

**Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596**

**DYNAMIC LOAD SHARING ALGORITHMS IN
ASYMMETRIC DISTRIBUTED SYSTEMS**

Alberto Avritzer

**August 1990
CSD-900023**

UNIVERSITY OF CALIFORNIA

Los Angeles

**Dynamic Load Sharing Algorithms in
Asymmetric Distributed Systems**

A dissertation

submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

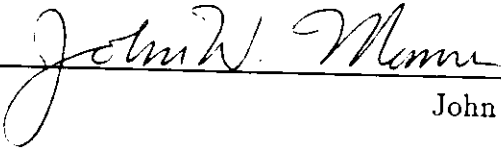
Alberto Avritzer

1990

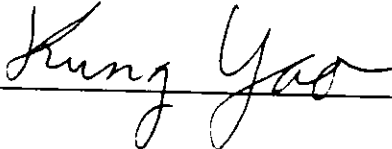
Copyright © 1990 by Alberto Avritzer

All Rights Reserved

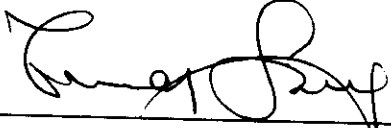
The dissertation of Alberto Avritzer is approved.



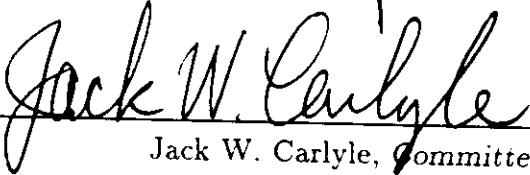
John W. Mamer



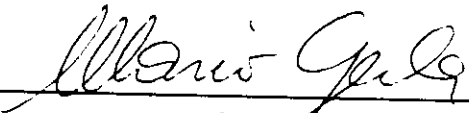
Kung Yao



Tomas Lang



Jack W. Carlyle, Committee Co-Chair



Mario Gerla, Committee Co-Chair

University of California, Los Angeles

1990

To my life companion, friend, and love, Orna

and

To my parents

Table of Contents

1	Introduction	1
1.1	Introduction	1
1.2	Problem Statement	3
1.3	Literature Review	4
1.3.1	Initial Work	4
1.3.2	Static Load Sharing	5
1.3.3	Dynamic Load Sharing in Homogeneous Systems	5
1.3.4	Dynamic Load Sharing in Asymmetric Distributed Systems	7
1.3.5	Aggregation and Decomposition	9
1.3.6	Real-Time Systems	10
1.4	Dissertation Outline	11
1.5	Contributions	11
2	Performance Modeling of Campus Computing Systems	14
2.1	Introduction	14
2.2	Environment and Model	19
2.3	A Self Tuning Dynamic Load Sharing Algorithm	20
2.4	Analytical Modeling of the Testbed	26
2.4.1	Markov Chain Modeling	26
2.5	Bounding Techniques	32
2.5.1	Bounds Example	41
2.6	Tighter Upper Bound	44
2.6.1	Bounds Example	50
2.7	Generalizations	52
2.8	Decomposition by Threshold Configuration Approach: Bursty Arrivals	53
2.9	Conclusions and Future Research	59
3	Sensitivity Analysis	61
3.1	The Model	62
3.2	Representative Algorithms	63
3.2.1	Dynamic Tuning Algorithm	64
3.2.2	Local Information Algorithm	64
3.2.3	Fixed thresholds algorithm	64
3.2.4	Optimal static algorithm	65
3.3	Simulation Results	65
3.3.1	Burstiness of the arrival process	65
3.3.2	Number of Hosts in the system	67

3.3.3	Speed Ratio between Fast and Slow processors	82
3.3.4	Sensitivity to threshold computation period (dynamic tuning only)	83
3.4	Sensitivity to the system model (dynamic tuning only)	86
3.5	Conclusions and Future Research	88
4	Hard Real Time Systems	92
4.1	Introduction	92
4.2	Literature Review	95
4.3	Evaluation of Real Time Systems Under Poisson Tasks	98
4.3.1	Model	98
4.3.2	Scheduling	99
4.3.3	Load Balancing Interconnection Network	100
4.3.4	Simulation Results-Scheduling Based on Laxity only	104
4.3.5	Simulation Results-Scheduling Based on Laxity and Threshold	112
4.3.6	Simulation Results - Bursty Arrivals	115
4.4	Evaluation of Real Time Systems Under Periodic Tasks	122
4.4.1	Model	122
4.4.2	Scheduling	123
4.4.3	Load balancing interconnection network – Periodic Tasks	125
4.4.4	Periodic and Poisson tasks mix	126
4.5	Performance Modeling	127
4.5.1	Markov Chain Modeling	127
4.5.2	Bulk Arrivals	128
4.5.3	$M^k/D/c$	130
4.6	Conclusions and Future Research	133
5	Conclusions and Future Research	137
	Bibliography	146

List of Figures

2.1	Simulation Model	19
2.2	13 Hosts, simulation and measurements, dynamic thresholds vs. fixed thresholds, average delay vs. arrival rate	25
2.3	Arrivals and departures at the slow processors when the system is saturated	28
2.4	State transitions when there are idle fast processors	28
2.5	State transitions when all fast processors are busy	29
2.6	Markov chain for the underthreshold state, arrivals	33
2.7	Markov chain for the underthreshold state, departures	34
2.8	Fast Processors Cluster Behavior	35
2.9	Markov chain for the slow processor cluster	37
2.10	Transitions in Region A	45
2.11	Transitions in Region B	46
2.12	Transitions in Region C	46
2.13	Transitions in Region D	47
2.14	Relation among regions	48
3.1	Simulation Model	62
3.2	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 10	68
3.3	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 26; number of hosts = 10	70
3.4	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 10	72
3.5	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 44; number of hosts = 10	74
3.6	Average delay vs. burstiness; burst length = 10; utilization = 25% ; number of hosts = 10	76
3.7	Average delay vs. burstiness; burst length = 0.5; utilization = 50% ; number of hosts = 10	77
3.8	Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 10.0; arrival rate = 22; number of hosts = 10	85
4.1	Real-time architecture considered	98
4.2	Load balancing interconnection network	101
4.3	Matching Network	102
4.4	Markov chain for bulk arrivals system	129
4.5	Markov chain for $M^k/D/C$ system	132

List of Tables

2.1	Bounds on Average Delay for the 60 Host Example	42
2.2	Simulation Results for 60 Hosts Example; Average Delay vs. Arrival Rate	43
2.3	Bounds on Average Delay for the 60 Host Example	51
2.4	Number of States for the Underthreshold Markov chain	54
3.1	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 10	67
3.2	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 10	69
3.3	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 22; number of hosts = 10	69
3.4	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 26; number of hosts = 10	71
3.5	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 10	71
3.6	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 10	73
3.7	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 44; number of hosts = 10	73
3.8	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 52; number of hosts = 10	75
3.9	Average delay vs. arrival rate ; number of hosts = 10 ; Poisson arrivals	75
3.10	Average delay vs. arrival rate ; number of hosts = 10 ; Poisson arrivals	75
3.11	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 50	79
3.12	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 20	80
3.13	Average delay vs. burst length ; burstiness = 2 ; arrival rate = 22; number of hosts = 20	80
3.14	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 20	81
3.15	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 44; number of hosts = 20	81
3.16	Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 50	82

3.17	Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 0.5; arrival rate = 22; number of hosts = 10	83
3.18	Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 1.0 ; arrival rate = 22; number of hosts = 10	84
3.19	Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 10.0; arrival rate = 22; number of hosts = 10	84
3.20	Average delay vs. threshold computation period ; burstiness = 2 ; number of hosts = 10	86
3.21	Average delay vs. threshold computation period ; burstiness = 4 ; number of hosts = 10	86
3.22	Average delay vs. threshold computation period ; number of hosts = 10; Poisson Arrivals	87
3.23	Average delay vs. threshold computation period; number of hosts = 10; burstiness = 4; arrival rate = 44	87
3.24	Average delay vs. threshold computation period; number of hosts = 10; burstiness = 4; arrival rate = 52	88
4.1	Fraction dropped vs. average laxity and communication delay; immediate scheduling based only on laxity	106
4.2	Fraction dropped vs. average laxity and communication delay; immediate scheduling based only on laxity; 1 slot average processing per task	107
4.3	Fraction dropped vs. average laxity and communication delay; immediate scheduling based only on laxity; 5 slot average processing per task	108
4.4	Fraction dropped vs. average laxity and communication delay; immediate local scheduling based only on laxity ; interconnection network effect included; 10 slot average processing per task	109
4.5	Fraction dropped vs. Number of Nodes; immediate scheduling . . .	110
4.6	Fraction dropped vs. average laxity and communication delay; delayed scheduling	111
4.7	Fraction dropped vs. average laxity and communication delay; delayed scheduling ; threshold = 20	113
4.8	Fraction dropped vs. average laxity and communication delay; immediate scheduling; threshold = 20	114
4.9	Fraction dropped vs. Number of Nodes; immediate scheduling . . .	115
4.10	Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 1	116

4.11	Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 10	117
4.12	Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 1; threshold = 20	118
4.13	Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 10; threshold = 20	119
4.14	Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 100; threshold = 20	120
4.15	Fraction dropped vs. burstiness and communication delay; bursty arrivals, burst length = 1, laxity = 1; no thresholds	121
4.16	Fraction dropped vs. average laxity and communication delay; delayed scheduling; bursty arrivals, burst length = 1, burstiness = 2, no thresholds	122
4.17	Poisson fraction dropped vs. average laxity and communication delay; immediate scheduling with periodic tasks	126
4.18	Fraction dropped vs. average laxity; analysis and simulation	130
4.19	Fraction dropped vs. average laxity; analysis and simulation, 10 nodes	133

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Professor Mario Gerla, and Professor Jack Carlyle for their continuous guidance, encouragement, and support during the elaboration of this work. To the members of the committee, Professor Tomas Lang, Professor Kung Yao, and Professor John Mamer, my deepest gratitude. I would like to acknowledge, also, the contributions of Professor Walter Karplus and Professor Izhak Rubin.

During the course of this work many Professors and colleagues contributed ideas and criticism. To all of them I am grateful. In particular, I would like to thank Professor Richard Muntz from UCLA, Richard Gail from IBM, Professor Edmundo de Souza e Silva from UFRJ, Professor José Monteiro from UFPe (on leave at UCLA), and Arnie Neidhardt from Bellcore. I would like to thank my co-worker Berthier Ribeiro, who implemented the load sharing algorithms and performed the measurements in the testbed.

I would like to thank the CS department staff for their invaluable contribution to this dissertation. In particular, I would like to mention: Verra Morgan, Doris Sublette, Rosie Murphy, Alexandra Pham, Judy Williams, Roberta Nelson, Saba Hunt, and Ann Goodwin.

My parents, Marcos and Aviva, brought me up in love, and instilled in me the drive to learn, to them I shall be forever grateful; without the love, care, friendship, and advise of my wife, Orna, this work would not have been possible: to them I dedicate this dissertation.

Last, I would like to thank the support of Professors Jack W. Carlyle, Walter Karplus, and Terry Gray; of CNPq-Brasil grant 20.0366/86; and of UCLA-IBM Joint Study grant D850915.

VITA

- Nov. 8, 1955 Born, Belo Horizonte, Minas Gerais, Brazil
- 1980 B.Sc., Computer Engineering
Israel Institute of Technology
Haifa, Israel
- 1981-1986 Member of the Research Staff
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
- 1983 M.Sc., Computer Science
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil
- 1986-1990 Member of the Research Staff
University of California
Los Angeles, California
- Summer-1987 Member of the Research Staff
IBM T.J. Watson Research Center
Yorktown Heights, New York

PUBLICATIONS

- Avritzer, A., Bigonha, R. S. (November, 1982). Mzunix: A Multiprocessing and Real Time Operating System (in Portuguese). In: *Anais do II Simposio Sobre o Desenvolvimento de Software Basico para Micros*.
- Avritzer, A. (July, 1983). Multiprocessing Implementation in MZunix (in Portuguese). In: *Anais do III Congresso da Sociedade Brasileira de Computação*.
- Avritzer, A. (September, 1983). MZunix : A Multiprocessing and Time Sharing Operating System (in Portuguese). Masters' Thesis, Universidade Federal de Minas Gerais, Computer Science Dep., Brazil.

Avritzer, A. (July, 1984). Mshell Implementation (in Portuguese). In: *Anais do IV Congresso da Sociedade Brasileira de Computacao.*

Avritzer, A. (November, 1985). DCCIX Implementation in the iAPX286 with Virtual Memory Management (in Portuguese). In: *Anais do V Simposio sobre desenvolvimento de Software Basico para Micros.*

Avritzer, A., Gerla, M., Carlyle, J. W., Karplus, W. J. (July, 1989). Load Balancing in a Distributed Transaction System. In: *Proc. IEEE SICON 89*, Singapore.

Avritzer, A., Gerla, M., Ribeiro, B. A., Carlyle, J. W., Karplus, W. J. (Sep, 1989). The Advantage of Dynamic Tuning in Distributed Asymmetric Systems. In: *Proc. IEEE INFOCOM' 90.*, San Francisco, CA.

Avritzer, A., Gerla, M., Ribeiro, B. A., Carlyle, J. W., Karplus, W. J. (November, 1989). Analytical Modeling of Dynamic Load Sharing Algorithms in Distributed Asymmetric Systems. In: *Proc. IASTED International Conference on Applied Simulation and Modelling.*

Betsler, J., Avritzer, A., Carlyle, J. W., Karplus, W. J. (July, 1988). Performance Modeling and Analysis for a Large Heterogeneous Distributed System: UCLA-SEASnet. In: *Proc. IEEE INFOCOM'89*, Also UCLA CSD Internal Report CSD-880073.

