find a suitable transfer partner increases if its load suddenly increases, or if a particularly small process arrives. However, until *SendProb* is next updated, all potential negotiations have the same likelihood of being initiated. To take these differences in potential for success into account, a node estimates how *optimistic* it is that it will find a suitable partner, given its current state. To measure its level of optimism, a potential sending node calculates *MaxRecvLoad* from equation 6.20. Instead of a single negotiation probability, a node maintains a table of probabilities having *ProbTableSize* entries, with each entry corresponding to a level of optimism. Having estimated its level of optimism, a node initiates negotiation with the probability found at the appropriate entry. No entry need be kept for $MaxRecvLoad < 0$, since sections 6.3.1 and 6.3.2 disallow negotiation in that case. The final entry in the table serves for $MaxRecvLoad \geq ProbTableSize - 1$. Each entry in the table is increased

or decreased in the way that has been described in the previous paragraphs with one addition. When the probability for a given level of optimism is reduced to *MinNegProb*, all probabilities for lower levels of optimism (lower indices) are set to zero. Conversely, if a probability is increased to 1, the probability at the next lower level of optimism is increased from zero to *MinNegProb*. Using this table of probabilities, the negotiation sessions that are allowed to be initiated are those most likely to succeed, while those most likely to fail are disallowed.

Simulation

To test the ability of this portion of the tolerance to achieve adaptive negotiation, we add it to AdaptSym of the previous subsection. To denote the addition of adaptive negotiation, the resulting algorithm is referred to as AdaptSym-N. The parameters for the adaptive negotiation portion of the tolerance are:

NegProbFactor	0.5
MinNegProb	0.01
ProbTableSize	10

The simulation assumptions of this section are the same as those used in the previous section for adaptive migration. A CPU overhead of 330 microseconds for calculating A_{neg} and manipulating the table of negotiation probabilities at the end of each negotiation session is included in the simulation.

Figure 6.9, plots $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ against relative migration overhead, which is manipulated by varying $\bar{X}$. This figure is identical to figure 6.8, except that the performance of AdaptSym-N has been added. At low levels of relative overhead, sender-initiated negotiation is cost-effective, so AdaptSym-N mimics AdaptSym, achieving the same performance. At intermediate levels of overhead, as the rate of failed negotiations initiated by AdaptSym increases, AdaptSym-N reduces its use of negotiation, achieving better performance than either AdaptSym or AdaptRecv. When overhead becomes high, nearly all sender-initiated negotiation is ineffective, and AdaptSym-N adapts to mimic AdaptRecv, achieving similar performance and avoids instability. Across the range of $\bar{X}$, AdaptSym-N is successful in choosing a level of sender-initiated negotiation that provides good performance.

Plotting $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ against offered system load, which is manipulated by varying the arrival rate, figure 6.10 is identical to figure 6.7, except that the performance of AdaptSym-N has been added. At low and intermediate system loads, suitable receiver nodes are easily found, making sender-initiated negotiations cost-effective. At such loads, AdaptSym-N mimics AdaptSym, achieving the same performance. At higher loads, suitable receivers become more rare, increasing the rate of failed negotiations initiated by AdaptSym. Under such a workload, AdaptSym-N reduces its sender-initiated negotiation rate, achieving better performance than either AdaptSym or AdaptRecv. Finally, at very high loads, suitable receiving nodes become rare, and nearly all sender-initiated negotiations are ineffective. At such loads, AdaptSym-N mimics AdaptRecv, providing nearly the same performance. Again, AdaptSym-N is able to adapt its use of negotiation to the system environment, achieving good performance across the spectrum of system loads.

In summary, the adaptive tolerance described in this subsection successfully ranges from receiver-initiated to symmetrically-initiated negotiation. By doing so, the range of system environments in which PollGen is able to improve performance, and the range in which stability is maintained, is significantly broadened.

6.3.4. Focusing on Wait Ratio Instead of Wait Time

Two sets of estimates for the cost-benefit of negotiations and transfers are developed in section 6.2. The difference between these sets is their unit of measure: wait time or wait ratio. While the components of cost, in units of wait ratio, are generally $M/\bar{X}$ those in units of wait time, benefit is only $1/\bar{X}$ that in units of wait time. As a result, when $M > 1$, as it is for $I\% > 77\%$, fewer negotiations and transfers are deemed cost-effective for wait ratio than for wait time. How much the negotiation and transfer rates are consequently reduced depends on M, which is determined by $I\%$, the portion of the processes for which wait ratio cost-effectiveness is enforced. Fewer negotiations and transfers are allowed if they must be cost-effective in terms of wait ratio for

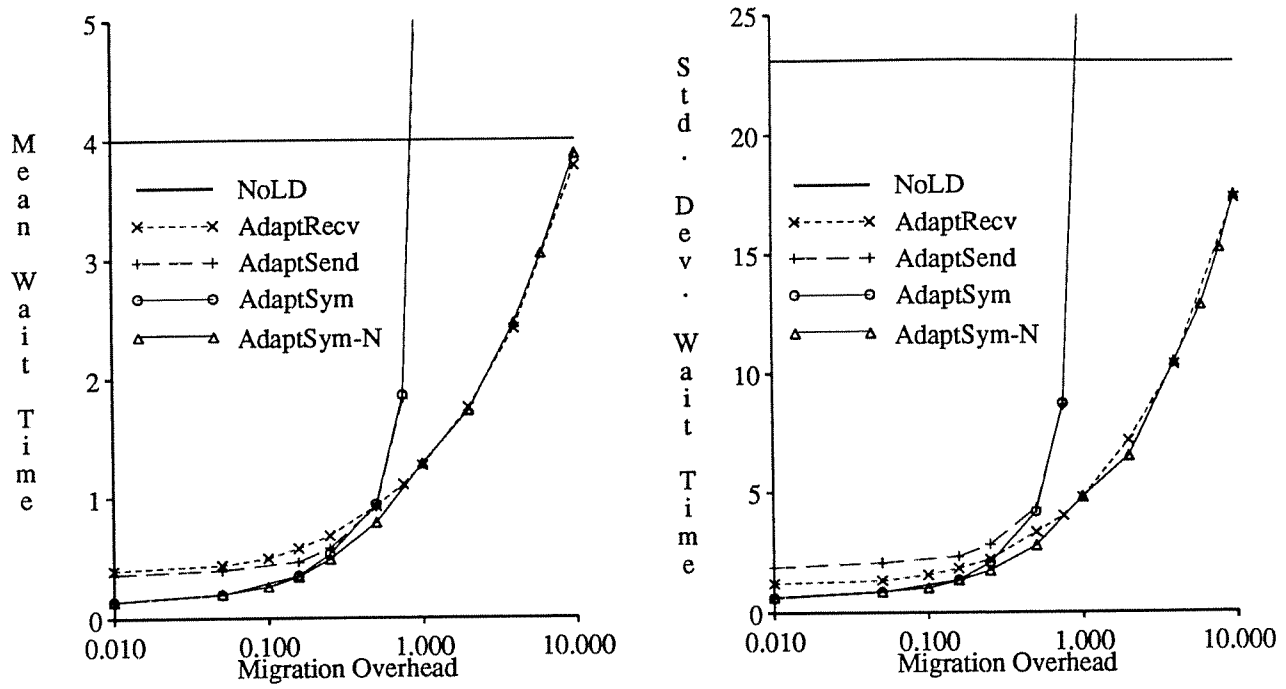

Figure 6.9 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. relative migration overhead (varying $\overline{X}$)

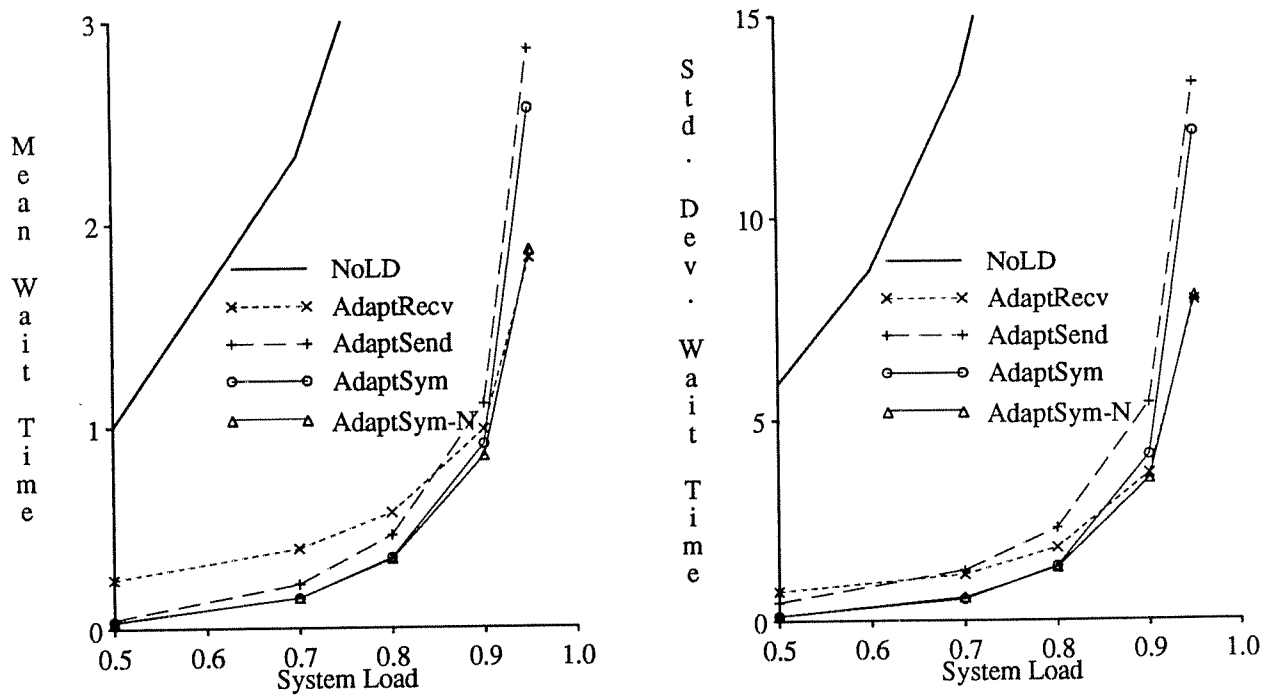

Figure 6.10 $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ vs. offered system load

99% of the processes, rather than 95% or 90% (see table B.1 in appendix B). If negotiation and transfer must be cost-effective in units of wait ratio for *all* processes, M is infinite and no negotiations or transfers are allowed.

Replacing the wait time estimates with those for wait ratio, and thus reducing the negotiation and transfer rates, can be expected to hurt performance in terms of the wait time indices. Some actions that would have been cost-effective for wait time are not performed, since they are not cost-effective for wait ratio. In this section, we compare the performance of two versions of AdaptSym-N. The first is identical to that of the previous section, using the wait time estimates, while the second instead uses the wait ratio estimates with $I\% = 99\%$ ($M = 4$). We refer to these algorithms as Adapt-WT and Adapt-WR, respectively. As in the previous sections, the improvement in performance resulting from these algorithms is evaluated relative to the null-algorithm, NoLD. Simulation assumptions are the same as for the previous section. For this section only, the results presented have less than 5% error at the 90% confidence level.

Comparing Adapt-WT with Adapt-WR, Figures 6.11 and 6.12 plot $\overline{WR}$, σ_{WR}, $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against relative migration overhead, which is manipulated by varying $\overline{S}$ (mean process size). These results are representative of the difference in performance between Adapt-WT and Adapt-WR. Similar results are shown when other parameters of the system environment are varied. Figure 6.11 shows that Adapt-WR improves $\overline{WR}$ and σ_{WR} relative to Adapt-WT. At high levels of overhead, Adapt-WT results in degradation of $\overline{WR}$ and σ_{WR} relative to NoLD, though Adapt-WR maintains cost-effectiveness in these indices across the range of overheads. As expected, however, this improvement in wait ratio indices is shown by figure 6.12 to generally be at the cost of poorer performance in terms of the wait time indices. An exception occurs at high relative overhead, where Adapt-WR performs better than Adapt-WT, even for these wait time indices. This anomaly is caused by inaccuracy in our cost-benefit estimates. When processes are very large, the simplifying assumptions on which these estimates are based result in the advantage of a transfer being over-estimated. However, the estimates are sufficiently accurate to allow Adapt-WT to avoid degradation in the wait time indices relative to NoLD.

A second look at figure 6.11 shows that the harm to $\overline{WR}$ and σ_{WR} resulting from Adapt-WT relative to Adapt-WR increases with increasing relative overhead. Adapt-WT is too lax in terms of the wait ratio indices, thus it allows actions to be performed that are ineffective in units of wait ratio. At low levels of overhead, these ineffective actions are not *terribly* disadvantageous, though at high levels of overhead, they are disastrous. In contrast, figure 6.12 shows that the harm to $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ resulting from Adapt-WR relative to Adapt-WT is largest at intermediate levels of overhead. Adapt-WR is too strict in terms of the wait time indices, disallowing some actions that would have been cost-effective. At low levels of overhead, this over-strictness has little effect, since nearly all actions are cost-effective using either criteria. At intermediate levels of overhead, however, failing to perform actions that would have been cost-effective has a significant effect on the wait time indices.

In summary, in designing an adaptive tolerance for a load distributing algorithm, the performance objective of the system must be considered. If that objective includes $\overline{WR}$ or σ_{WR}, an algorithm that enforces cost-effectiveness in terms of those indices may cause performance in terms of wait time indices to suffer. If $\overline{WR}$ and σ_{WR} are ignored, however, an algorithm that enforces cost-effectiveness only in terms of $\overline{WT}$ and σ_{WT} may result in $\overline{WR}$ and σ_{WR} being significantly poorer than if the system made no use of load distributing.

6.4. Applying an Adaptive Tolerance to the Random Algorithm

The framework developed in section 6.2 for constructing an adaptive tolerance is quite general. Few assumptions are made about the design of the load distributing algorithm to which it is applied. In the previous section, we showed that this framework is easily applied to the PollGen algorithm. In this section, we consider how this framework might be applied to an algorithm that is significantly different, the Random algorithm described by Eager, Lazowska and Zahorjan [Eager86].

The Random algorithm is a member of the simplest class of dynamic load distributing algorithms. Transfer decisions are based solely on local state information; negotiation and remote information maintenance are not used. In addition, only newly-arrived processes are transferred, avoiding complicated process selection criteria. We begin by studying how an adaptive tolerance can be constructed for even this very rudimentary algorithm. Continuing, we study the performance improvements that result from this adaptive tolerance through simulation.

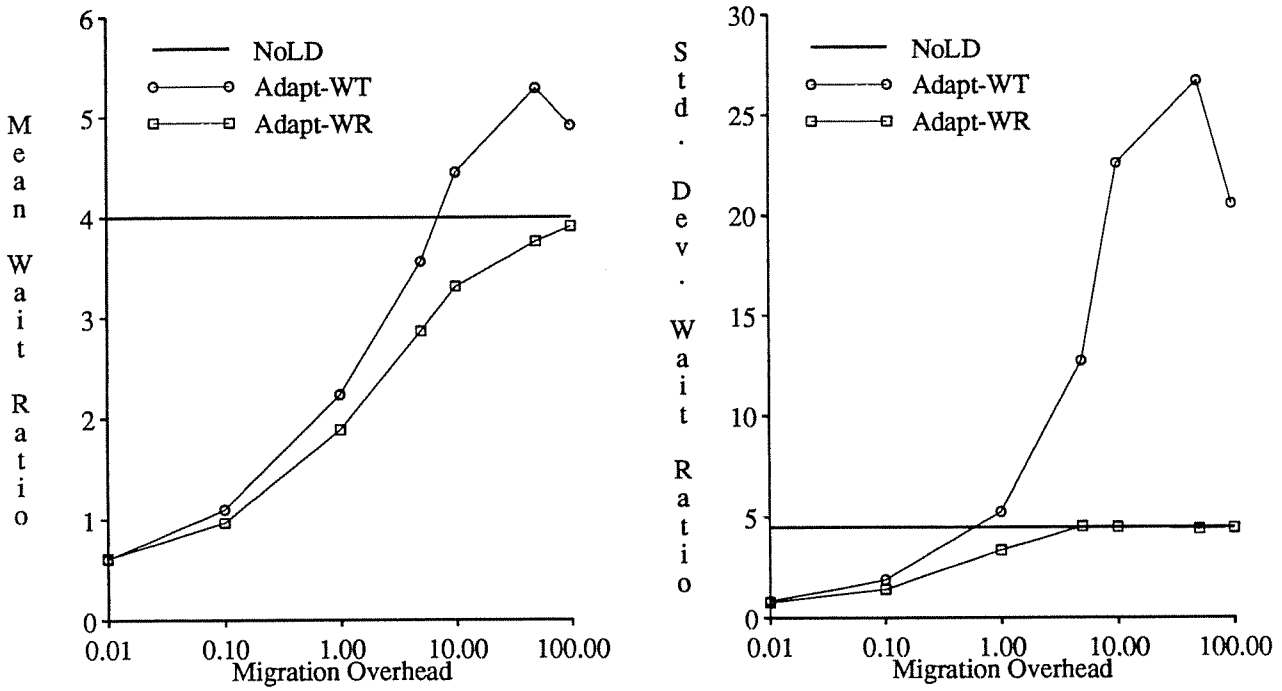

Figure 6.11 $\overline{WR}$ and σ_{WR} vs. relative migration overhead (varying $\bar{S}$)