Betsler, J., Avritzer, A., Carlyle, J. W., Karplus, W. J. (August, 1988). Configuration Synthesis for a Heterogeneous Backbone Cluster and PC - Interface Network. In *ACM Computer Science Conference 1989*, also UCLA CSD Internal Report CSD-880074.

Ciciani, B., Dias, D. M., Yu, P.S., Avritzer, A. (September, 1987). Analysis of Protocols for Hybrid Distributed-Centralized Database Systems. IBM Internal Report : RC 13162(#58858).

ABSTRACT OF THE DISSERTATION

Dynamic Load Sharing Algorithms in Asymmetric Distributed Systems

by

Alberto Avritzer

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1990

Professor Mario Gerla, *Co-Chair*

Professor Jack W. Carlyle, *Co-Chair*

We develop analytical models for load sharing policies that schedule jobs based on global system state. In particular, we are interested in large highly asymmetric distributed computing systems. Our approach includes algorithm formulation, testbed measurements, simulation, and analytical modeling. In addition, we have studied the case of high-performance systems for real-time computing, proposing algorithms and carrying out analysis and simulation.

We calibrate our performance model based on measurements carried out on our testbed. We present a new approach for the modeling of dynamic load sharing algorithms in asymmetric systems that employs a global state Markov Chain. The model is validated through simulation and a very good agreement is obtained at low to medium loads. We analyze large systems as well, by bounding the performance measure (long run average delay.)

We present a family of algorithms for dynamic load sharing which cover a

wide range of costs and information gathering. We carry out a sensitivity analysis on their performance as a function of various parameters, including burstiness in the arrival process, number of hosts in the system, and speed ratio between slow and fast processors. We conclude that in asymmetric systems random allocation causes major performance degradation as compared to shortest queue allocation. The proper tuning of thresholds to the hosts capacities and to the arrival process is critical.

We evaluate the use of load sharing algorithms in the control of distributed hard real-time systems, assuming the availability of emerging fast interconnection networks. The traditional approach for scheduling in loosely coupled, hard real time systems has been to use good heuristics for local scheduling. Instead, we use simple local scheduling algorithms and enforce deadline constraints by postulating a fast load sharing interconnection network that can schedule tasks with communication delays on the order of microseconds. Preliminary experiments show that our approach can be used effectively in order to schedule jobs in hard real time systems.

Chapter 1

Introduction

“Continuo aprendiz, mesmo em quarta edição,
e de quanto aprendi foi a melhor lição.”

I am still an apprentice, even in the fourth edition of my book,
and for all that I have learned that was the best lesson.

Carlos Drummond de Andrade, In: Viola de Bolso

1.1 Introduction

With the increasing availability of powerful low end machines the perspective of building highly asymmetric distributed systems has become an exciting reality. These systems may be composed of a few top of the line mainframes, tens of powerful workstations and hundreds of low end machines, manufactured by different vendors and owned by many organizations. Due to the high degree of heterogeneity present, designers are being faced with many difficult challenges.

- How can we take advantage of the huge processing capacity that is now available? Many technical as well as political issues arise when one tries to share huge computing resources that vary greatly in size. While it is easy to see how a huge and expensive computer can be well utilized, it is not clear what is the best operating point when that same capacity is spread over thousands of desk-top computers.

- What are the implications of heterogeneity on the design of a load balancing system? Much care must be employed when managing highly asymmetric systems since the rewards for good design decisions and the penalties for mistakes are much higher than in homogeneous systems. In the latter systems we can ship tasks around almost randomly without incurring a major degradation in system performance, since all the sub-systems have the same capacity and can handle any type of jobs. In such systems, very simple control algorithms were shown to perform almost as well as very sophisticated algorithms [ELZ86a]. This is not the case for highly asymmetric distributed systems where a big job can overwhelm a slow processor while a fast and expensive processor may be underutilized.

Many other difficult issues need to be addressed. What is the cost of controlling this complex environment? What fraction of the total capacity will be invested in overhead for controlling the system? Will that capacity loss be user transparent or will the users pay for it in terms of performance degradation? How can we balance the need to be fair to all users while providing fast response time? If somebody must pay a price for better overall resource utilization, how will the price be shared among all users in a reasonable way?

When the workload is composed from jobs that have very different service requirements we have a new dimension of heterogeneity: traffic heterogeneity.

Algorithms that dynamically control the allocation of work within the system can assist us in answering some of these questions. One of the goals of the present work is the evaluation of cost/benefit tradeoffs involved in using load sharing policies to manage highly asymmetric systems.

The term Load Balancing has been used in the literature to denote algorithms that try to equalize the utilizations on different machines. Load Sharing algorithms

on the other hand try to optimize performance (e.g. minimize the overall average system response time.) In this work we will use the term Load Sharing since our main objective is to optimize the overall average system response time.

1.2 Problem Statement

Given a large asymmetric distributed system we want to design and evaluate dynamic load sharing algorithm that optimize the system performance in respect to a given performance metric .

The traditional performance measure is the average delay.

Other performance metrics are also important depending on the example at hand:

- Real-time systems - fraction of jobs that complete their service satisfying a given deadline constraint.
- campus computing systems - fraction processed remotely, delay of remote jobs, variance in the delay.

A problem closely related to load sharing is the routing of packets in packet switching networks.

There is a mapping between these two problems:

- Processors in the distributed system may be mapped into links in the packet switching network
- Jobs in the distributed system may be mapped into packets in the packet switching network

What are the main differences between the routing of P/S networks and load sharing of distributed systems?

Jobs in distributed systems require service usually at only one processor (we are not considering multiple task systems) while path length in P/S networks is usually multi-hop.

Load sharing is not required for normal operation of distributed systems but is provided as a way to improve performance and computing capacity available to users.

Packet switching networks are designed top-down with flow control issues addressed at an early stage. Distributed systems are usually assembled bottom-up with load sharing policies considered afterwards.

The most popular routing solutions for P/S networks are distributed in flavor due to consideration of scaling, reliability and overheads [Sch87]. Our intuition is that practical dynamic load sharing policies for large asymmetric systems connected by a local area network will tend to be centralized in nature, since the main performance issue is the timely allocation of jobs to underutilized processors. The issue of reliability can be taken care of by providing a spare for the load control site.

1.3 Literature Review

1.3.1 Initial Work

The potential for load sharing in distributed systems has been identified in [LM82]. Livny et al. used a model of k independent systems subjected to Poisson arrivals and exponential service time distributions to show that there exists a high probability of finding an idle host where there is at least one job queued in another host. Initial work in the field addressed static policies where the assignment of jobs to hosts was done in a deterministic way or using precomputed probabilities. Those

policies were applied to both homogeneous systems where the hosts are identical and experience the same workload pattern, and asymmetric systems where some systems may have different capacities and workload characterization.

1.3.2 Static Load Sharing

[dSeSG84] proposed a static load sharing algorithm based on the flow deviation method that computes shipping probabilities in order to minimize average response time. Since job transfer decisions are made without taking into consideration the current state, the algorithm cannot cope with unexpected load patterns. In [dSeSG87] the model is extended to incorporate multiple classes of jobs, multi-tasking within each job and jobs with spawned tasks.

In [TT85] an optimal static load balance algorithm for a load balancing network is considered. They show that if the triangle inequality is assumed (the cheapest route to transmit a job from node a to node b is the direct route that connects node a to b) the optimal solution divides the network in three classes of nodes: sources (ship jobs to be processed remotely), sinks (accepts remote jobs to be processed locally) and neutrals (processes all its local jobs).

1.3.3 Dynamic Load Sharing in Homogeneous Systems

Dynamic load sharing algorithms balance the load when jobs arrive by making an assignment that is a function of the current state. Most of the work done in dynamic load sharing considers homogeneous systems.

[ELZ86a] showed that very simple dynamic load sharing algorithms were able to improve performance in homogeneous distributed systems. The main contribution of their paper was to show that very little improvement in performance compared to what was already achieved by the simple dynamic algorithms could be achieved

when highly tuned algorithms were employed. [ELZ86b] divided the dynamic load sharing policies into sender-initiated and receiver-initiated classes. Sender Initiated should be used in systems with light to moderate load. Receiver Initiated should be used in high loads. It was stressed that the main problems of complex policies come from overhead costs and unpredictable behavior when confronted with inaccurate information.

[LT86b] define 3 dynamic load sharing algorithms for homogeneous systems that use only local information. They divide the jobs in a host in two classes: local and remote. The 3 algorithms are threshold type based in the length of the local job queue and give different priorities to the local and remote jobs. They conclude that the selfish algorithm(priority to local jobs) have worse performance since remote jobs are delayed until the queue of local jobs is exhausted. The altruistic(priority to remote) is the best since the most expensive jobs are done quickly. They don't degrade performance of the local jobs because their relative number is small. The balanced priority policy(priority to the type that has most jobs in queue) has similar performance to the altruistic but is a little worse at high utilizations(more than .7)

In [LT86c] a distributed algorithm is presented were each host compares its incremental delay to the incremental delay of other hosts in order to determine the optimal value of its threshold. The algorithm is iterative and adapts to changes in the job arrival rates.

[Zho87] performed a sensitivity analysis of seven dynamic load sharing algorithms in a homogeneous environment. Instead of driving the simulations by probabilistic distributions he used a job trace, measured at the production machine running 4.3 BSD. The main conclusion of the work were the following:

- algorithms using periodic and non-periodic information policies produce sim-

ilar performances,

- the use of a central host instead of using a distributed decision scheme to make all placement decisions yields the best performance, since the central scheduler could make consistent decisions,
- the improvement achieved by using load sharing levels off around 33 hosts,
- load sharing can be used effectively even when some jobs are pinned to particular hosts,
- load sharing has a beneficial effect on every host,
- load sharing improves performance over a wide range of overhead assumptions,
- large jobs benefit more from load sharing than short jobs since the overhead incurred is divided over a long useful life.

1.3.4 Dynamic Load Sharing in Asymmetric Distributed Systems

As for load sharing in asymmetric systems, there have been recently a number of contributions. We review some of them in the sequel.

[Gre88] analyzed a load sharing algorithm based on thresholds for a asymmetric system where each host has two servers: a processing server and a communication server. Each host is assigned a different threshold. He showed through simulation that the arrival rate from other hosts could be reasonably well approximated as a Poisson process even for a small number of sources. The variance of the simulated process was higher than an exponential interarrival rate. This happens because the overflow process from hosts is more bursty than the Poisson process.

[LT86a] defines and analyzes a dynamic load sharing policy for a asymmetric system composed of homogeneous clusters that uses only local information. A queueing model of the system was formulated and solved to produce a closed form expression that is used to pose a non-linear optimization problem with the response time as optimization metric and the thresholds of each host as the optimization variable. It was shown that the remote arrival process at each host in their model approaches a Poisson process as the number of clusters goes to infinity, maintaining a constant number of hosts per cluster.

In a recent paper [MTS89], the effect of delays in the network interface in limited probing dynamic load sharing algorithms was studied. The model used was the same as in [LT86a]. The performance modeling was done by constructing the algorithm Markov Chain and was solved using the Matrix-Geometric solution technique [Neu81]. An evaluation of the algorithms was done by measuring the sensitivity of the response time to the variation in the thresholds, change in probe limit and biasing in the probe probabilities. They concluded that the benefits of load sharing and the sensitivity of the response time to threshold tuning become more important with increased heterogeneity.

[RP88] considered a queueing system composed of n exponential servers each with a different state dependent service rate. Customers arrive according to a Poisson process. They succeeded in defining an ideal system that provides an upper bound on performance. In that system each server optimally selects a customer fraction independent of the other server choices.

[SW88] reported a asymptotic analysis of large asymmetric queueing systems. Their model considered a single queue multi-server system. They simplified their system by considering that server speeds can be grouped in two classes: fast and slow. They model the system as a $M/M/m/K$ queue where K is the sum of

the threshold to the number of servers. They concluded that in the limit as the number of servers goes to infinity, the policy of never queueing achieves optimal performance while the policy that tries to optimize the individual performance of each job ('greedy') does not.