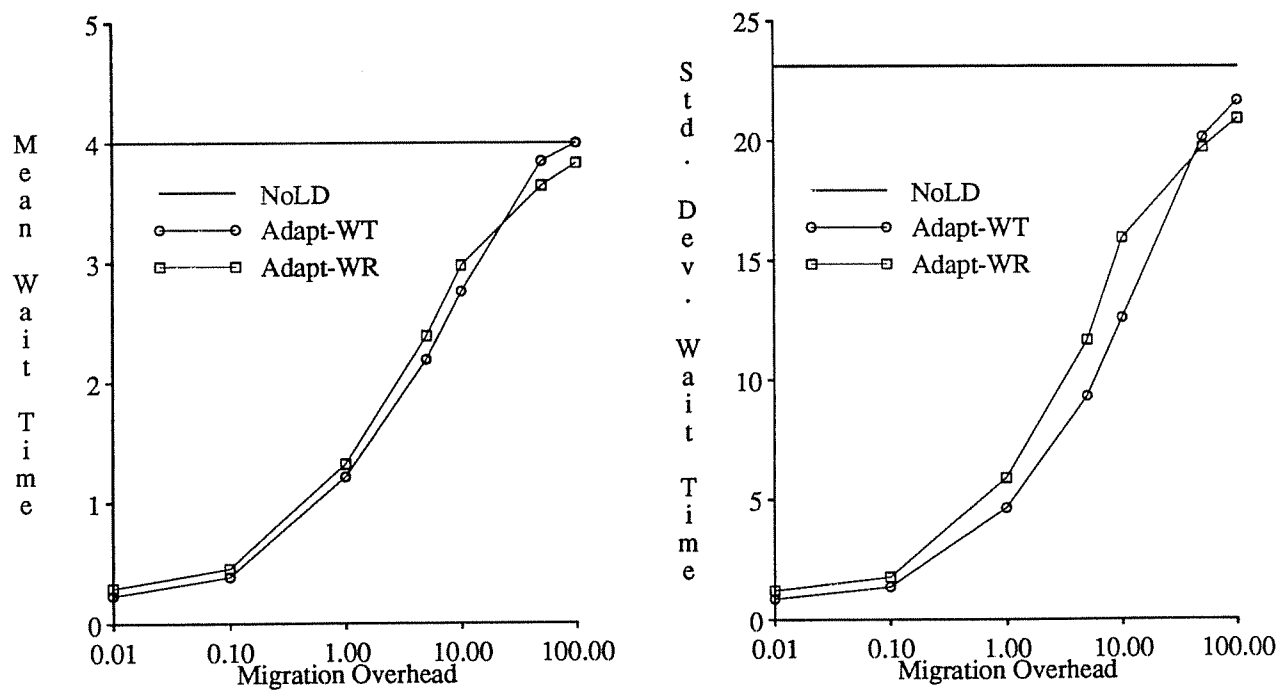

Figure 6.12 $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ vs. relative migration overhead (varying $\bar{S}$)

To briefly describe the Random algorithm, when a new process arrives at a node that already has at least T processes in its CPU queue, it is transferred to a randomly chosen node. To avoid instability, an exception is made for processes that arrive as the result of a transfer. If the process has already been transferred L_T times, it is kept at the node regardless of the length of its queue.

The simple design of Random limits its ability to make use of an adaptive tolerance. However, a tolerance can be constructed that adapts by manipulating the criteria for identifying nodes that transfer processes (T). Since the load of a receiving node is unknown to the sender, the CPU Transfer Advantage estimated by equation 6.19 can not be calculated. While the effect of this limitation is that not all disadvantageous transfers can be avoided, the transfers that are *most* disadvantageous *can* be avoided in the following way: When a process that has previously been transferred less than L_T times arrives at a node already having at least T processes, *MaxRecvLoad* is calculated from equation 6.20 based on the new process. The process is then transferred only if *MaxRecvLoad* ≥ 0. A transfer that is 'approved' in this way is not necessarily advantageous, since the load of the receiving node is not necessarily 0. Transfers that can not possibly be advantageous, however, are avoided.

This adaptive tolerance for Random can be broadened to consider Communication Cost in the same way that PollGen was extended in section 6.3.1. Every *ThreshChangePeriod* seconds, CC is calculated from equation 6.25. If, due to the communication traffic of load distributing, CC becomes greater than $MaxCC$, T is increased by *ThreshChangeUnit*. Later, if CC drops below $MinCC$, T is reduced. While T has no upper limit, it is never reduced below 1. As in section 6.3.1, the following parameters are assumed for this portion of the tolerance:

ThreshChangePeriod	1 second
ThreshChangeUnit	0.1
MaxCC	1.8
MinCC	1.4

We compare the performance resulting from the Random algorithm modified to include this simple adaptive tolerance, which we refer to as Random-AT, with that of the original algorithm, as well as the null-algorithm, NoLD:

	L_T	T
NoLD	0	∞
Random	1	1
Random-AT	1	adaptive

We examine the results from an adaptive tolerance that maintains cost-effectiveness in units of wait time. Modifying the calculation of *MaxRecvLoad* to use wait ratio as a unit of measure has an effect similar to that shown in section 6.3.4. Random, Random-AT and NoLD are simulated under the same assumptions used in section 6.3.1, except that the offered system load is assumed to be 0.7. As in the previous sections, a CPU overhead of 300 microseconds for calculating *MaxRecvLoad* is included in the simulation.

The ability of Random-AT to adapt to the utilization of the communication device is illustrated by figure 6.13, which plots $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against utilization *other than* for load distributing, with the mean process placement size assumed to be 32K bytes. Communication is simulated in the same way as in section 6.3.1. At low levels of utilization, most transfers are cost-effective, and Random-AT mimics Random, achieving the same performance. As utilization increases, however, cost-effectiveness drops off, and Random degrades performance, resulting in instability at high utilizations. In contrast, Random-AT adapts to the level of utilization, disallowing those transfers that are most ineffective. As a result, Random-AT improves performance relative to NoLD throughout the range of communication device utilization.

Continuing, figure 6.14 plots $\overline{WT}/\overline{X}$ and $\sigma_{WT}/\overline{X}$ against relative migration overhead, which is manipulated by varying the mean process placement size. While the adaptive tolerance of Random-AT is insufficient to avoid degradation in performance over the entire range of relative overheads, it is sufficient to avoid

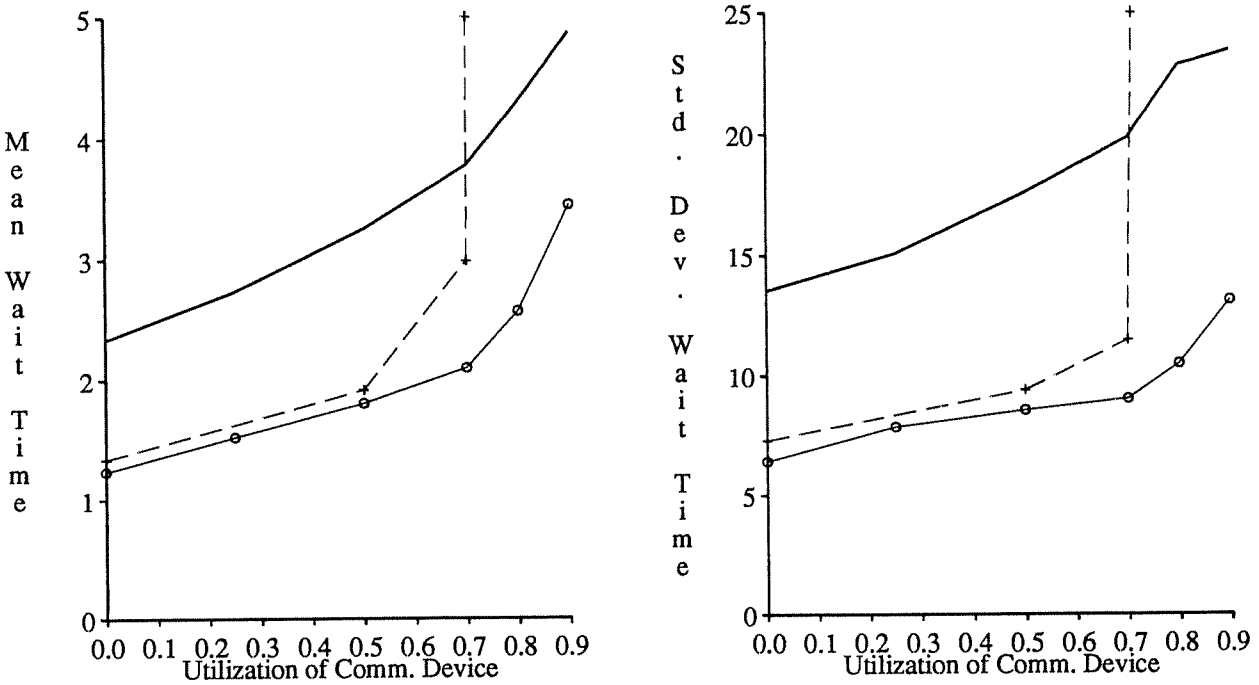

Figure 6.13 $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ vs. utilization of the communication device other than for load distributing