1.3.5 Aggregation and Decomposition

The aggregation and decomposition method is a well known approach that was introduced into the area of Computer Networks by Courtois. The approach was first introduced by Simon and Ando in the area of Economics [Cou77]. Informally, the decomposition approach applies to a class of the so called near-complete decomposable systems [Cou77], [Cou82], [CS84], [CS86]. In such systems the infinitesimal generator matrix \mathbf{Q} can be decomposed into square matrixes $\mathbf{Q}(\alpha, \alpha)$ such that transitions within each aggregate group are more frequent than between different groups. The approach consists of 3 main steps: decomposition, aggregation, and unconditioning. In the decomposition phase, the off diagonal row sum is distributed among the non zero row terms of the Matrixes $\mathbf{Q}(\alpha, \alpha)$. Each aggregate generates an independent system of equations which is solved to obtain the steady state distribution conditioned on the system being in a state of the block matrix $\mathbf{Q}(\alpha, \alpha)$. In the aggregation phase, an infinitesimal generator \mathbf{A} is constructed to represent the transition rates between aggregates. It is then solved to generate the steady state probability of being in an aggregate. In the unconditioning phase, the conditional steady state vector obtained in the decomposition phase is unconditioned by the steady state vector obtained in the aggregation phase to yield an approximation on the steady state probability vector of the original matrix \mathbf{Q} . The main difficulty in the approach resides in the way to distribute the off diagonal row sum in order to minimize the error introduced by the approximation. Courtois ad-

dressed this problem in [Cou82], and [CS84]. In [Cou82] the error minimization in decomposable stochastic models was considered. Let \mathbf{P} be a finite, non-negative, normalized stochastic matrix of the following form:

$$\begin{pmatrix} Q_1 & E \\ F & G \end{pmatrix}$$

Let \mathbf{Q}_1^* be constructed by the distribution of the row sums of \mathbf{E} into the non-zero rows of \mathbf{Q}_1 . In [Cou82] Courtois showed that if $\mathbf{Q}_1^* = \mathbf{Q}_1 + \mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}$, then the decomposition approach yields exact results. Usually, the computation of $\mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}$ is prohibitive, since it involves the inversion of a matrix that have only one aggregate less than the full system. In [CS84] it was shown that if \mathbf{E} and \mathbf{F} are fully known tight bounds on $\mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}$ can be obtained as follows:

$$[\mathbf{EF}]_{ij} \leq [\mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}]_{ij} \leq [\mathbf{E}\mathbf{1}]_i - \sum_{k \neq j} [\mathbf{EF}]_{ik}$$

Furthermore, in [CS84] looser bounds are given in the case where \mathbf{F} is not fully known.

In [MdSeSG89] an approximation method to reduce the Markov chain state space involved in the numerical solution of real models was presented. The method involved the detailed modeling of the part of the Markov Chain where most of the mass probability was located, and a aggregated representation of the rest of the Markov chain. Through the computation of upper and lower bounds on the performance, a bound on the error incurred was obtained.

1.3.6 Real-Time Systems

In Section 4.2 we present a literature review on real-time systems.

1.4 Dissertation Outline

The Dissertation is composed of the following Chapters:

1. In Chapter 1 we present the motivation for the work and the literature review,
2. In Chapter 2, we obtain parameters for our performance modeling by performing measurements on our testbed (for campus computing systems) and by running simulations(for the campus computing system). In order to assess the cost/performance characteristics of those systems we present algorithms for delay/throughput estimation as well as upper and lower bounds in the performance of those systems.
3. In Chapter 3 we present a family of algorithms for dynamic load sharing that covers a range of costs and information gathering, and perform a sensitivity analysis of the average delay on a range of parameters.
4. In Chapter 4 we study the use of load sharing algorithms in order to schedule task in hard-real time systems.

1.5 Contributions

- *New Algorithms for Load Sharing in Asymmetric Systems*

The use of varying thresholds to reflect the result of a centralized optimization based in the flow deviation approach to perform a dynamic tuning of load sharing algorithms to the instantaneous global load in the system has not been reported in the literature reviewed. The use of this approach substitutes the use of ad-hoc heuristics by the formulation of a mathematical model that expresses the problem constraints. [Zho87] reports the use of the

centralized/distributed approach for the allocation of tasks in homogeneous distributed systems. [Rud76] use also the idea of centralized/distributed control in the delta routing algorithm. [LT86c] propose the use of distributed algorithms to iteratively compute the optimal thresholds.

- *Bound Computation for Load Sharing Systems*

The analytical modeling of multiple queue asymmetric systems is very difficult since the state space is huge. We presented a new approach for the modeling of dynamic load sharing algorithms in asymmetric systems that employs a global state Markov Chain. The model was validated through simulation and a very good agreement was found at low to medium loads. We managed to analyze large systems as well, by bounding the performance measure (long run average delay.) The lower bound obtained is tight , since the states discarded have very low probabilities. The upper bound proposed is tight as well.

- *Sensitivity Analysis of Algorithms for Load Sharing in Asymmetric Systems*

We have compared the cost/performance of different algorithms employing the cost measured in our testbed. We have demonstrated that the proper tuning of thresholds to the hosts capacities and arrival process is critical. The mismatch between host capacities and thresholds is responsible for a major degradation in the performance of fixed thresholds algorithms, as compared to the same algorithms with tuned thresholds. Dynamic tuning of thresholds exhibits the best performance over all algorithms analyzed for range of parameters. We evaluated the sensitivity of the algorithms to the speed ratio between fast and slow processor. This is a very important measure since it will indicate how the algorithms behave in face of unpredictable system

behaviors that affect processor speed. We concluded that dynamic tuning outperforms fixed thresholds for all speed ratios.

- *Load Sharing in Hard Real-Time Systems*

We have proposed an alternative approach for scheduling in hard real-time systems. We propose to use simple local scheduling algorithms and enforce deadline constraints by using a fast load sharing interconnection network that can schedule tasks with communication delays on the order of microseconds. We have evaluated our approach under Poisson and bursty arrivals assumptions. We have simulated the effect of system size, different local scheduling algorithms, burstiness, burst length, thresholds, synchronous allocation due to the interconnection network, and of a job mix composed of Poisson and periodic tasks. We have shown that load sharing can be used effectively in order to schedule jobs in hard real time systems. A simple inexpensive hardware base algorithm for local scheduling can be used together with a fast interconnection network. However, the burstiness of the process must be carefully studied, since it has a major impact on performance. Synchronous remote allocation can be used effectively to reduce the fraction of tasks that are dropped. The current approach to improve performance has been to reduce processor utilization. We have shown that reducing communication delay is an effective alternative.

- *Performance Modeling of Hard Real-Time Systems*

We have proposed a new approach for modeling distributed real-time systems with exponentially distributed deadlines. We managed to obtain a very good approximation by applying a $M^k/D/c$ model.

Chapter 2

Performance Modeling of Campus Computing Systems

“Digo: o real não está na saída nem na chegada:

ele se dispõe para a gente é no meio da travessia.”

I say: the real is not in the departure or in the arrival:

it reveals itself to us during the journey.

João Guimarães Rosa In: Grande Sertão: Veredas, pp 62

2.1 Introduction

In this chapter we develop an analytical performance modeling approach for load sharing policies that schedule jobs based on global system state. In particular we are interested in modeling highly asymmetric systems. Those systems are composed of hosts of many different speeds which are subjected to heterogeneous workloads. They are object code compatible in the sense that programs can be run everywhere. Examples of such systems are IBM-clusters running AIX/370, workstation networks composed of different cpu speed models, and distributed databases. One of the most critical decisions in the performance modeling of load sharing algorithms is the way to account for overhead in remote job initiation. In this work we use a three-pronged approach, in which experiments, analytical modeling, and simulation reinforce one another. We implement the algorithms in a LAN testbed composed of IBM-RTs, HPs, and SUN workstations (all running variations of the UNIX operating system) and measure the overhead incurred to

initiate jobs remotely. We then include the measured parameters in our analytic and simulation models.

The implementation of load sharing algorithms can be done at two levels:

1. Inside the operating system either at the kernel or shell level. This option is the most efficient in terms of overhead incurred for remote job initiation. It has however a huge drawback, that is, the need for compatibility between the versions implemented. Different vendors will have to agree to a common standard if their load sharing operating systems are supposed to talk to each other. This and the need to modify very complex programs are the probable reasons why load sharing operating systems are not commonplace, despite the great performance advantages that has been demonstrated in the literature [Zho88].
2. At the user level as a shell front end. The overhead incurred to initiate a job remotely at this level will be an order of magnitude larger, since we will use high level primitives to initiate jobs. The advantages however are multiple. First, we can easily implement a job control manager that handles systems from many different vendors. Second, load sharing can be made transparent to machines and users, since only machines running clients to the load sharing algorithm will be involved. Third, users can be excluded or included in the load sharing population by having them run the shell front end.

A well known result in the area of load sharing for **homogeneous** distributed systems [ELZ86a] is that very little improvement in performance can be achieved with highly tuned algorithms as compared with simple dynamic algorithms. In this chapter we show that in highly **asymmetric** systems this is not the case, since the penalty for bad decisions (sending a job to an overloaded and slow ma-

chine) and the rewards for the timely scheduling of jobs have a major impact on performance. If high overheads are incurred to initiate a job remotely (as in the user level load sharing approach) the penalty for bad decisions are even higher. To this end, we consider two similar algorithms that route jobs to the shortest queue: one, with fixed and identical thresholds for all sites; the other, with quasi-static thresholds. The huge improvement observed in the performance, when using quasi-static thresholds, indicates that in highly asymmetric systems sophisticated algorithms must be employed. Furthermore, the use of load sharing algorithms in large asymmetric networks will drive users to run more and more complex applications (e.g. distributed simulations) and utilize the available resources in many new ways. The optimal control of this large and costly environment will be critical in order to provide quality service in a user transparent way.

The term *load balancing* has been used in the literature to denote algorithms that try to equalize the utilizations on different machines. *Load sharing* algorithms on the other hand try to optimize performance (e.g. minimize the overall average system response time) by routing jobs to underutilized hosts. In this work we will use the term load sharing since our main objective is to optimize the overall average system response time. Static load sharing algorithms assume knowledge of the offered workload in order to compute routing probabilities that minimize average response time [dSeSG84]. Since job transfer decisions are made without taking into consideration the current system state, the algorithm cannot cope with unexpected load patterns. In this chapter we use the static load sharing approach in order to periodically compute the optimal routing probabilities that drive the system in a near optimal state. The offered workload is measured and the optimal routing probabilities are converted into the optimal thresholds, which are the fraction of the instantaneous load that would be assigned to a host by

the routing probabilities. Dynamic load sharing algorithms balance the load when jobs arrive by making an assignment that is a function of the current state. In [ELZ86b], the dynamic load sharing policies were classified into sender-initiated and receiver-initiated policies. Sender initiated should be used in systems with light to moderate load. Receiver initiated should be used at high loads. It was stressed that the main problems with using complex policies come from overhead costs and unpredictable behavior when confronted with inaccurate information. In our approach, the input parameters for the computation of the optimal routing probabilities will be measured periodically, thus partially overcoming the problem of inaccurate workload characterization.

Zhou [Zho88] performed a sensitivity analysis of seven dynamic load sharing algorithms in a homogeneous environment. Instead of driving the simulations by probabilistic distributions, he used a job trace measured at the production machine running 4.3 BSD. It was found that the use of a central host to make all placement decisions yields the best performance, since the central scheduler could make perfect decisions. The use of a central site to compute the optimal dynamic thresholds has many advantages over the decentralized approach. The central site can be a special purpose computer optimized for the routing function and designed to be fault-tolerant [Avi67]. The central site can also contribute to system reliability by keeping track of the available resources and adjusting the optimization to account for down sites. The availability of a central optimization makes it easier to redefine system objectives, to maintain, and to upgrade the system.

In the present work we employ a centralized/distributed approach by computing the optimal thresholds periodically at the central site and shipping them to each local host. We will refer to this approach as *Dynamic Tuning*. The local sites

accept jobs to be run locally by using only local information. Remote requests are directed to the central site. We employ as performance measure the average total response time.

The main contributions of this chapter are the following:

- Derivation of analytical bounds to compute the average delay in large asymmetric systems under dynamic load sharing control,
- Experimental derivation of parameters which affect the implementation of load sharing policies at the user level,
- Demonstration that load sharing is effective even in the presence of high overheads,
- Demonstration that in the asymmetric environment under study, carefully tuned algorithms for load sharing provide a significant improvement in performance over simpler algorithms.

The outline of the rest of the chapter is as follows: In Section 2.2 we describe the environment used in the measurements and the model used in the algorithms evaluation through analysis and simulation. In Section 2.3 we present a new self tuning load sharing algorithm that employs dynamic thresholds, and its performance evaluation through measurement and simulation. In Section 2.4 we present the analytical modeling of the testbed. In Section 2.5, 2.6, 2.7, and 2.8 we present bounding techniques to compute the average delay in large asymmetric systems under dynamic load sharing control.

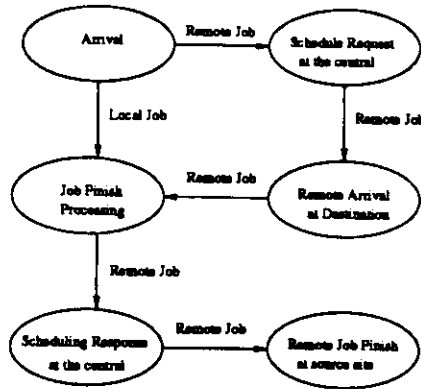


Figure 2.1: Simulation Model

2.2 Environment and Model

We performed measurements on the testbed available at the UCLA Computer Science Department. The testbed was composed of 5 IBM PC/RTs (4 mips), 5 HP-9000 (0.3 mips), and the residual capacity of 3 Suns when the production workload is present (0.4 mips.) The sites are connected by a local area network (Ethernet.) The load sharing control algorithm was implemented at the user level. This environment enabled us to demonstrate the usefulness of load sharing even when high overheads are present. We found that a processing overhead (due to software protocol layering) of 0.3 secs and a communication delay of 20ms is incurred for every remote job. The simulation model was the following:

1. Jobs arrive at the local sites according to the arrival process under study (in this paper we adopted a Poisson arrival process.)
2. The local scheduling algorithm is employed and the job is classified as local or remote.
3. Local jobs are queued in the local cpu according to FCFS scheduling and are run for a exponentially distributed number of instructions with 1M instruc-

tions average.

4. Remote jobs suffers 0.125 secs (processing overhead) and 5 ms communication delay to get to central site. There, they queue for the central cpu and use 100k instructions processing overhead and 5 ms communication delay to be scheduled for remote execution.
5. Upon arrival at the remote site, they are queued in the remote cpu according to FCFS scheduling (one queue for local and remote jobs) and are run for an exponentially distributed number of instructions with 1M instructions average.
6. Once a local job completes its running time it is done. A remote job suffers another 0.125 secs (processing overhead) and 5 ms communication delay to get to the central site and must queue again for the central cpu to consume another 100k instructions processing overhead and 5 ms communication delay, before being sent to its originating site for completion.

In a user level implementation the main overheads are in the cpu, as the Ethernet is usually underloaded. We account for the communication delays in the Ethernet, but we don't model any queueing for the cable.

2.3 A Self Tuning Dynamic Load Sharing Algorithm

In this section we introduce a new load sharing algorithm for distributed asymmetric systems. It is a threshold type algorithm that varies the thresholds dynamically, adjusting them to the load in order to keep an optimal number of tasks in each site. The algorithm periodically gathers information on the global load in the system and analytically computes an optimal task allocation in order to

minimize the total average response time. This optimal allocation is a function of the total measured load. The site thresholds are set to be equal to the optimal task allocations.

The two main decisions implemented in the algorithm are the thresholds computation and the task allocation and shipping. A designated site is responsible for computing the global load and the thresholds that each site will use in the near future. Periodically all sites send to the central site the number of arrivals they received from the external world in the last time slice. Using a steepest descent technique [FGK73] the central site solves the following optimization problem.

Let,

λ_i^0 – average arrival rate to site $i, i > 0$,

λ_i^a – average remote arrival rate to site i , due to load sharing, $i > 0$,

λ_i^d – average remote departure rate from site i , due to load sharing, $i > 0$,

λ_i – average rate of jobs processed at site $i, i > 0$,

λ_0 – average rate of jobs processed at the central site (processing overhead),

τ_i^* – optimal threshold for site i ,

T_i – average job delay in site i ,

T – total average delay,

n_i – average number of jobs in site i , site 0 is the central site,

n_d – total average number of jobs due to comm. delay,

c_d – cost due to communication delay (comm. delay)

c_p – cost due to processing overhead (to send a job remotely)

μ_i – average processing rate at site i ,

μ_i^j – average job processing rate at site i ,

γ – total external arrival rate to the system.

TASK ALLOCATION PROBLEM

Given: $\{\lambda_i^0, \mu_i, c_d, c_p\}$

Minimize: T

With Respect to: $\{\lambda_i\}$

$$\begin{aligned}\gamma &= \sum_{i>0} \lambda_i^0 \\ \lambda_i &= \lambda_i^0 + \lambda_i^a - \lambda_i^d, i > 0 \\ \lambda_0 &= 2 \sum_{i>0} \lambda_i^a \\ \sum_i n_i &= \sum_i \frac{\lambda_i}{\mu_i - \lambda_i} \\ n_d &= 2\lambda_0 c_d \\ \mu_0^{-1} &= (\mu_0^J)^{-1} \\ \mu_i^{-1} &= (\mu_i^J)^{-1} + c_p \frac{\lambda_i^a + \lambda_i^d}{\lambda_i}, i > 0\end{aligned}$$

Using Little's result:

$$T = \frac{\sum_i \frac{\lambda_i}{\mu_i - \lambda_i}}{\gamma} + \frac{2\lambda_0 c_d}{\gamma}$$

Solving this optimization problem we obtain a set λ_i^* . To obtain the optimal thresholds we compute:

$$\tau_i^* = \frac{\lambda_i^*}{\mu_i - \lambda_i^*}, i > 0$$

These will be real numbers that must be rounded to yield the approximate sub-optimal thresholds we are after.

Task allocation and shipping is as follows:

1. Upon an external arrival to a site, if the number of tasks present is less than its threshold, the arrival is accepted as local.
2. Otherwise, a scheduling request together with the task is sent to the central site. The central site allocates remote requests to the site that has the biggest positive difference between its threshold and its queue length and ships the task directly to that site. If no site can satisfy the request the task is returned to the source site.
3. Upon a remote task completion, the results are shipped to the central site and forwarded from there to the originating site.

The decision to route tasks and results through the center (instead of having them transmitted directly from the source to the destination) was made to avoid the delay incurred in establishing a connection every time an allocation is made. Thus, for our purposes the network is a virtual star with every site connected to the central site. In order to make proper task allocation the central site must have a consistent view of all sites. To this end, queue length measurements are periodically broadcast from each site to the central site. The idea of combining centralized/distributed algorithms was used previously in [Zho88] for dynamic load sharing in homogeneous systems, and in the *delta routing algorithm* [Rud76] for routing in packet switching networks.

The performance evaluation of the dynamic tuning algorithm was done by comparing it to an algorithm where all thresholds were fixed and identical to one. This is the best reported algorithm for homogeneous systems [Zho88]. The

example reflects a situation where a large number of homogeneous workstations is sharing their residual capacity, defined as the fraction of the processing power not used by the workstation owner. In this case thresholds can not be set a priori to be proportional to the (unknown) available processing power. Figure 2.2 reports typical measurement and simulation results for the average delay of the *dynamic thresholds* algorithm and of the *fixed thresholds* algorithm. Our conclusions from the evaluation are the following:

- The self tuning algorithm is able to improve performance for a wide range of arrival rates. The dynamic threshold algorithm is able to reduce the average delay to more than 50% of the value obtained when the fixed threshold of 1 is used. Furthermore, it can support an arrival rate 20% larger than the fixed threshold algorithm.
- Load sharing implementation at the user level incurs high overhead (0.3 secs cpu overhead/job initiation and 20 ms communication delay.) Load Sharing is still very attractive in the environment considered: asymmetric systems with many cpus speeds running jobs with 1M instructions average.
- Fixed and uniform thresholds algorithms in asymmetric environments are highly sensitive to tuning. The thresholds must be well matched to the service demands and capacities of the site in order to achieve good performance. This is the main drawback of this kind of algorithm since workloads must be very well specified in advance. Furthermore the algorithm fails to adapt to varying workloads. The dynamic computation of thresholds solves this problem.
- The dynamic computation of thresholds leads to a better utilization of the more expensive resources since it tends to make better use of the higher capacity servers.

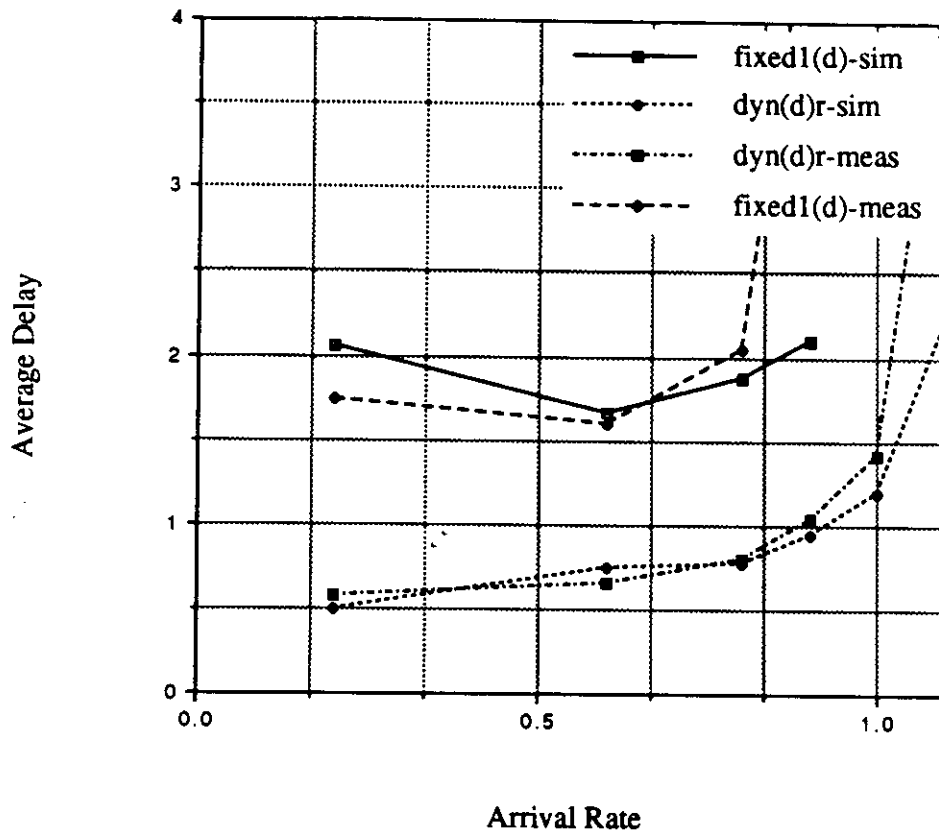


Figure 2.2: 13 Hosts, simulation and measurements, dynamic thresholds vs. fixed thresholds, average delay vs. arrival rate

- The simulation model gives very good performance prediction for the dynamic tuning algorithm but disagrees in about 30% for the fixed thresholds algorithm before the knee of the curve. The reason for the disagreement in the fixed threshold algorithm is that it exhibits a larger statistical variation, since the slower hosts are run at very high utilizations.

2.4 Analytical Modeling of the Testbed

In this section we will describe the analytical approach used in order to model and evaluate the *Dynamic Tuning Algorithm* in our testbed. The main challenge is the explosion of the number of states of the Markov Chain when the system grows large.

An previously reported approach for the modeling of dynamic load sharing algorithms ([ELZ86a],[LT86c]) applies decomposition to generate a separate Markov Chain for each site. The main simplifying assumption was that the remote arrival process due to threshold overflow is Poisson. The key advantage of that approach was that the number of states in each Markov Chain is very small. The decomposition approach however has a drawback: it cannot properly model an algorithm that allocates jobs based on global information. In order to cope with this problem, we propose to model the system using a global Markov Chain. We conclude this section by comparing the analytical results with simulation. In Section 2.5, we show a method for avoiding the combinatorial state explosion typical of global models when size becomes large, using the bounding techniques as proposed in [MdSeSG89].

2.4.1 Markov Chain Modeling

In order to provide a **first** order approximation on algorithm performance, we assumed that the slower site thresholds were fixed and equal to zero and the faster site thresholds were fixed and equal to 2.

In order to maintain the number of states in our Markov chain bounded, we assume that,

- fast processors will accept a maximum of four jobs (two above the threshold).

- the slow machines are aggregated and modeled as a single M/M/1 server (with cumulative service rate.) A maximum of three jobs are permitted in this server. The slow machines receive jobs only when all fast processors are at or above their thresholds (the slow processors thresholds are all zero). This approach will provide a **lower bound** on the delay performance of the slow machines.
- a job arriving when the queues are full (i.e. 4 for each fast process and 3 for the cluster of slow processors) is discarded. This is also an ‘optimistic’ approximation.

The state of the system is $S = (N1, N2, N3, N4, N5)$ where,

- $N1$ = number of fast processors with exactly one job,
- $N2$ = number of fast processors with exactly two jobs,
- $N3$ = number of fast processors with exactly three jobs,
- $N4$ = number of fast processors with exactly four jobs,
- $N5$ = number of jobs in the slow processors.

Note that S completely defines the system, since the scheduling of arrivals depends only on the difference between the site threshold and its queue length, and, the service rate of a state depends only on how many processors of a certain class are busy.

Let,

- N be the total processor number,
- N_f be the total fast processor number,
- N_s be the total slow processor number,

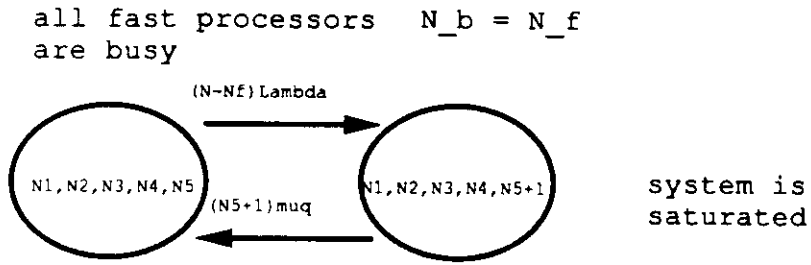
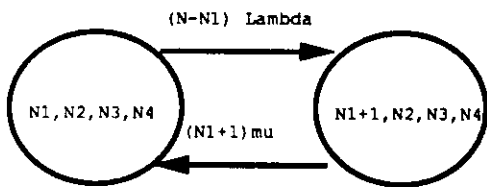


Figure 2.3: Arrivals and departures at the slow processors when the system is saturated

idle processors $N_b < N_f$



idle processors $N_b < N_f$

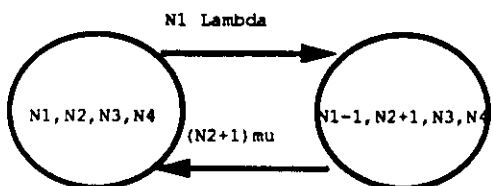
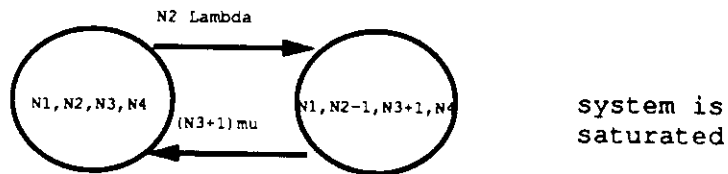
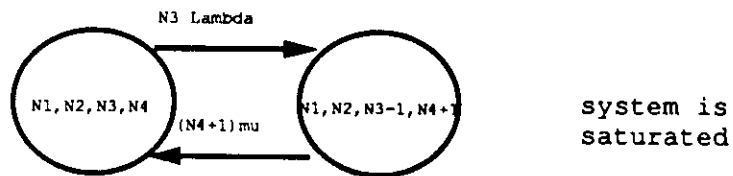


Figure 2.4: State transitions when there are idle fast processors

all fast processors are busy $N_b = N_f$
 busy



all fast processors $N_b = N_f$
 are busy



all fast processors $N_b = N_f$
 are busy

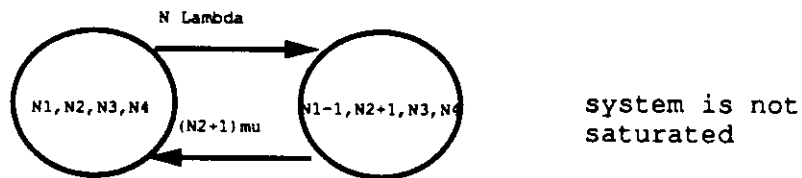


Figure 2.5: State transitions when all fast processors are busy

- λ be the uniform average arrival rate at each processor from the external world,
- μ be the fast processor job departure rate,
- μ_s be the slow processor job departure rate,
- μ_q be the equivalent departure rate of the lumped server that represents the slow processors($\mu_q = \mu_s N_s$) ,
- $S = (N1, N2, N3, N4, N5)$ be a state in the Markov Chain,
- $S + i - j$ be the state $(N1, \dots, Ni + 1, \dots, Nj - 1, \dots)$,
- N_b be the number of busy fast processors, that is, $N1 + N2 + N3 + N4$.