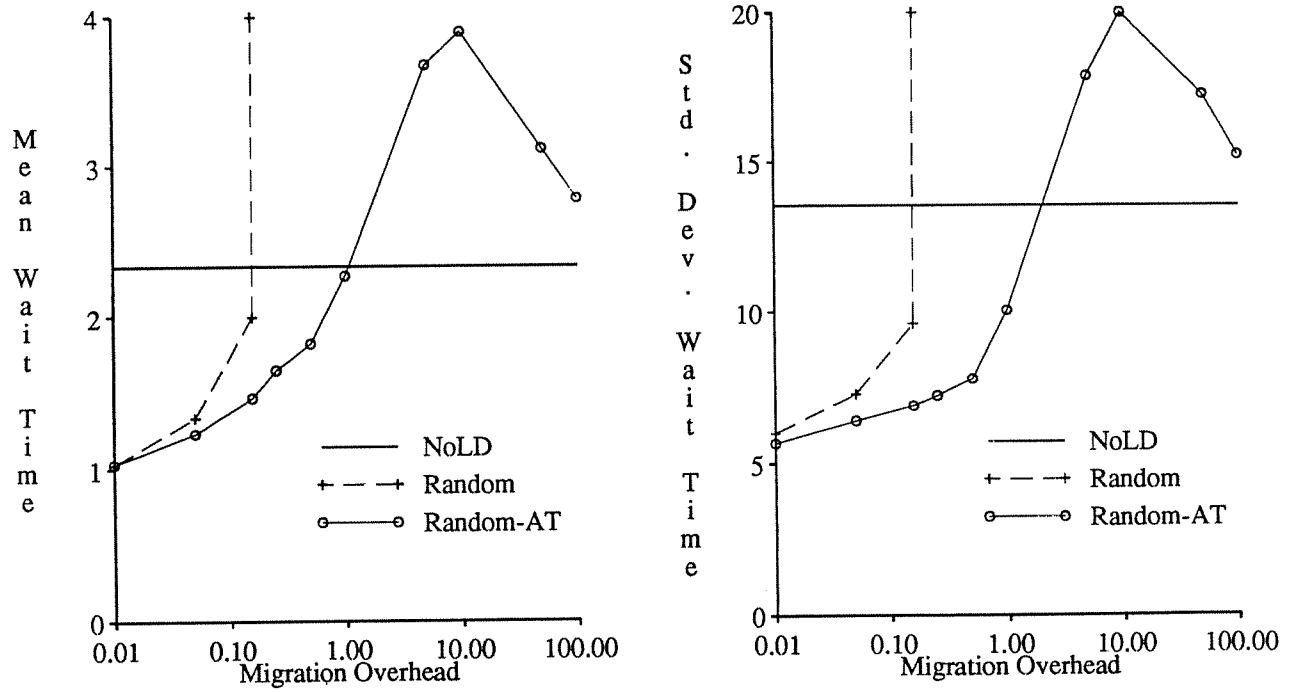

Figure 6.14 $\overline{WT}/\bar{X}$ and $\sigma_{WT}/\bar{X}$ vs. relative migration overhead (varying mean process size)

instability. Though Random results in instability at rather low levels of relative overhead, Random-AT does not *seriously* degrade performance at any point in the range of relative overhead. In addition, Random-AT significantly broadens the range in which performance is improved. Random-AT has difficulty identifying ineffective transfers when relative overhead is greater than 1. Accuracy improves, however, when overhead is very high, where nearly every potential transfer can be identified as ineffective.

In summary, even for a very simple load distributing algorithm such as Random, an adaptive tolerance can result in significant performance gains. Such a tolerance considerably broadens the range of system environments in which the Random algorithm is able to improve performance, as well as the range in which stability is maintained.

6.5. Summary and Conclusions

The previous chapter showed that the degree of load distributing that provides the best performance, and the range in degree that maintains stability for the system, are dependent on many characteristics of the system and its workload that are subject to change over time. In this chapter, we described a framework for constructing load distributing algorithms that adapt in degree, allowing them to be robust in the face of the wide range of conditions that may occur in a distributed system over time. The observation underlying this framework is that performance is most improved if only those load distributing actions that improve performance are allowed to occur, while those that carry so much overhead compared with their potential benefit that they degrade performance are disallowed. From this perspective, the prerequisites for an adaptive algorithm are the ability to discern effective actions from those that are ineffective, and the ability to discriminate on that basis in determining which actions are performed. Beginning with this latter point, we define the tolerance of a load distributing algorithm as a refinement of the notion of the degree of load distributing performed by an algorithm. The first element of our framework, then, is a catalog of mechanisms for manipulating the tolerance of an algorithm. These mechanisms can be used to determine which actions are performed. Completing the framework is a set of estimates for the cost-effectiveness of load distributing actions that take the system environment, as well as the node state, into account. These estimates can be used to discern effective from ineffective actions, and to adjust the tolerance so that potentially ineffective actions are not performed. To allow flexibility in the choice of a performance objective, separate estimates were derived for objectives based on process wait times and for those centered on wait ratio, and for those considering the mean exclusively as well as for those that also take the standard deviation into account. The resulting tolerance adapts to the system environment, allowing the load distributing algorithm to improve performance when possible and, otherwise, to avoid performance degradation.

To provide an example of how this framework can be applied, it was used to construct an adaptive tolerance for the PollGen algorithm described in chapter 5. The improvement in performance provided by this adaptive tolerance was measured through simulation. We found that manipulating the tolerance to avoid process transfers that are not cost-effective offers significant improvement in performance, as well as broadening the range of system environments in which stability is maintained. However, this level of adaptivity is not sufficient to avoid degradation in performance in some system environments. To avoid this degradation, we found it necessary to avoid performing ineffective sender-initiated negotiations, as well. Interestingly, we found that ineffective receiver-initiated negotiations had no similar negative effect on performance. The adaptive tolerance was broadened to avoid such negotiations, considerably extending the range of system environments in which PollGen is able to improve performance, as well as the range in which the stability of the system is maintained.

Modifying the PollGen adaptive tolerance to maintain cost-effectiveness in units of wait ratio, instead of wait time, enhances the ability of PollGen to improve $\overline{WR}$ and σ_{WR}. However, this improvement is generally at the cost of poorer $\overline{WT}$ and σ_{WT}. Thus, in designing an adaptive tolerance, the performance objective of the distributed system must be taken into account. An adaptive tolerance that enforces cost-effectiveness in units of wait time does so at the cost of the wait ratio indices, and vice-versa.

As a second example of how our framework can be applied, it was used to construct an adaptive tolerance for a load distributing algorithm bearing little resemblance to PollGen: the Random algorithm. Though the simplicity of this algorithm limits its ability to make use of an adaptive tolerance, simulation results show significant performance benefits result from even a rudimentary adaptive tolerance. Again, the range of system

environments in which Random is able to improve performance is considerably extended, as is the range of environments in which stability is maintained.

From these examples, we found that the framework is simple enough to be easily applied, yet general enough to be applicable to diverse load distributing algorithms having a variety of performance objectives. In addition, the balance between accuracy and efficiency in the cost-effectiveness estimates was found to provide effective control for an adaptive tolerance. In summary, this framework shows significant promise for increasing the ability of load distributing algorithms to improve performance, while reducing their threat to system stability.

CHAPTER 7

Summary and Future Research

7.1. Summary

Over periods of minutes, hours or days, a distributed operating system may experience wide variation in its environment. Environmental parameters that are subject to change include the system load, the utilization of the communication device, the distributions of process sizes and service demands, the number of nodes participating in the system and the subset of nodes that are actively generating work for the system. We have shown that each of these parameters has a role in determining which distributed scheduling policy is best able to improve performance. As a result, a policy that provides good performance at some moment may degrade performance at a later time, possibly even resulting in instability. This dissertation has focused on a problem that has not previously been addressed: the design of distributed scheduling algorithms that adapt to changes in the system environment. The goal of such algorithms is to be robust across the wide range of conditions that may occur in a distributed system over periods of minutes, hours or days.

We began this research by laying a foundation for the remainder of the study. Considering a variety of distributed scheduling policies, we studied the potential ability of each to improve performance. We defined this potential to be the performance achieved in the limiting case, when scheduling overhead is negligible. A broad view was taken of performance, with several performance indices considered, including the means and standard deviations of wait time and wait ratio. Considerable diversity was found in the performance attainable by different policies. Policies combining preemptive local scheduling, such as Processor Sharing, with load balancing were shown to improve the broadest range of performance indices across the broadest range of workloads. The use of either non-preemptive local scheduling or load sharing was found to be appropriate only when wait ratio indices are considered unimportant, process service demands are nearly exponentially distributed and process initiation rates are nearly homogeneous. Because of the likelihood that at least one of these latter restrictions on the workload will at times be violated, and because of its strong ability to improve the wait ratio performance indices, we conclude that a distributed scheduling policy composed of load balancing together with a preemptive local scheduling discipline, such as Processor Sharing, has broad applicability to distributed systems.

Unfortunately, scheduling overhead in a practical distributed system is not negligible, so the performance potential of a distributed scheduling policy can not be fully attained. Recognizing that the largest portion of distributed scheduling overhead arises from load distributing, the next portion of this study examined the effect of non-negligible overhead on the performance resulting from load distributing. Performance was examined under a variety of system and workload conditions. We found that, under many conditions, performance can be substantially improved by reducing the degree of load distributing, making compromises in how strictly the chosen distributed scheduling policy is followed. Three continua, along which the degree of load distributing pursued by an algorithm can be reduced, were studied. For one of these continua, ranging from full use of migration to no migration, no performance advantage was shown for reducing the degree to the no-migration extreme. Reducing the degree along the remaining two continua, however, was shown to result in considerable performance improvement in many system environments. These continua range from load balancing, through load sharing to no load distributing, and from symmetrically-initiated to receiver-initiated load distributing. The point on the plane formed by these continua that provides the best performance, as well as the area that maintains system stability, are dependent on a variety of characteristics of the system and its workload. Because these characteristics are subject to change over time, we conclude that an algorithm implementing a static point on this plane is insufficient to provide acceptable performance over the wide range of conditions that may occur in a distributed system. To be robust over this range of system environments, a load distributing algorithm must adapt to the system environment.

To achieve this goal, our study next focused on the design of adaptive load distributing algorithms. Recognizing the diversity of performance objectives and resources available to such algorithms, no attempt was made to design a single 'general purpose' algorithm. Instead, the goal of this section was to provide a framework, applicable across a broad range of algorithms, for adding adaptivity to an algorithm. This framework is composed of two parts. First, clarifying the notion of the degree of load distributing implemented by an algorithm by defining the tolerance of an algorithm, several methods were identified through which the tolerance can be manipulated. Completing the framework, a set of efficient estimates of the cost-effectiveness of load distributing actions were derived. These estimates can be used to discern effective from ineffective actions, allowing the tolerance to be adjusted so that potentially ineffective actions are not performed. To allow flexibility in the choice of a performance objective, separate estimates were derived for objectives based on process wait times and for those centered on wait ratio, and for those considering the mean exclusively as well as for those that also take the standard deviation into account.

To provide examples of how this framework might be applied, it was used to construct adaptive tolerances for two load distributing algorithms having significantly different designs. For one of these algorithms, two alternative adaptive tolerances were constructed, with one assuming a performance objective based on process wait time and the other assuming an objective based on wait ratio. In all cases, application of the framework was relatively easy in light of the substantial improvement in performance gained. The adaptive tolerances were shown to considerably broaden the range of system and workload conditions under which the algorithms improve performance, as well as the range of conditions in which the stability of the system is maintained. From these examples, we found that the framework is simple enough to be easily applied, yet general enough to be applicable to diverse load distributing algorithms having a variety of performance objectives. In summary, this framework shows significant promise for increasing the ability of load distributing algorithms to improve performance, while reducing their threat to system stability.

7.2. Future Research

Future work relating to this research will follow three general paths: measurement, extension and implementation.

The purpose of an adaptive tolerance is to allow a load distributing algorithm to adapt to variability in the system environment. Little is known, however, about the extent of variability in a distributed system. Assumptions about variability lead to design trade-offs in an adaptive tolerance between responsiveness and accuracy, and between complexity and range of adaptability. For example, for the two adaptive tolerances described, the mean process service demand is estimated at a node by the sample mean of the most recent processes to have completed at the node. A larger sample would allow a more accurate estimate, while a smaller sample would result in an estimate more responsive to change in this system parameter. As a second example, if utilization of the communication device is *never* high, the complexity of measuring and adapting to this utilization can be avoided. A measurement study, with the goal of measuring variability in the system environment, would aid in the design of load distributing algorithms with appropriate degrees of adaptivity.

A second goal for future research is to extend the range of workloads for which an adaptive tolerance is applicable. For example, in this study, processes are assumed not to block until they have completed execution. To remove this assumption, the measure used for the load at a node should take blocked processes into account, since they represent a future workload commitment. Transfer of blocked processes, while having no immediate effect on the load of a node, may prove advantageous in avoiding future overloaded conditions. Another extension will be to study the applicability of an adaptive tolerance to workloads containing distributed computations.

Finally, as always, the best test for an adaptive load distributing algorithm will be an implementation for a functioning distributed system.

APPENDIX A

Glossary of Notation

The following notation is used throughout this dissertation:

m	Number of nodes comprising the distributed system
n	Total number of processes present in the distributed system
$\overline{N}$	Mean number of processes present in the distributed system
p_n	Probability that the number of processes in the system is n (see [Laven83])
n_i	Number of processes residing at node i
λ	Arrival rate of processes to the system
λ_i	Rate at which processes initiate at node i
μ_i	Process service rate at node i
ρ_i	Utilization of node i $= \lambda_i / \mu_i$
ρ	System load $= \sum_{i=1}^{m} \rho_i / m$
x	Service demand of a process
$\overline{X}$	Mean process service demand
σ_X	Standard deviation of process service demands
C_X	Coefficient of variation of process service demand $= \sigma_X / \overline{X}$
C_A	Coefficient of variation of process inter-arrival times
$\overline{WT}$	Mean process wait time
$\overline{WR}$	Mean process wait ratio
$\overline{RT}$	Mean process response time
$\overline{RR}$	Mean process response ratio
σ_{WT}	Standard deviation of wait time
σ_{WR}	Standard deviation of wait ratio
σ_{RT}	Standard deviation of response time
σ_{RR}	Standard deviation of response ratio
$r(wait\ time, x)$	Correlation between wait times and service demands
$r(wait\ ratio, x)$	Correlation between wait ratios and service demands
$E(Y)$	Expected value of random variable Y
div	Integer division operator: $y\ div\ z$ is equal to the truncated quotient of y/z
mod	Modulus operator: $y\ mod\ z$ is equal to the remainder of $y\ div\ z$
r_{com}	Response time of a packet on the communication device
μ_{com}	Service rate of the communication device
ρ_{com}	Utilization of the communication device

r_{non}	Response time of a message on the communication device when load distributing is *not* active
r_{LD}	Response time of a message on the communication device when load distributing *is* active
ρ_{LD}	Utilization of the communication device for load distributing
ρ_{non}	Utilization of the communication device other than for load distributing
ρ_{tot}	Total communication device utilization $= \rho_{non} + \rho_{LD}$
$\overline{S}_p$	Mean packet size
s	Amount of data that must be transmitted to accomplish a process transfer, in packets
s_a	Amount of data remaining to be transferred *after* transferring process has blocked
OV_d	CPU service demand for disconnecting a process
OV_c	CPU service demand for reconnecting a process
OV_s	CPU service demand for sending a packet on the communication device
OV_r	CPU service demand for receiving a packet from the communication device
t_b	Time during which transferred process is blocked
n_s	CPU queue length at the sending node before the transfer is performed
n_r	CPU queue length at the receiving node before the transfer is performed
x'_t	Residual service demand of the transferred process
x'_s	Residual service demand of a process left behind at the sending node
x'_r	Residual service demand of a process residing at the receiving node before the transfer
M	(Defined by equation 6.4)
$I\%$	Percent of processes included in the wait ratio cost estimates (equation 6.5)
k	Number of negotiation messages in a given negotiation session
$n_{r,i}$	CPU queue length at the node receiving negotiation message i

APPENDIX B

Measuring Wait Ratio Statistics

While the service demand of a process can not be less than the time required to execute an instruction, analysis and simulation of computer systems are simplified if service demands are modeled to follow a continuous distribution, such as exponential or hyperexponential, which allows infinitesimal values. Measurement of wait ratio statistics in such "artificial" systems is somewhat complicated if processes are scheduled in such a way that they may not receive service immediately upon initiating. Examples of such scheduling policies include all non-preemptive scheduling disciplines, such as FCFS, LB-FCFS and LS-FCFS, as well as preemptive priority scheduling disciplines, such as preemptive HOL.[1] For preemptive scheduling disciplines, it is useful to distinguish the wait time experienced by a process before it has received initial service (wt_b) from that experienced after it has begun execution (wt_a). The expected wait ratio is then:

$$\overline{WR} = E(wait\ time / x) = E((wt_b + wt_a) / x) = E(wt_b / x) + E(wt_a / x)$$

Since we assume, as in the remainder of this appendix, that scheduling makes no use of deterministic a priori information about process service demands, wt_b for a process is independent of its service demand. So:[1]

$$= E(wt_b)\,E(1/x) + E(wt_a / x) \tag{B.1}$$

For non-preemptive scheduling disciplines, $wt_a = 0$, so this simplifies to:

$$= E(wait\ time)\,E(1/x)$$

Similarly, for preemptive disciplines:

$$\overline{WR^2} = E(wait\ time^2/x^2)$$

$$= E((wt_b + wt_a)^2/x^2)$$

$$= E(wt_b{}^2/x^2) + 2E(wt_b\,wt_a/x^2) + E(wt_a{}^2/x^2)$$

$$= E(wt_b{}^2)\,E(1/x^2) + 2E(wt_b\,wt_a / x^2) + E(wt_a{}^2 / x^2)$$

which, for non-preemptive disciplines simplifies to:

$$= E(wait\ time^2)\,E(1/x^2)$$

When service demands are exponentially distributed, with probability density function $f_x(x) = e^{-x/\bar{X}} / \bar{X}$:[2]

1. See [Klein76] and chapter 4 for descriptions of these scheduling disciplines.

1. When random variables Y and Z are independent, E[g(Y)h(Z)] = E[g(Y)] E[h(Z)].

2. Since $E(g(x)) = \int_{-\infty}^{\infty} g(x) f(x)\,dx$

$$E(1/x) = \int_0^\infty (1/x)(e^{-x/\bar{X}}/\bar{X})\,dx = (1/\bar{X})\int_0^\infty (e^{-u}/u)\,du = \infty$$

where $u = x/\bar{X}$. Substituting the above into eq. B.1, we see that whenever $E(wt_b) > 0$, $\overline{WR}$ is infinite. Similarly:

$$E(1/x^2) = \int_0^\infty (1/x^2)(e^{-x/\bar{X}}/\bar{X})\,dx = (1/\bar{X})^2\int_0^\infty (e^{-u}/u^2)\,du = \infty$$

When $E(wt_b) > 0$, $\overline{WR^2}$ is also infinite. From this result, it can be shown that:

$$\sigma_{WR} = \left[\overline{WR^2} - \overline{WR}^2\right]^{1/2} = \infty$$

Similar arguments show that the service demand distribution can be generalized to a hyperexponential or a 2 or 3-phase Erlang distribution with the same results. In addition, these results are independent of the arrival process, being valid for G/G/m and G/G/m-like systems.