The transitions can be divided in the following groups:

1. (there are idle fast processors) $N_b < N_f$:
 - arrivals to idle fast processors occur at a rate of $(N - N1) \times \lambda$, causing the transition from State S to $S + 1$,
 - arrivals to fast processors with one job occur at a rate of $N1 \times \lambda$, from State S to $S - 1 + 2$.
2. (all fast processors are busy) $N_b = N_f$:
 - (system is saturated – all processors are at or above their thresholds)
 - arrivals to the Ni fast processors occur at a rate of $Ni \times \lambda$, from State S to $S - i + (i + 1)$ ($1 < i < 4$),
 - arrivals to the slow processors occur at a rate of $(N - N_f) \times \lambda$, from State S to $S + 5$.

- (system is not saturated) arrivals to fast processors with one job occur at a rate of $N \times \lambda$, from State S to $S - 1 + 2$,
- 3. Departures from a fast processor with i jobs. Occurs at a rate of $N_i \times \mu$, from State S to $S + (i - 1) - i$.
- 4. Departures from slow processors. Occurs at a rate of $\mu_q \times N5$, from State S to $S - 5$ ($N5 > 0$).

The main Markov Chain transitions are described in Figure 2.3 and Figures 2.4 and 2.5 (state $N5$ is omitted for simplicity). From the steady state probabilities (that are obtained using a Markov Chain solver) we can obtain the average delay as follows:

- Let $P(S)$ be the probability that the system is in state S of the Markov Chain,

- Let $n_{fp}(S)$ be the number of jobs in the fast processors in state S .

$$n_{fp}(N1, N2, N3, N4, N5) = N1 + 2 \times N2 + 3 \times N3 + 4 \times N4.$$

Compute N_{fp} the average number of jobs in the fast processors .

$$N_{fp} = \sum P(S) \times n_{fp}(S).$$

- Let $n_{sp}(S)$ be the number of jobs in the slow processors in state S .

$$n_{sp}(N1, N2, N3, N4, N5) = N5.$$

Compute N_{sp} the average number of jobs in the slow processors .

$$N_{sp} = \sum P(S) \times n_{sp}(S).$$

- Compute ps the probability that the system is in saturation (i.e. all processors are at or above their thresholds.) $ps = \sum P(S), \{S | N2 + N3 + N4 = N_f\}$.

- Using Little's result, compute T_f , the average job delay in a fast processor. The average arrival rate to a fast processor is $N\lambda$ if the system is not saturated, and is $N_f\lambda$ if the system is saturated. However, all arrivals that find 4 jobs in a processor are discarded.

$$T_f = \frac{N_{fp}}{N\lambda(1 - ps) + N_f\lambda ps - \lambda N 4 |_{sys.saturated}}$$

- Using the M/M/1 formula, compute T_s , the average job delay in a slow processor.

$$T_s = (\mu_s - \lambda ps)^{-1}$$

- Compute T , the average job delay.

$$T = \frac{N_{fp}T_f + N_{sp}T_s}{N_{fp} + N_{sp}}$$

2.5 Bounding Techniques

The previous approximate approach is applicable to small systems only. As system size increases, the number of states in the complete Markov Chain grows combinatorially with the number of sites in the system modeled. In order to permit a scaling of system size, we will compute lower and upper bounds on the system performance [MdSeSG89]. If these bounds are close to each other, we have obtained a good estimate of the desired performance, and we know the error of such estimate.

First, we compute the upper bound. The dynamics of job allocation in the system is described below. Jobs that arrive to the slow processors see the fast processors cluster as a gated system. Namely, when the cluster is below threshold (i.e. at least one fast processor is below the threshold) the arriving job will

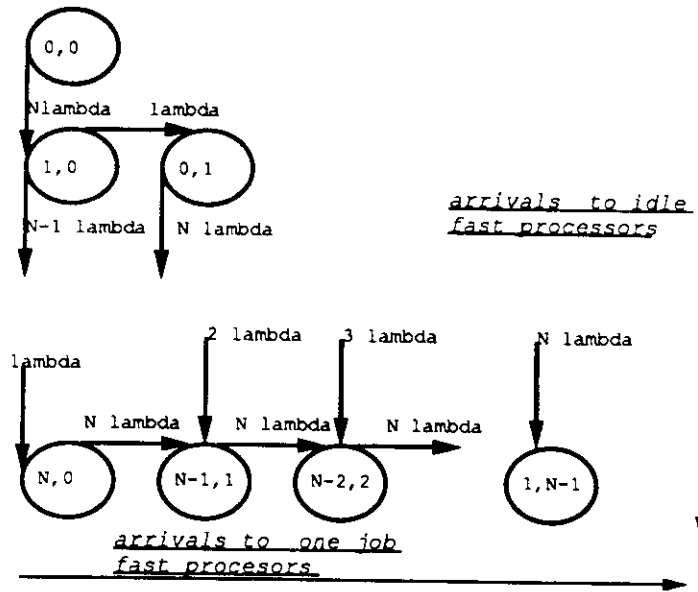


Figure 2.6: Markov chain for the underthreshold state, arrivals

be allocated to the fast processor that has the biggest positive difference between threshold and queue length . On the other hand, if the fast processors cluster is at threshold (all processors at threshold) or above threshold (at least one processor above threshold), the job will be allocated to be run in the local slow host (all slow host have a threshold of zero.) Jobs that arrive to fast processors will behave similarly. Namely, when the fast processors cluster is at, or above threshold these jobs will run locally in the originating processors. When the system is below threshold, the job will be allocated to the fast processor that has the biggest difference between threshold and queue length.

As we shall see, we can decompose the above threshold model into two sub-models corresponding to the fast and slow processors clusters.

Let us study the fast processors cluster first. When the fast processors cluster is at saturation or below, the state of its Markov Chain is given by $N1$ (the

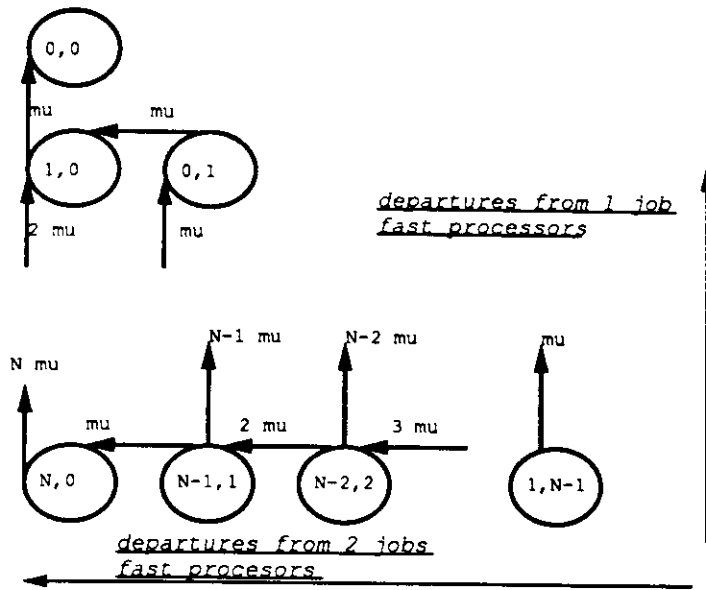


Figure 2.7: Markov chain for the underthreshold state, departures

number of fast processors with 1 job) and by $N2$ (the number of fast processors with 2 jobs.) All fast processors have the same threshold of 2. This Markov chain is the same as the one introduced in the previous section with states $N3$, $N4$ and $N5$ removed and is shown in figures 2.6, and 2.7. In order to model the behavior of the system above threshold, we use the following approach. First, recall that the system is above threshold if there is at least one fast processor queue above threshold (this, of course, implies that all queues are at, or above threshold.) We assume that, as long as this condition persists:

1. no job is reallocated to another processor (i.e. all jobs are run locally); this is consistent with our load balance algorithm definition.
2. each processor is allowed to process only jobs above threshold; this is clearly a conservative assumption, since it forces the fast processors with only 2 jobs

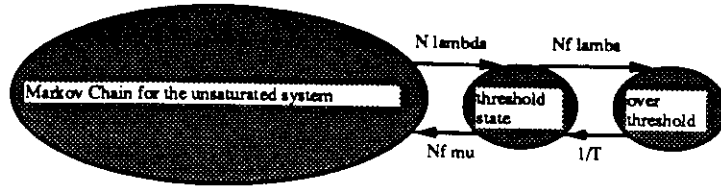


Figure 2.8: Fast Processors Cluster Behavior

in the queue to remain idle until the system returns to the threshold state.

Thus, the first arrival to a system in threshold state ‘freezes’ the processing of all fast processor jobs found in the system. This arrival generates a busy period for the fast processors cluster, where all arrivals are processed locally. The busy period terminates when the system returns to the threshold state. This behavior will be modeled by aggregating all above threshold states and adding to the Markov Chain an aggregate above threshold state that will model the busy period for the fast processor cluster. Arrivals to the above threshold state occurs at a rate $N_f \lambda$ from the threshold state (i.e. $N_2 = N_f$.) Departures from this state occur at a rate $1/T$ (where T is the average length of the N_f fast processors cluster busy period.) Figure 2.8 illustrates this fast processors cluster behavior.

In order to compute T we use the following renewal theory approach: Given that we have Poisson arrivals, the points where the N_f queues become idle are renewal points for the N_f fast processors cluster busy period. Let,

P_0 be the probability of a empty system

I be the mean length of the idle period

T be the mean length of the busy period

$$I = 1/(\lambda N_f)$$

$$P_0 = (1 - \rho)^{N_f}$$

$$P_0 = I/(I + T)$$

Solving for T yields:

$$T = (\lambda N_f (1 - \rho)^{N_f})^{-1} - (\lambda N_f)^{-1}$$

with $\rho = \lambda/\mu$.

Having computed T we can solve the Markov Chain and obtain the steady state probabilities for the number of jobs in the fast processors when the system is at threshold and below. The average number of fast processor jobs given that the system is in above threshold state can also be easily computed as follows. The average number in N_f independent M/M/1 queues is clearly $N_f \rho / (1 - \rho)$. We note, however, that in the above threshold state at least one of the queues is non-empty. Thus, we must compute $[Q/\text{non empty}] = \text{average queue length given that at least one of the queues is non empty}$. Clearly,

$$N_f \rho / (1 - \rho) = [Q/\text{non empty}] \text{Prob}[Q \text{ non empty}] + [Q/\text{empty}] \text{Prob}[Q \text{ empty}]$$

Since $[Q/\text{empty}] = 0$, we have :

$$[Q/\text{non empty}] = \frac{N_f \rho}{(1 - \rho)(1 - (1 - \rho)^{N_f})}$$

Normalizing by the probability of being in the above threshold state we get the average number in the fast processors when the system is in the above threshold state.

We now move on to examine the slow processors cluster. The arrivals to the slow processors are processed locally only when the fast processors cluster is at or over threshold. Since there is no load sharing among the slow processors clusters, we can model a single slow queue and obtain the average delay for the whole cluster.

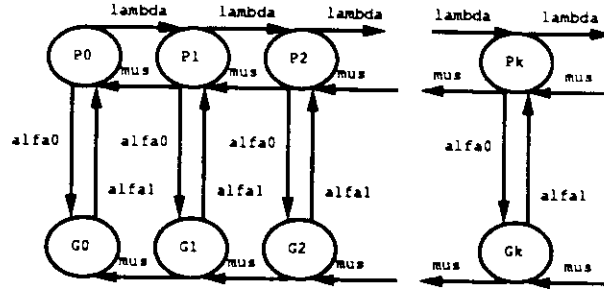


Figure 2.9: Markov chain for the slow processor cluster

The approach follows closely [ZR86]. The arrival rate to each slow processor when the fast processors' cluster is on the threshold state or in the overthreshold state is λ . When the gate for the fast processors' cluster is open, the arrival rate to the slow processors' cluster is zero. The state of the Markov chain for a slow processor is given by the pair: number in system, state of the gate for the fast processor cluster. The Markov chain is shown in figure 2.9.

We assume that α_0 (the rate that transitions are made from the saturated state to the unsaturated state in the fast processors' cluster), and α_1 (the rate back from the unsaturated state to the saturated state) are exponential rates in order to obtain an upper bound on the performance of the slow processors cluster. α_0 , and α_1 are obtained from the solution of the fast processors' cluster, as follows:

$psat$ is the probability that the fast processors' cluster is at or over threshold.

By aggregating all the saturated and unsaturated states and solving the balance equation we get:

$$psat = \frac{\alpha_1}{\alpha_0 + \alpha_1}$$

This result could also be obtained also by using renewal theory arguments. Using a first passage times approach we get, α_0 :

$$\begin{aligned}\frac{1}{\alpha_0} &= \frac{1}{N_f(\lambda + \mu)} + \frac{N_f\lambda}{N_f(\lambda + \mu)}\left(T + \frac{1}{\alpha_0}\right) \\ \alpha_0 &= \frac{N_f\mu}{1 + \lambda N_f T}\end{aligned}$$

The balance equations for the slow processor Markov chain are as follows:

For the set of states when the gate for the fast processors' cluster is closed (all arrivals to the slow processor are processed locally),

$$\begin{aligned}(\lambda + \alpha_0)p_0 &= \mu_s p_1 + \alpha_1 g_0, \quad k = 0 \\ (\lambda + \alpha_0 + \mu_s)p_k &= \mu_s p_{k+1} + \alpha_1 g_k + \lambda p_{k-1}, \quad k > 0\end{aligned}$$

For the set of states when the gate for the fast processors' cluster is open (no arrivals into the slow processor),

$$\begin{aligned}\alpha_1 g_0 &= \mu_s g_1 + \alpha_0 p_0, \quad k = 0 \\ (\alpha_1 + \mu_s)g_k &= \mu_s g_{k+1} + \alpha_0 p_k, \quad k > 0\end{aligned}$$

Following the approach in [ZR86] we define:

$$\Pi_0(Z) = \sum_{k=0}^{\infty} p_k z^k$$

$$\begin{aligned}
\Pi_1(Z) &= \sum_{k=0}^{\infty} g_k z^k \\
\Pi_0(Z) &= \frac{N_0(Z)}{D(Z)} \\
\Pi_1(Z) &= \frac{N_1(Z)}{D(Z)} \\
\Pi(Z) &= \Pi_0(Z) + \Pi_1(Z)
\end{aligned}$$

From [ZR86] we know:

$$\begin{aligned}
D(Z) &= -(\mu_s \lambda + \lambda \alpha_1) Z^2 + \mu_s (\lambda + \alpha_0 + \alpha_1 + \mu) Z - \mu_s^2 \\
A_0(Z) &= Z \alpha_0 - (1 - Z)(\mu_s - Z \lambda) \\
A_1(Z) &= Z \alpha_1 - (1 - Z) \mu_s \\
B_0 &= \mu_s p_0 \\
B_1 &= \mu_s g_0 \\
N_0(Z) &= A_1(Z) B_0(Z) + Z B_1(Z) \alpha_1 \\
N_1(Z) &= A_0(Z) B_1(Z) + Z B_0(Z) \alpha_0
\end{aligned}$$

which gives,

$$\begin{aligned}
N_0(Z) &= \mu_s p_0 [Z \alpha_1 - (1 - Z) \mu_s] + \mu_s g_0 Z \alpha_1 \\
N_1(Z) &= \mu_s g_0 [Z \alpha_0 - (1 - Z)(\mu_s - Z \lambda)] + \mu_s p_0 Z \alpha_0
\end{aligned}$$

Let Q^s be the average queue length in the slow processors' cluster.

$$\begin{aligned}
Q^s &= \Pi'(1) = \left(\frac{N1}{D1}\right)' \\
N'(1) &= (\mu_s g_0 + \mu_s p_0)(\alpha_0 + \alpha_1 + \mu_s) - \lambda \mu_s g_0 \\
D'(1) &= -2\lambda \alpha_1 + \mu_s(\alpha_0 + \alpha_1) + \mu_s^2 - \mu_s \lambda \\
D(1) &= -\lambda \alpha_1 + \mu_s(\alpha_0 + \alpha_1) \\
N(1) &= (\mu_s g_0 + \mu_s p_0)(\alpha_0 + \alpha_1)
\end{aligned}$$

After some algebra we get,

$$Q^s = \frac{\mu_s(\alpha_0 + \alpha_1) - \lambda \alpha_0}{(\alpha_0 + \alpha_1)^2} - \frac{\mu_s^2 - \lambda(\mu_s + \alpha_1)}{\mu_s(\alpha_0 + \alpha_1) - \lambda \alpha_1}$$

Using the same approach as in Section 2.4 , we can compute the upper bound on the system average delay since we know the average arrival rate to the slow and fast processors clusters and the average number of jobs in each cluster.

In order to compute the lower bound, we use a similar approach as the upper bound computation. Here, instead of having a above threshold state, we simple discard arrivals to the fast processors that find the system at threshold. For the slow processor cluster, we use a Poisson model in order to have a lower bound on the performance. This is necessary because the rates at which transitions are made between the saturated and unsaturated states are not exponential. The exponential rates used in the computation of the upper bound provide us with an upper bound on the performance of the slow processors' cluster, since in the real system we have a sum of exponential rates, which has a variation less than one. For the lower bound, a single slow processor queue will then be modeled as

a birth-death Markov Chain with arrival rate of λp_{sat} and service rate of μ_s . We can solve for Q^s , the single slow processor average queue length, analytically:

$$Q^s = \frac{\rho_s}{1 - \rho_s}$$

with $\rho_s = \lambda p_{sat} / \mu_s$.

Theorem 1 *The procedures described in this section compute upper and lower bounds on the average delay.*

Proof: The upper bound procedure models the slow jobs without discarding any jobs and processing locally the jobs which arrive when the fast processors are at or above threshold. The probability that the system is over threshold is overestimated, since in the real system, the gate for the fast processors cluster will be opened as soon as one fast processor goes below the threshold (while, in the upperbound model, the gate opens only when **all** fast processors are at or below threshold.) Also, in the over threshold state, we ‘freeze’ fast processors at the threshold; this clearly increases the over threshold interval. Thus, the average delay of slow processor jobs is overestimated. The average delay of the fast processors job is also overestimated, since all jobs that were in the fast processors cluster, when the system enters the over threshold state, are frozen.

The lower bound procedure underestimates the probability that the system is above threshold, thus the delay of slow processor jobs is underestimated. Since all jobs that arrive to the fast processors cluster to find the system at or above threshold are discarded, the delay of the fast processors job is underestimated.

2.5.1 Bounds Example

Arrival Rate	Upper Bound	Lower Bound	Simulation \pm 95% cf
0.2	0.261904	0.261904	0.267 ± 0.0001
0.6	0.282654	0.282654	0.288 ± 0.0001
0.8	0.293119	0.293061	0.298 ± 0.0001
1.0	0.420428	0.312424	0.320 ± 0.0002
1.1	NA	0.331483	0.36 ± 0.007
1.2	NA	0.357993	0.50 ± 0.04
1.5	NA	0.435952	NA

Table 2.1: Bounds on Average Delay for the 60 Host Example

A. R.	run 1	run 2	run 3	run 4	run 5
0.2	0.267731	0.267317	0.267541	0.267502	0.26744
0.6	0.288350	0.288009	0.28147	0.288049	0.287824
0.8	0.298457	0.298221	0.298468	0.298399	0.298203
1.0	0.320281	0.320111	0.320027	0.320162	0.319479
1.1	0.362110	0.359434	0.376739	0.364300	0.355754
1.2	0.561083	0.476405	0.478484	0.506030	0.455610
seed	2027350275	648473574	1396717869	377003613	1752629996

Table 2.2: Simulation Results for 60 Hosts Example; Average Delay vs. Arrival Rate

Table 2.1 shows the results obtained for upper and lower bounds for a system with 20 fast processors (4mips RT) and 40 slow processors (0.4 mips HP). The table also shows the simulation results. The simulation was run with 5 replications, each run for 800,000 local jobs. The transient phase was estimated to be of 100,000 jobs. The % 95 confidence interval was computed using the method of independent replications [Lav83]. The bounds are tight in the working range (0.2-0.8) agreeing on the first 3 digits. At 0.8 arrival rate there is already a substantial load balancing activity. In fact, without load balancing, we would have all slow processors with utilization of 2.0 in a M/M/1 system ($\mu_s = 0.4$). We observe that for an arrival rate of 1.0-1.1 the lower bound gives a very good performance prediction. The upper bound diverges from the lower bound for higher loads(1.0-) but it can still be used to determine the knee of the curve. Table 2.2 shows the simulation results and the seed for the 60 hosts example. The average and the confidence interval obtained are shown in table 2.1.

2.6 Tighter Upper Bound

The bounding approach described in section 2.5 works very well from small to medium loads, since most of the probability mass in the Markov Chain resides in the under threshold region. At high loads, however, the busy period generated by the overthreshold state (with all jobs on the threshold 'frozen'), caused the upper bound to diverge. There, we considered that the first arrival to a system in threshold state 'froze' the processing of all fast processor jobs found in the system. That arrival generated a busy period for the fast processors cluster, where all arrivals were processed locally.

In this section we develop a tighter upper bound on the average delay. We compute T (the average time to go from the overthreshold state to the threshold

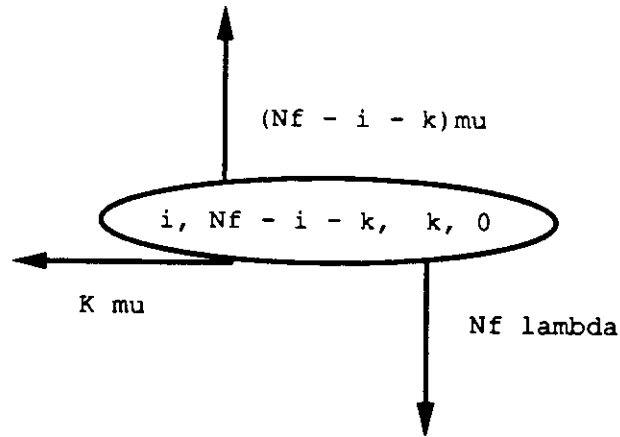


Figure 2.10: Transitions in Region A

state) by building a Markov Chain that models the fast processors cluster behavior in the overthreshold state and by using a first passage time computation approach [HS82]. We then use the same approach as in section 2.5 as described in figure 2.8, that is, we still consider all jobs found in the fast processor cluster to be frozen.

The state of the system in the overthreshold state is $S = (N1, N2, N3, N4)$ where,

- $N1$ = number of fast processors with exactly one job,
- $N2$ = number of fast processors with exactly two jobs,
- $N3$ = number of fast processors with exactly three jobs,
- $N4$ = number of jobs that arrived to a three job processor.

In order to keep the number of states in the Markov Chain bounded we divided the Markov Chain in 4 regions:

- Region A: At least one processor has only one job. We assume that the one

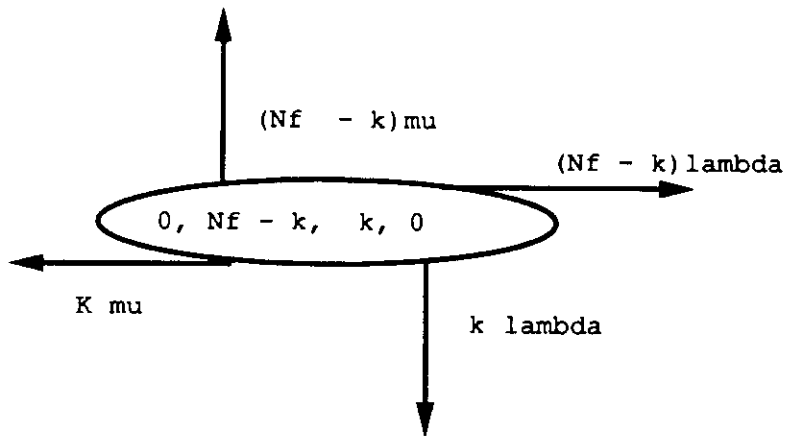


Figure 2.11: Transitions in Region B

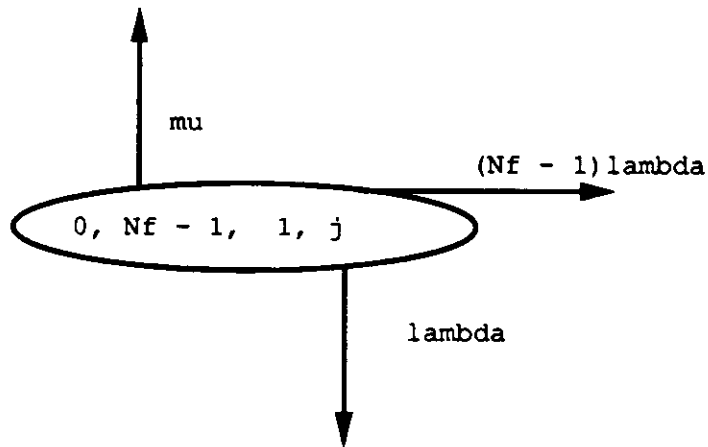


Figure 2.12: Transitions in Region C

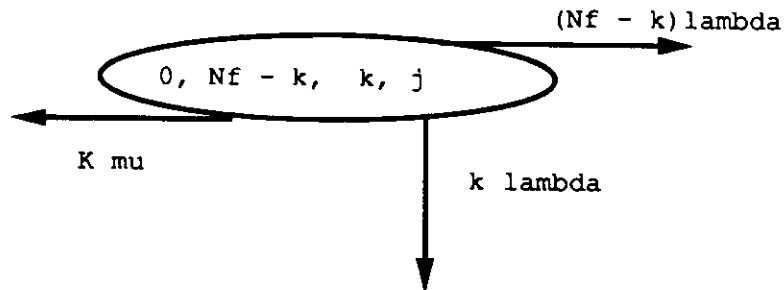


Figure 2.13: Transitions in Region D

job processors are frozen and that the load sharing algorithm works only for the fast processors. The transitions in region A are described in figure 2.10.