It is only due to the processes having infinitesimal service demands that $E(1/x)$ and $E(1/x^2)$ are infinite, resulting in infinite $\overline{WR}$ and σ_{WR}. For $C > 0$, the conditional expectations $E(1/x \mid x > C)$ and $E(1/x^2 \mid x > C)$ are finite, resulting in finite *conditional* $\overline{WR}$ and σ_{WR}. Thus, for "artificial" systems in which service demands strictly follow an exponential, hyperexponential or 2 or 3-phase Erlang distribution and processes are scheduled such that they do not necessarily receive service immediately upon initiating, $\overline{WR}$ and σ_{WR} are stable and can be meaningfully measured only over the portion of the population remaining after those processes having service demands less than $C > 0$ are removed. The value chosen for C has an effect on the quantity of data that must be collected to achieve a given level of accuracy in measurements of conditional $\overline{WR}$ and σ_{WR}. Because the conditional moments of wait ratio increase with decreasing C, achieving the same level of accuracy requires more wait ratio samples (i.e. longer runs) for smaller values of C. Table B.1 shows, for several values of C and under an assumption of exponentially distributed process service demands, the portion of processes remaining after the shortest processes have been trimmed (where $I\% = 100\int_C^\infty f(x)\,dx$), and the conditional expectation of $1/x$. For convenience, conditional expectations listed in this table are multiplied by $\bar{X}$ (where $M = \bar{X}\,E(1/x \mid x > C)$).

In contrast, PS scheduling results in finite $\overline{WR}$ and σ_{WR} for G/G/m queues. This result follows from the definition of PS scheduling. When n processes reside in the system, each process receives service at the rate $r = 1/n$, resulting in a *partial* wait ratio, the wait ratio over that period, for each process of $r - 1$. Assuming that n is finite, as it is if $\rho < 1$ and the variance in process interarrival times is finite [Laven83], processes

C	$I\%$	M
0	100%	∞
.0001	99.99%	8.63
.0010	99.9%	6.33
.0101	99%	4.03
.0513	95%	2.44
.1054	90%	1.78
.2231	80%	1.14

Table B.1 Values for $I\%$ and M for selected values of C, assuming exponentially distributed service demands.

receive service at a finite rate, and each partial wait ratio is finite. Since the overall wait ratio of a process is the weighted sum of its partial wait ratios, the wait ratio of each process is finite, as are $\overline{WR}$ and σ_{WR}. This result generalizes to G/G/m-like systems using PS as a local scheduling discipline and having a work-conserving load distributing strategy. Since every process continuously receives service at a finite rate, all wait ratios are finite, as are $\overline{WR}$ and σ_{WR}.

APPENDIX C

The Mean and Standard Deviation of Wait Ratio for LB-PS

For an M/G/m-like system using the LB-PS distributed scheduling policy (without shuffling):

$$\overline{WR} = (1/\rho)\, E(E(wait\ ratio \mid n)) = (1/\rho) \sum_{n=0}^{\infty} E(wait\ ratio \mid n)\, p_n \tag{C.1}$$

where p_n is the probability that the system contains n processes. This probability has been found for work-conserving systems having workloads with exponentially distributed service demands [Klein76].

For load balancing, when the number of processes in the system is n, the number of nodes having greater than the mean number of processes is $k = n\ mod\ m$, and the number of processes residing on each of these nodes is $n_h = (n\ div\ m)+1$. The wait ratio of each of these processes during the period that there are n processes in the system is n_h-1. Similarly, $m-k$ nodes each have n_h-1 processes, each of which has a wait ratio of n_h-2 during this period. Thus the expected wait ratio when there are n processes in the system is:

$$E(wait\ ratio \mid n) = \frac{1}{n}\Big[k\, n_h\, (n_h-1) + (m-k)\,(n_h-1)\,(n_h-2)\Big] \tag{C.2}$$

$$= n_h\Big[(k/n)+1\Big] - 2$$

Substituting the results of eq. C.2 into eq. C.1 we see that $\overline{WR}$ for LB-PS is solely dependent on the system load and number of nodes. Similarly, an upper bound for σ_{WR} can be found:

$$Upper\ bound\ \text{for}\ \sigma_{WR} = \left[(1/\rho) \sum_{n=0}^{\infty} E(wait\ ratio^2 \mid n)\, p_n - \overline{WR}^2\right]^{1/2} \tag{C.3}$$

where:

$$E(wait\ ratio^2 \mid n) = \frac{1}{n}\Big[k\, n_h\, (n_h-1)^2 + (m-k)\,(n_h-1)\,(n_h-2)^2\Big]$$

This is an upper bound because it depends on the assumption that each job receives the same wait ratio throughout its execution (this assumption does not affect the result of eq. C.1). If some process experiences more than one wait ratio in the course of its execution, σ_{WR} is reduced. However, simulation results show that this upper bound is a close approximation.

APPENDIX D

Negotiation, Transfer and Unshared Rates in an M/M/m-like System

In this appendix, we derive the negotiation and transfer rates necessary to maintain a balanced or shared state in a distributed system under idealized assumptions. In addition, we calculate the rates at which individual nodes enter the unshared state, which is the sum of the rate at which jobs arrive at busy nodes while some nodes are idle and the rate at which nodes become idle while processes wait for service. A distributed system is in a balanced state if no pair of nodes differs in load by more than one, and in a shared state if no node lies idle while more than one process resides at some other node. Since both the LB and LS strategies conserve work in an M/M/m-like system, the results of this section are valid for any work-conserving local scheduling discipline. In the interest of generality, in deriving negotiation rates, we calculate the rates at which negotiation *sessions* are initiated, rather than the rates at which individual negotiation messages are generated.

Holding other parameters constant, the lower bound for each of the rates considered occurs when process initiation rates are homogeneous, while the upper bound is reached in the *single-source case*, when all processes initiate at a single node, which we refer to as the *initiation node*.

D.1. Load Balancing

To assure that an M/M/m-like system is always in a balanced state, negotiation must be initiated whenever either of the following events occurs:

Send A process initiates at a node that is busy

Recv A process completes execution

These events can be further subdivided into occurrences that spark successful negotiations and result in process transfers and those that do not:

SendSuccess A process initiates at a node having greater than the mean number of processes

SendFail A process initiates at a node having the arithmetic mean or fewer processes

RecvSuccess A process completes at a node having less than the mean number of processes

RecvFail A process completes at a node having the arithmetic mean or more processes

The negotiation rate is the sum of the rates of occurrence of events *Send* and *Recv*, denoted *SendRate* and *RecvRate*. The rate of sender-initiated process transfers is denoted *SendSuccessRate*, while the rate of receiver-initiated transfers is *RecvSuccessRate*.

D.1.1. Lower Bound: Homogeneous Arrival Rates

When process initiation rates are homogeneous, the rate at which processes initiate at busy nodes is:

$$SendRate = \lambda_i E(number\ of\ busy\ nodes) = \lambda_i \left[\sum_{n=1}^{m-1} n p_n + m \sum_{n=m}^{\infty} p_n \right]$$

For $\rho \leq 1$, the rate at which processes complete execution is equal to the initiation rate: $RecvRate = \lambda$. Since the number of nodes having more than the mean number of processes is $n \bmod m$, the sender-initiated transfer rate is:

$$SendSuccessRate = \lambda_i E(number\ of\ nodes > mean) = \lambda_i \sum_{n=1}^{\infty} (n \bmod m)\, p_n$$

When $n > m$, it is possible for event *RecvSuccess* to occur. The number of nodes with less than the mean number of processes is $m - (n \bmod m)$, so the receiver-initiated transfer rate is:

$$RecvSuccessRate = \mu_i E(number\ of\ nodes > 0 \text{ and } < mean)$$

$$= \mu_i \sum_{n=m+1}^{\infty} (m - (n \bmod m))\, p_n$$

Under load balancing, processes can arrive at busy nodes while nodes are idle only when $1 \le n < m$, and nodes can become idle while processes wait for service only when $m < n < 2m$, so:

$$Unshared\ Rate = \lambda_i \sum_{n=1}^{m-1} n p_n + \mu_i \sum_{n=m+1}^{2m-1} (2m-n)\, p_n$$