- Region B: All processors have 2 or 3 jobs. No jobs are frozen in this state. The transitions in region B are described in figure 2.11.
- Region C: In this region we have $N_f - 1$ processors with 2 jobs, 1 processor with 3 and zero processors with 1 job. We consider that all jobs that arrived to 3 processor jobs are stacked in the same processor (worst case) and are thus processed serially. The 2 job processors are frozen. The transitions in region C are described in figure 2.12.
- Region D: In this region we have at least one job that arrived to a three job processor . Those jobs and the jobs in the 2 job processors are frozen. In this region we process only jobs in the 3 job processors. The transitions in region D are described in figure 2.13.

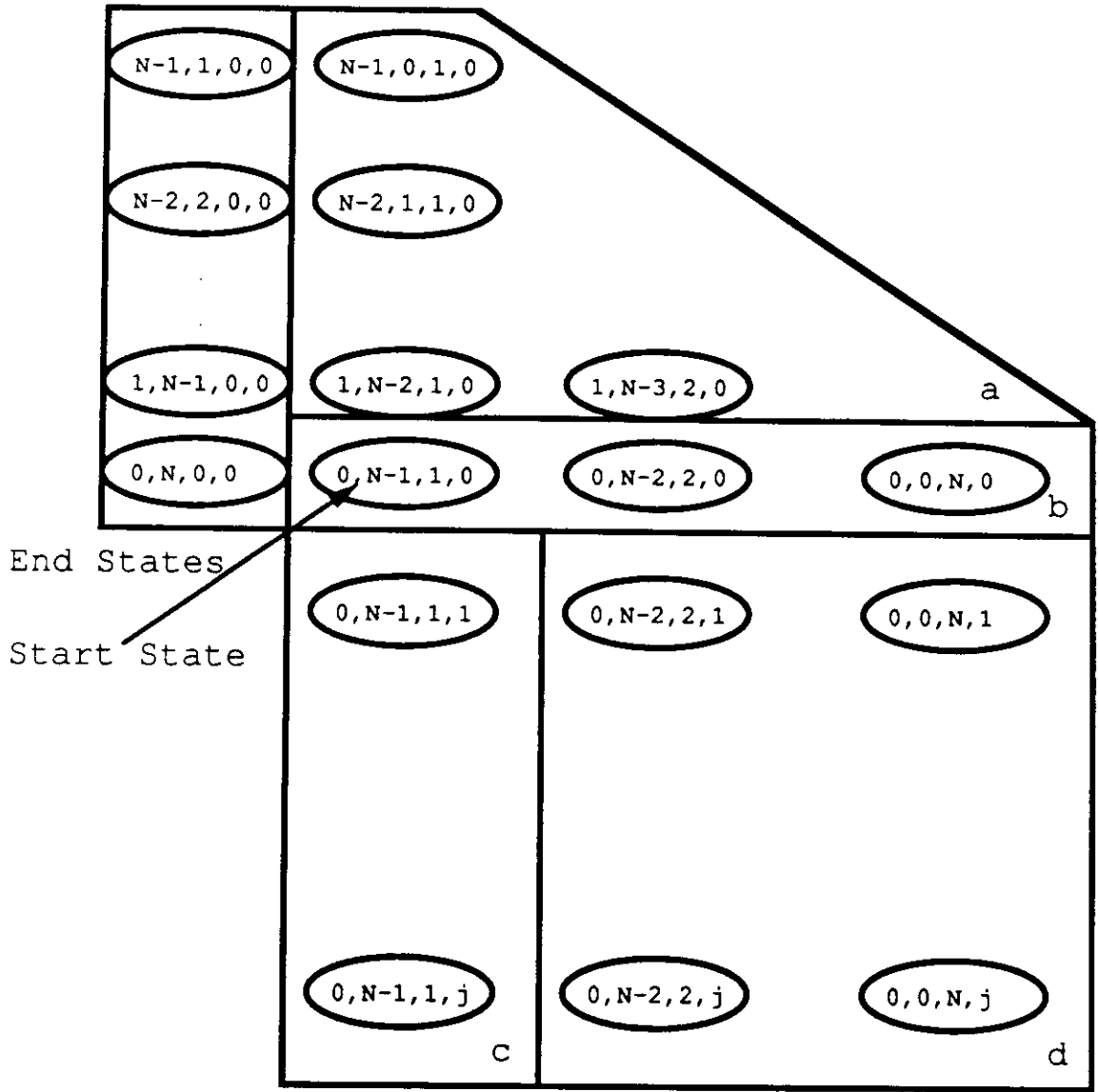


Figure 2.14: Relation among regions

Figure 2.14 shows the full Markov Chain state space and its division in the regions A,B,C and D. The transitions are described in figures 2.10, 2.11, 2.12 and 2.13.

In order to compute T (the average time to go from the overthreshold state to the threshold state) we write down a linear set of equations for $T(S)$ (the first passage time from a general state S to any state were the gate is open.)

Region A:

$$\begin{aligned}
T(i, N_f - i - k, k, 0) = & (N_f \lambda + (N_f - i) \mu)^{-1} + \\
& \frac{N_f \lambda}{N_f \lambda + (N_f - i) \mu} T(i - 1, N_f - i + 1 - k, k, 0) + \\
& \frac{k \mu}{N_f \lambda + (N_f - i) \mu} T(i, N_f - i + 1 - k, k - 1, 0) + \\
& \frac{(N_f - i - k) \mu}{N_f \lambda + (N_f - i) \mu} T(i + 1, N_f - i - k - 1, k, 0)
\end{aligned}$$

Region B:

$$\begin{aligned}
T(0, N_f - k, k, 0) = & (N_f \mu + N_f \lambda)^{-1} + \frac{(N_f - k) \mu}{N_f \mu + N_f \lambda} T(1, N_f - k - 1, k, 0) + \\
& \frac{(N_f - k) \lambda}{N_f \mu + N_f \lambda} T(0, N_f - k - 1, k + 1, 0) + \\
& \frac{k \mu}{N_f \mu + N_f \lambda} T(0, N_f - k + 1, k - 1, 0) + \\
& \frac{k \lambda}{N_f \mu + N_f \lambda} T(0, N_f - k, k, 1)
\end{aligned}$$

Region C:

$$\begin{aligned}
T(0, N_f - 1, 1, j) = & (\mu + N_f \lambda)^{-1} + \frac{\mu}{\mu + N_f \lambda} T(0, N_f - 1, 1, j - 1) + \\
& \frac{(N_f - 1) \lambda}{\mu + N_f \lambda} T(0, N_f - 2, 2, j) + \\
& \frac{\lambda}{\mu + N_f \lambda} T(0, N_f - 1, 1, j + 1)
\end{aligned}$$

Region D:

$$T(0, N_f - k, k, j) = (k \mu + N_f \lambda)^{-1} + \frac{k \mu}{k \mu + N_f \lambda} T(0, N_f - k + 1, k - 1, j) +$$

$$\frac{(N_f - k)\lambda}{k\mu + N_f\lambda} T(0, N_f - k - 1, k + 1, j) +$$

$$\frac{k\lambda}{k\mu + N_f\lambda} T(0, N_f - k, k, j + 1)$$

This is an infinite sparse linear set of equations. In order to solve it numerically we truncate Regions C and D for values of j that have very small probability of occurring. In the example considered in Section 2.6.1 for values of j greater than 20 we already have very small probabilities (less than $10e^{-8}$). The solution for $T(0, N - 1, 1, 0)$ was obtained by iteration. The number of steps needed for convergence is of the order of the number of states in the Markov Chain [PFTV86]

The average number of fast processor jobs given that the system is in above threshold state can be computed as in Section 2.5.

Theorem 2 *The procedure described in this section compute upper bound on the average delay.*

Proof:

The probability that the system is over threshold is overestimated, since in the real system, the gate for the fast processors cluster will be opened as soon as one fast processor goes below the threshold (while, in the upperbound model, the gate opens only when **all** fast processors are at or below threshold.) Also, in the over threshold state, we ‘freeze’ two job processors in region C and D and we ‘freeze’ one job processors in region A. This clearly increases the over threshold interval. Thus, the average delay of slow processor jobs is overestimated. The average delay of the fast processors job is overestimated, since all jobs that were in the fast processors cluster, when the system enters the over threshold state, are frozen.

2.6.1 Bounds Example

Arrival Rate	Upper Bound	Lower Bound	Simulation \pm 95 % cf
0.2	0.261904	0.261904	0.267 ± 0.0001
0.6	0.282654	0.282654	0.288 ± 0.0001
0.8	0.293119	0.293061	0.298 ± 0.0001
1.0	0.351617	0.312424	0.320 ± 0.0002
1.1	0.697628	0.331483	0.36 ± 0.007
1.2	5.25	0.357993	0.50 ± 0.04
1.5	NA	0.435952	NA

Table 2.3: Bounds on Average Delay for the 60 Host Example

Table 2.3 shows the results obtained for the upper bound for a system with 20 fast processors (4mips RT) and 40 slow processors (0.4 mips HP). The table also shows the simulation results. The simulation was run with 5 replications, each run for 800,000 local jobs. The transient phase was estimated to be of 100,000 jobs. The 95 % confidence interval was computed using the method of independent replications [Lav83]. The bounds are tight in the whole working range diverging only, when the system cannot sustain the offered load any more.

2.7 Generalizations

In the previous Section we described the modeling of the dynamic load sharing algorithm, when the thresholds were fixed and equal to zero for the slow processors and equal to two for the fast processors. In this Section we analyze the case where the thresholds for the fast processors' cluster can assume a higher value, but is still identical to some integer k , for all fast processors. In this case we can aggregate the states of the Markov Chain in the same way as before. What is different now, is the Markov Chain for the under threshold states for the fast processors' cluster. When the thresholds were equal to two we had a two-dimension Markov Chain with vertical and horizontal transitions only. When the thresholds can assume higher values for each step in the thresholds value a new dimension is added to the Markov Chain.

The state of the Markov chain when the thresholds are equal to k is:

$$S = (S_1, S_2, \dots, S_i, \dots, S_k),$$

where S_i is the number of processors with i jobs. Using the same approach as before we can obtain an upper bound on the performance of the system. The problem here is the combinatorial explosion in the number of states of the Markov chain. Let,

n – number of processors

k – threshold for the fast processors' cluster

Then the number of states in underthreshold the Markov chain is given by:

$$\begin{pmatrix} n + k \\ k \end{pmatrix}$$

Currently it is possible to solve Markov chains with about tens of thousands of states. In order to give a flavor on which size of examples may be solvable using state of the art/1990 computers, table 2.4 shows the number of states of the Markov chain as a function of the number processors and the threshold for the fast processors' cluster.

2.8 Decomposition by Threshold Configuration Approach: Bursty Arrivals

In systems with fluctuating parameters, quasi-static thresholds can be used to dynamically match the load sharing algorithm to the system parameters. In order to solve for the response time of such systems we apply the well know decomposition/ aggregation approach [Cou77], [CS84], [CS86], [CG89].

Let Q be an irreducible infinitesimal generator matrix of order n associated with the homogeneous, continuous time Markov Chain with global state-space S that describes the complete load sharing problem. Let us assume that Q can be partitioned into submatrices $Q(\alpha, \beta)$ such that the states represented in the diagonal square matrix $Q(\alpha, \alpha)$ are strongly coupled (i.e. transitions within $Q(\alpha, \alpha)$ are much more frequent than across them.) Each $Q(\alpha, \alpha)$ represents the infinitesimal

Number of Processors	Threshold	Number of States
10	2	66
10	5	3,003
10	7	19,448
10	8	43,758
20	2	231
20	4	10,626
20	5	53,130
30	2	496
30	3	5,456
30	4	46,376
50	2	1,326
50	3	23,426
70	2	2,556
70	3	62,196

Table 2.4: Number of States for the Underthreshold Markov chain

generator for a given threshold configuration as shown in section 2.5. Let $n(\alpha)$ be the order of the the square matrix $\mathbf{Q}(\alpha, \alpha)$. Let $s_{i\alpha}$ be the off diagonal row sum associated with the i row of the square matrix $\mathbf{Q}(\alpha, \alpha)$. Also, let's assume that \mathbf{Q} is normalized, so that the maximum value of any external row sum is one. Let $\mathbf{Q} = \mathbf{Q}^* + \epsilon\mathbf{C}$ be a decomposition where $\mathbf{Q}^* = \text{diag}(\mathbf{Q}_1^*, \dots, \mathbf{Q}_M^*)$, \mathbf{Q}_α^* is an infinitesimal generator matrix constructed from $\mathbf{Q}(\alpha, \alpha)$ by distributing the quantity $s_{i\alpha}$ over the non-zero elements of the i th row of $\mathbf{Q}(\alpha, \alpha)$, for all rows of $\mathbf{Q}(\alpha, \alpha)$. Let $\epsilon = \max_i s_{i\alpha}$. ϵ is called the maximum degree of coupling between the $\mathbf{Q}(\alpha, \alpha)$.