D.1.2. Upper Bound: Single Arrival Node

By examining a state diagram for an M/M/m queue (see [Trive82], pg. 374), the probability that the initiation node has greater than the mean number of processes, given that the system contains n processes, can be seen to be:

$$O_n = \begin{cases} 0 & \text{if } n \bmod m = 0 \\ \dfrac{\lambda p_{n-1} + \left\lceil \dfrac{n \bmod m}{(n \bmod m)+1} \right\rceil (n+1)\mu_i O_{n+1} p_{n+1}}{\lambda p_{n-1} + (n+1)\mu_i p_{n+1}} & \text{if } 0 < n < m \\ \dfrac{\lambda p_{n-1} + \left\lceil \dfrac{n \bmod m}{(n \bmod m)+1} \right\rceil m \mu_i O_{n+1} p_{n+1}}{\lambda p_{n-1} + m \mu_i p_{n+1}} & \text{if } n > m \text{ and } n \bmod m <> 0 \end{cases}$$

From this, the rates of sender-initiated negotiation and transfer are:

$$SendRate = \lambda \left[\sum_{n=1}^{m-1} O_n p_n + \sum_{n=m}^{\infty} p_n \right] \qquad SendSuccessRate = \lambda \sum_{n=1}^{\infty} O_n p_n$$

Again, since processes can arrive at busy nodes while nodes are idle only when $1 \le n < m$, and nodes can become idle while processes wait for service only when $m < n < 2m$, so:

$$Unshared\ Rate = \lambda \sum_{n=1}^{m-1} O_n p_n + \mu_i \sum_{n=m+1}^{2m-1} (2m-n) p_n$$

RecvRate, *RecvSuccessRate* and *RecvFailRate* are identical to those in a system having homogeneous initiation rates, since these events depend on process completions rather than initiations, and, under load balancing, the distribution of processes among nodes is not dependent on the homogeneity of initiation rates.

D.2. Load Sharing

An algorithm that guarantees that an M/M/m-like system is always in a shared state initiates negotiation whenever either of the following events occur:

Send A process initiates at a node that is busy

Recv A node becomes idle

These events can be subdivided into those that result in successful negotiations and those that do not:

SendSuccess	A process initiates at a busy node while some node is idle
SendFail	A process initiates at a busy node while no node is idle
RecvSuccess	A node becomes idle while at least two processes reside at another node
RecvFail	A node becomes idle when no node has more than one process

For load sharing, the unshared rate is equal to the migration rate ($E\,\hat{1}s + E\,\hat{2}s$).

D.2.1. Lower Bound: Homogeneous Arrival Rates

The rate at which processes initiate at busy nodes is:

$$SendRate = \lambda_i E(number\ of\ busy\ nodes) = \lambda_i \left[\sum_{n=1}^{m-1} np_n + m \sum_{n=m}^{\infty} p_n \right]$$

Separating this, $SendSuccessRate = \lambda_i \sum_{n=1}^{m-1} np_n$ and $SendFailRate = \lambda_i m \sum_{n=m}^{\infty} p_n$.

Our derivation of *RecvSuccessRate* when process initiation rates are homogeneous is approximate and is similar to that of Livny [Livny83], though we find a tighter lower bound. Under load sharing, event *RecvFail* occurs whenever a process completes at a node when $n \le m$, so $RecvFailRate = \mu_i \sum_{n=0}^{m} np_n$. Conversely, *RecvSuccess* can occur only when $n > m$. Under the assumption that, when $n > m$, all assignments of the $n-m$ 'excess' processes to nodes are equally likely, the probability that all $n-m$ processes are at nodes other than a given node is $((m-1)/m)^{n-m}$, and $RecvSuccessRate = \mu_i m \sum_{n=m+1}^{\infty} \left[\frac{m-1}{m} \right]^{n-m} p_n$. Simulations show that, rather than being equally likely, there is a tendency toward assignments of processes to nodes that are more unbalanced. This tendency increases the actual receiver-initiated transfer rate, so our calculation of *RecvSuccessRate* is a lower bound. However, simulation results show close agreement with this lower bound.

D.2.2. Upper Bound: Single Arrival Node

By examining a state diagram for an M/M/m queue, the probability that the initiation node is busy, given that there are n processes in the system, can be seen to be:

$$B_n = \begin{cases} 0 & \text{if } n = 0 \\ \dfrac{\lambda p_{n-1} + \left[\dfrac{n \bmod m}{(n \bmod m)+1} \right] (n+1)\mu_i B_{n+1} p_{n+1}}{\lambda p_{n-1} + (n+1)\mu_i p_{n+1}} & \text{if } n < m \\ 1 & \text{if } n \ge m \end{cases}$$

From this, $SendSuccessRate = \lambda \sum_{n=1}^{m-1} B_n p_n$. Since, under load sharing, no node is idle if $n \ge m$, $SendFailRate = \lambda \sum_{n=m}^{\infty} p_n$. When all processes initiate at the same node, load sharing allows no other node to have more than one process. The rate at which these other nodes become idle when at least two processes reside at the initiation node is $RecvSuccessRate = \mu_i (m-1) \sum_{n=m+1}^{\infty} p_n$. When $n \le m$, no node has more than one process, so $RecvFailRate = \mu_i \sum_{n=1}^{m} np_n$.

APPENDIX E

Description of the PollGen Load Distributing Algorithm

This appendix contains a pseudo-code description of the PollGen load distributing algorithm. This algorithm is a symmetrically-initiated polling algorithm having the following parameters:

Parameter	Description	Procedures
T_{Sneg}	Load threshold for sender-initiated negotiation	SendProcessActive TransferProcess SendProcessPassive
T_{Rmax}	Maximum load of a node receiving a sender-initiated transfer	SendProcessActive
T_{Tdiff}	Minimum difference in load between transfer partners	SendProcessActive
SendProb	Probability that, when a process arrival results in a load $\geq T_{Sneg}$, negotiation is initiated	SendProcessActive
RecvProb	Probability that, when a process completion leaves a node idle, negotiation is initiated	RecvProcessActive
PollLimit	Maximum number of nodes polled when searching for a transfer partner	SendProcessActive RecvProcessActive

The *load* of a node is calculated as the number of processes in the CPU queue, plus the variable *Reservations*, which is initialized to zero at each node. *Reservations* is manipulated in the procedures RecvProcessPassive and ReceivedSorryMessage. The messages that may be sent as a part of a negotiation session are:

Message	Description	Procedures
SendPoll	Poll sent by a potential sending node. Contains the field *UpperLimit*, which is the upper limit for the load of a suitable receiving node.	SendProcessActive
RecvPoll	Poll sent by a potential receiving node.	RecvProcessActive
Accept	Sent in response to a SendPoll. Indicates that a receiving node is willing to accept a process transfer.	RecvProcessPassive
SendReject	Sent in response to a RecvPoll. Indicates that a request for a process transfer is rejected.	SendProcessPassive
RecvReject	Sent in response to SendPoll. Indicates that a request for a process	RecvProcessPassive

	transfer is rejected. Contains the field *Load*, which is the load of the sender of this message.	
Sorry	Sent in response to an Accept. Indicates that the sender no longer has a process that can be transferred to the receiver.	SendProcessActive

```
(* ---------- Procedures to handle sender-initiated transfers ---------- *)

(* Initiated with probability SendProb if, when a new process arrives, load ≥ T_Sneg *)
procedure SendProcessActive
if some process meets the transfer selection criteria
  Send SendPoll messages, containing UpperLimit = 0, to a
    random set of PollLimit nodes until an Accept is received.
  Save the Load returned in each RecvReject message.
  if an Accept is received
    Call procedure TransferProcess
  else
    Sort the list of nodes that returned RecvReject messages by load
    upper_limit := min(load − T_Tdiff , T_Rmax)
    for each node having load ≤ upper_limit,
      send a SendPoll message, with UpperLimit = upper_limit
    if an Accept is received, call procedure TransferProcess
  end
end
end SendProcessActive

(* Called by SendProcessActive *)
procedure TransferProcess
if load is still ≥ T_Sneg and a process meets the transfer selection criteria,
  Transfer it to the node that sent the Accept
else
  Send a Sorry message to the node that sent the Accept
end
end TransferProcess

(* Initiated whenever a SendPoll message is received *)
procedure RecvProcessPassive
if load ≤ UpperLimit (contained in SendPoll message)
  Increment Reservations
  Send Accept message to sender of SendPoll
  Decrement Reservations when process arrives
else
  Send RecvReject message containing load
end
end RecvProcessPassive

(* Initiated whenever a Sorry message is received *)
procedure ReceivedSorryMessage
Decrement Reservations
end ReceivedSorryMessage
```

(* ---------- **Procedures to handle receiver-initiated transfers** ---------- *)

(* Initiated with probability *RecvProb* if, when a process completes, the load = 0 *)
procedure **RecvProcessActive**
Send a RecvPoll message to each of a set of *PollLimit* randomly chosen nodes until either:
  (1) a process is sent,
  (2) *PollLimit* is exhausted, or
  (3) a new process initiated, causing load to be greater than zero
end RecvProcessActive

(* Initiated whenever a RecvPoll message is received *)
procedure **SendProcessPassive**
if load $\geq T_{Sneg}$ and some process meets the transfer selection criteria
  Transfer it to the sender of the RecvPoll message
else
  Send a SendReject message
end
end SendProcessPassive

REFERENCES

[Ander74] H. A. Jr. Anderson and R. G. Sargent, "Investigation into Scheduling for an Interactive Computing System," *I.B.M. Journal of Research and Development*, pp. 125-137 (March 1974).

[Balan76] V. Balanchandran, J. W. McCredie, and V. I. Mikhail, "Models of the Job Allocation Problem in Computer Networks," *IEEE COMPCON Fall 1976*, pp. 211-214 (1976).

[Barak85] A. B. Barak and A. Shiloh, "A Distributed Load-balancing Policy for a Multicomputer," *Software-Practice and Experience 15*, 9, pp. 901-913 (September 1985).

[Blaze86] J. Blazewicz, M. Drabowski, and J. Weglarz, "Scheduling Multiprocessor Tasks to Minimize Schedule Length," *IEEE Transactions on Computers C–35*, 5, pp. 389-393 (May 1986).

[Bokha79] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Transactions on Software Engineering SE-5*, (July 1979).

[Bokha81] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Transactions on Software Engineering SE-7*, 6, (November 1981).

[Bryan81] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Second International Conference on Distributed Computing Systems*, pp. 314-323 (April 1981).

[Chang86] H. Chang, "Load Sharing and Sender Initiated vs. Receiver Initiated Strategies," *Proceedings of the Real-Time Systems Symposium*, (1986).

[Chou82] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Transactions on Software Engineering SE-8*, 4, (July 1982).

[Chou86] T. C. K. Chou and J. A. Abraham, "Distributed Control of Computer Systems," *IEEE Transactions on Computers C-35*, 6, (June 1986).

[Chu69] W. W. Chu, "Optimal File Allocation in a Multiple Computing System," *IEEE Transactions on Computers C–18*, 10, pp. 885-889 (October 1969).

[Chu80] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer 13*, 11, pp. 57-69 (November 1980).

[Eager86a] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Dynamic Load Sharing," *Performance Evaluation 6*, 1, pp. 53-68 (March 1986).

[Eager86] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering SE-12*, 5, pp. 662-675 (May 1986).

[Eager88] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing," *ACM SIGMETRICS 1988*, (May 1988).

[Edwar75] J. A. Edwards, A Two-Level Hierarchical Control System for a Distributed Network of Computers, PhD Thesis, University of Oklahoma (1975).

[Efe82] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *Computer*, pp. 50-56 (June 1982).

[Farbe73] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe, "The distributed computing system," *Proceedings of the Seventh Annual IEEE Computer Society International Conference*, pp. 31-34 (1 March 1973).

[Garey78] M. R. Garey, R. L. Graham, and D. S. Johnson, "Performance guarantees for scheduling algorithms," *Oper. Res. 26*, pp. 3-21 (January 1978).

[Gursk] M. Gursky, Some Complexity Results for a Multiprocessor Scheduling Problem, unpublished.

[Gylys76] V. B. Gylys and J. A. Edwards, "Optimal Partitioning of Workload for Distributed Systems," *Digest of Papers, IEEE COMPCON Fall 1976*, pp. 353-357 (September 1976).

[Jenny77] C. J. Jenny, "Process partitioning in distributed systems," *Digest of Papers National Telecommunications Conference '77*, pp. 31:1-1 - 31:1-10 (1977).

[Klein67] L. Kleinrock, "Time-Shared Systems: A Theoretical Treatment," *Journal of the ACM 14*, pp. 242-261 (1967).

[Klein76] L. Kleinrock, *Queuing Systems: Volume 2, Computer Applications*, John Wiley & Sons (1976).

[Krish81] S. Krishnaprasad and C. C. Price, "Distributed computer network scheduling algorithms and their complexities," Technical Report No. CSE-8105, Southern Methodist University Department of Computer Science and Engineering (August 1981).

[Krueg84] P. Krueger and R. Finkel, "An Adaptive Load Balancing Algorithm for a Multicomputer," Technical Report 539, University of Wisconsin—Madison, Dept. of Computer Sciences (April 1984).

[Krueg87] P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," *Proc. 7th International Conference on Distributed Computing Systems*, pp. 242-249 (September 1987).

[Krueg87a] P. Krueger and M. Livny, "When is the Best Load Sharing Algorithm a Load Balancing Algorithm?," Technical Report 694, University of Wisconsin—Madison, Dept. of Computer Sciences (April 1987).

[Krueg88] P. Krueger and M. Livny, "A Comparison of Preemptive and Non-Preemptive Load Distributing," *Proc. 8th International Conference on Distributed Computing Systems*, (June 1988).

[Laven83] S. S. Lavenberg, *Computer Performance Modeling Handbook*, Academic Press (1983).

[Lee77] R. P. Lee and R. R. Muntz, "On the Task Assignment Problem for Computer Networks," *Proc. 10th Hawaii International Conference on System Sciences*, pp. 5-9 (January 1977).

[Lee77a] R. P. Lee, Optimal Task and File Assignment in a Distributed Computing Network, PhD Thesis, UCLA (1977).

[Lelan86] W. E. Leland and T. J. Ott, "Load-balancing Heuristics and Process Behavior," *Proceedings of PERFORMANCE '86 and ACM SIGMETRICS 1986*, p. 54 (May, 1986).

[Litzk88] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor - A Hunter of Idle Workstations," *Proc. 8th International Conference on Distributed Computing Systems*, (June 1988).

[Livny82] M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Computer Network Performance Symposium*, pp. 47-55 (April 1982).

[Livny83] M. Livny, The Study of Load Balancing Algorithms for Decentralized Distributed Processing Systems, PhD Thesis, Weizmann Institute of Science, Rehovot, Israel (available as Technical Report 570, University of Wisconsin—Madison Computer Sciences) (August 1983).

[Livny85] M. Livny and U. Manber, "Distributed Computation via Active Messages," *IEEE Transactions on Computers*, (December 1985) to appear.

[Mutka87] M. Mutka and M. Livny, "Profiling Workstation's Available Capacity for Remote Execution," Proceedings of PERFORMANCE '87 (December 1987).

[Ni85] L. M. Ni and K. Hwang, "Optimal Load Balancing in a Multiple Processor System with Many Job Classes," *IEEE Transactions on Software Engineering SE-11*, 5, pp. 491-496 (May 1985) Relevant correspondence appears in Volume SE-12, number 3.

[Ni85a] L. M. Ni, C. W. Xu, and T. B. Gendreau, "A Distributed Drafting Algorithm for Load Balancing," *IEEE Transactions on Software Engineering SE-11*, 10, pp. 1153-1161 (October 1985).

[Nicho87] D. Nichols, "Using Idle Workstations in a Shared Computing Environment," *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pp. 5-12 (8-11 November 1987) In *ACM Operating Systems Review 21*:5.

[Ohtsu75] T. Ohtsuki, "Heuristic algorithms for large-scale combinatorial problems," *J. Inst. Electron. Commun. Eng. Japan 58*, pp. 417-423 (April 1975).

[Powel83] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," *Proc. Ninth ACM Symposium on Operating Systems Principles*, pp. 110-118 (10-13 October 1983) In *ACM Operating Systems Review* 17:5.

[Rao79] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Transactions on Computers C-28*, 4, (April 1979).

[Rosin65] R. F. Rosin, "Determining a Computing Center Environment," *CACM 8*, 7, (July 1965).

[Shen85] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Transactions on Computers C–34*, 3, pp. 197-203 (March 1985).

[Smith80] R. G. Smith, "The contract net protocol: high-level communication and control in a distributed problem solver," *IEEE Transactions on Computers C-29*, 12, (December 1980).

[Stank84] J. A. Stankovic, "Simulations of Three Adaptive, Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks 8*, pp. 199-217 (1984).

[Stank85] J. A. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *IEEE Transactions on Computers C-34*, 2, pp. 117-130 (February 1985).

[Stone77] H. S. Stone and H. S. Stone, "Multiprocessor Scheduling with the aid of Network Flow Analysis," *IEEE Transactions on Software Engineering SE-3*, 5, pp. 315-321 (May 1977).

[Stone78] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *Computer 11*, 7, pp. 97-106 (July 1978).

[Stone78a] H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Transactions on Software Engineering SE-4*, pp. 254-259 (May 1978).

[Tanta85] A. N. Tantawi and D. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *JACM 32*, 2, pp. 445-465 (April 1985).

[Theim85] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proc. Tenth ACM Symposium on Operating Systems Principles*, pp. 2-12 (December 1985).

[Tilbo84] A. M. Van Tilborg and L. D. Wittie, "Wave Scheduling - Decentralized Scheduling of Task Forces in Multicomputers," *IEEE Transactions on Computers C--33*, 9, pp. 835-844 (September 1984).

[Trive82] K. S. Trivedi, *Probability & Statistics with Reliability, Queuing and Computer Science Applications*, Prentice-Hall (1982).

[Zatti85] S. Zatti, *A Multivariable Information Scheme to Balance the Load in a Distributed System*, Department of Electrical Engineering and Computer Science, University of California, Berkeley (May 12, 1985).

[Zhou86] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing," Technical Report, UCB/CSD 87/305, Computer Science Division, University of California, Berkeley (September 1986).

[Zhou87] S. Zhou, "An Experimental Assessment of Resource Queue Lengths as Load Indices," *Proceedings of the 1987 Winter USENIX Conference*, p. 73 (January 1987).