The error introduced by the decomposition is known to be $O(\epsilon)$. The proportionality constant is however dependent on the way the distribution of the off diagonal row sum is chosen [Cou82].

We now assume that the arrival process is bursty, i.e., it fluctuates between two arrival states: State A where the arrival rate is λ_1 , and state B where the arrival rate is λ_2 . Transitions between these modes are exponentially distributed with rate α_1 from state A to state B , and rate α_2 from state B to state A .

Clearly \mathbf{Q} is given by:

$$\begin{pmatrix} & & & \alpha_1 & & & & & \\ & & & & \alpha_1 & & & & \\ & & Q(1,1) & & & \alpha_1 & & & \\ & & & & & & \dots & & \\ & & & & & & & & \alpha_1 \\ \alpha_2 & & & & & & & & \\ & \alpha_2 & & & & & & & \\ & & \alpha_2 & & Q(2,2) & & & & \\ & & & \dots & & & & & \\ & & & & & & & & \alpha_2 \end{pmatrix}$$

Where $Q(1,1)$ and $Q(2,2)$ are computed as in Section 2.5.

Application of the Simon and Ando Decomposition and Aggregation :

1. **Decomposition** Construct Q^* from Q , and find v^* , where $v^*(Q^* + I) = v^*$ and $v^* = (v_1^*, v_2^*)$, and $v_\alpha^* = (v_{1\alpha}^*, \dots, v_{n(\alpha)\alpha}^*)$, subject to the constraint that $v_\alpha^* \mathbf{1}^T = 1$, for $1 \leq \alpha \leq 2$.
2. **Aggregation** Construct the infinitesimal generator A , of order M , where $a_{i,j}$ is the (i, j) element of A , let

$$a_{ij} = \begin{cases} \mathbf{v}_i^* \mathbf{Q}(i,j) \mathbf{1}^T, & \text{for } i \neq j, \\ -\sum_{k=1, k \neq i}^2 a_{ik}, & \text{for } i = j. \end{cases}$$

Clearly,

$$\mathbf{v}_1^* \mathbf{Q}(1,2) \mathbf{1}^T = \alpha_1$$

$$\mathbf{v}_2^* \mathbf{Q}(2,1) \mathbf{1}^T = \alpha_2$$

and the infinitesimal generator \mathbf{A} reduces to:

$$\begin{pmatrix} -\alpha_1 & \alpha_1 \\ \alpha_2 & -\alpha_2 \end{pmatrix}$$

Find the equilibrium distribution \mathbf{u} of the reduced system \mathbf{A} , by solving $\mathbf{u}(\mathbf{A} + \mathbf{I}) = \mathbf{u}$, with $\mathbf{u} \mathbf{1}^T = 1$. This step is straight forward, since in our case the \mathbf{v}_α^* factored out.

3. *Unconditioning* Use the vector \mathbf{u} in order to uncondition \mathbf{v}^* . We do that by computing the desired approximation for $\pi = [[u_1 \mathbf{v}_1^*], [u_2 \mathbf{v}_2^*]]$.

In order to compute the distribution of the external row sum into the diagonal block matrixes we proceed as follows. Without loss of generality we can consider the case of stochastic matrixes only [MdSeSG89]. Let \mathbf{P} be a finite, non-negative, normalized, stochastic matrix. Let \mathbf{P} be:

$$\begin{pmatrix} Q_1 & E \\ F & G \end{pmatrix}$$

The above form is considered in order to simplify notation. The approach is from [CS84]. Let \mathbf{Q}_1^* be constructed by the distribution of the row sums of \mathbf{E} into the non-zero rows of \mathbf{Q}_1 . In [Cou82] Courtois showed that if $\mathbf{Q}_1^* = \mathbf{Q}_1 + \mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}$, then the decomposition approach yields exact results.

In our case \mathbf{E} :

$$\begin{pmatrix} \alpha_1 & & & & \\ & \alpha_1 & & & \\ & & \alpha_1 & & \\ & & & \ddots & \\ & & & & \alpha_1 \end{pmatrix}$$

And \mathbf{F} :

$$\begin{pmatrix} \alpha_2 & & & & \\ & \alpha_2 & & & \\ & & \alpha_2 & & \\ & & & \ddots & \\ & & & & \alpha_2 \end{pmatrix}$$

Usually, the computation of $\mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}$ is prohibitive, since it involves the inversion of a matrix that have only one aggregate less than the full system. In our case, we can compute the decomposition exactly by inverting $\mathbf{I} - \mathbf{G}$, $\mathbf{G} = \mathbf{Q}(2, 2)$.

If \mathbf{E} and \mathbf{F} are fully known tight bounds on $\mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}$ can be obtained as follows [CS84]:

$$[\mathbf{EF}]_{ij} \leq [\mathbf{E}(\mathbf{I} - \mathbf{G})^{-1}\mathbf{F}]_{ij} \leq [\mathbf{E}\mathbf{1}]_i - \sum_{k \neq j} [\mathbf{EF}]_{ik}$$

In [CS84] looser bounds are given in the case where \mathbf{F} is not fully known.

It is interesting to point out that the analysis of systems with bursty departures (i.e. the service rate can assume many different states) can be done by following the same procedure as in this section. There, one model must be computed for each departure rate configuration.

2.9 Conclusions and Future Research

We presented a new approach for the modeling of dynamic load sharing algorithms in asymmetric systems that employs a global state Markov Chain. The model was validated through simulation and a very good agreement was found at low to medium loads. We managed to analyze large systems as well, by bounding the performance measure (long run average delay.) The lower bound obtained is tight (in the range of arrival rates between 0.2-1.0), since the states discarded have very low probabilities. The upper bound proposed is tight as well in the same range.

We are continuing and extending these studies in the following directions:

- *Optimal Job Mix*

The optimal way to mix different classes of jobs(e.g., Short and long cpu intensive and Interactive) in order to minimize average delay. This question involves the modeling of multiple class load sharing asymmetric systems and the problem of workload characterization since we usually do not know what are the job demands for resources a priori.

- *Configuration Design*

The configuration design of such systems under load sharing control requires an efficient performance modeling algorithm. We intend to study the suitability of our performance bounds to the configuration design of such systems.

Chapter 3

Sensitivity Analysis

“No mais, mesmo, da mesmice, sempre vem a novidade...”

So, really, the obviousness still brings novelty.

João Guimarães Rosa In: Primeiras Estórias, pp 106

When comparing representative algorithms for load sharing under bursty arrivals, the performance measure sensitivity to the variation of the following parameters must be evaluated:

- Number of hosts in the system,
- Burstiness of the arrival process,
- Speed ratio between fast and slow processors,
- Sensitivity to the system model (dynamic tuning only,)
- Sensitivity to the optimization algorithm (dynamic tuning only,)
- Sensitivity to the threshold computation period (dynamic tuning only.)

In this Chapter we present the evaluation of seven algorithms for load sharing in distributed systems according to the criteria described above. The sensitivity to the optimization algorithm is left as a topic for future research.

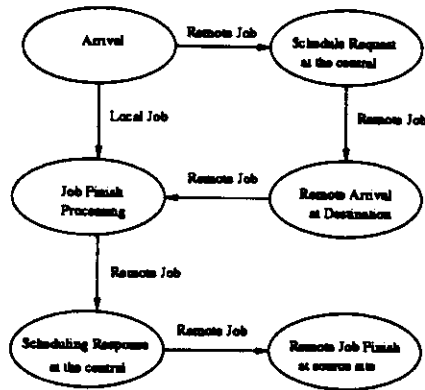


Figure 3.1: Simulation Model

3.1 The Model

The simulation model considered was the following:

1. The system under study is composed of fast processors (40 mips) and slow processors (4 mips,)
2. Jobs arrive at the local sites according to a bursty Poisson arrival process. The arrival process is composed of two phases: an active phase where arrivals are accepted and a passive phase where arrivals are discarded. The length of the active phase (with burst length average) and of the passive phase (with idle length average) are exponentially distributed. The **burstiness** of the process is defined to be the ratio: $(\text{burstlength} + \text{idle length})/\text{burstlength}$,
3. The local scheduling algorithm is employed and the job is classified as local or remote,
4. Local jobs are queued in the local cpu according to FCFS scheduling and are run for an exponentially distributed number of instructions with 1M instructions average,

5. Remote jobs suffers 12.5 ms (processing overhead) and 5 ms communication delay to get to central site. There, they queue for the central cpu and use 100k instructions processing overhead and 5 ms communication delay to be scheduled for remote execution,
6. Upon arrival at the remote site, they are queued in the remote cpu according to FCFS scheduling (one queue for local and remote jobs) and are run for an exponentially distributed number of instructions with 1M instructions average,
7. Once a local job completes its running time it is done. A remote job suffers another 12.5ms (processing overhead) and 5 ms communication delay to get to the central site, and must queue again for the central cpu to consume another 100k instructions processing overhead and 5 ms communication delay, before being sent to its originating site for completion.

3.2 Representative Algorithms

We will start by presenting the algorithms we have chosen to perform the sensitivity analysis. Those algorithms use different amounts of information in order to perform load sharing. In this work we assume that we have a workstation (e.g. 40 mips IBM PC-RT) dedicated to job routing in the system. The different algorithms presented represent the information that is used by the router in order to perform global scheduling, and the conditions for a remote request to be issued by a local site.

3.2.1 Dynamic Tuning Algorithm

This is the algorithm that was presented in Chapter 2. It is a threshold type algorithm that varies the thresholds dynamically adjusting them to the load in order to keep an optimal number of tasks in each site. The algorithm periodically gathers information on the global load in the system and analytically computes an optimal task allocation in order to minimize the total average response time. This optimal allocation is a function of the total measured load. The site thresholds are set to be equal to the optimal task allocations. We will consider two versions of this algorithm. One where the computation of thresholds is done every second **dynamic(1)** and another where it is done every five seconds **dynamic(5)**.

3.2.2 Local Information Algorithm

This algorithm is a threshold based algorithm, where remote jobs are scheduled randomly. It was originally described in [LT86a]. We assigned a threshold of 2 to the fast processors and a threshold of 0 to the slow processors. We refer to it in the tables as **Random(LT)**.

3.2.3 Fixed thresholds algorithm

This is the best scheme found in [Zho87]. We will consider a version with thresholds identical to 1 for all sites, a version where the thresholds are set to 0 for the slow sites and 2 for the fast sites, and a version where the thresholds are set to 0 for the slow sites and set to 10 for the fast sites. We refer to them in the tables as **Fixed(1-1)**, **Fixed(0-2)**, and **Fixed(0-10)**, respectively.

3.2.4 Optimal static algorithm

This algorithm routes jobs according to pre-computed probabilities, which optimize the average delay statically. This approach is the one described in [FGK73]. We refer to it in the tables as **Static**.

3.3 Simulation Results

In this section we report simulation results obtained when the representative algorithms are employed. The performance measure used was the long range average delay. All simulation were run for 800,000 local jobs with a transient phase estimated at 100,000 jobs. The simulations were run with 5 independent replications, which were used in order to compute the 95 % confidence intervals, using the method of independent replications. Each simulation point took 10 cpu hours on a 4mips PC/RT to compute. In the following tables we report average delays and the 95 % confidence intervals.

3.3.1 Burstiness of the arrival process

The environment considered in this subsection was composed of 5 slow hosts (4 mips) and 5 fast hosts (40 mips). The fixed(1-1), random with local thresholds, and static algorithms diverged for the experiment reported on tables 3.3 and 3.4. Tables 3.1,3.2,3.3, 3.4, 3.5,3.6,3.7, 3.8, 3.9,and 3.10 report the results for the simulation with burstiness of 2 and 4 for the burstlength varying from 0.1-10.0. The main conclusions to be drawn from the tables are the following:

- The seven algorithms considered can be divided in two groups that exhibit major difference in performance, according to their performance at low burstiness and low utilizations (25 %). The first group is the one that main-

tains global information about queue length, schedules jobs to the queue with maximum difference between threshold and queue length, and has a threshold of 0 for the slow processors (table 3.1.) The second group include static load sharing, identical threshold of 1 for all sites with shortest queue allocation, and random allocation with local thresholds (table 3.2.) Figures 3.2, and 3.4 compare the performance of the algorithms in the first group with a representative algorithm of the second group **dynamic(5)**. From the figures we learn that in asymmetric systems under bursty arrivals great care must be used when selecting load sharing algorithms. We conclude that random allocation in asymmetric systems produces major degradation in performance as compared to shortest queue allocation.

- Burstiness and burstlength have a major impact on performance on all algorithms.
- The proper tuning of thresholds to the hosts capacities and arrival process is critical. The mismatch between host capacities and thresholds is responsible for a major degradation in the performance of the fixed(1-1) algorithm, as compared to the same algorithms with tuned thresholds.
- Dynamic tuning of thresholds exhibits the best performance over all algorithms analyzed for range of parameters. The fixed(0-10) exhibits a similar performance at low to medium loads and low burst lengths, since its thresholds are matched to the sites capacities. However, when the system is stressed to its limits (tables 3.4, 3.7, 3.8, and figure 3.3) we observe that the dynamic tuning is able to improve performance over the fixed(0-10) by factors of 2-2.6.

In figures 3.6, and 3.7 we plot the average delay as a function of the burstiness while maintaining the utilization constant. Those plots reinforce our previous

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.062 ± 0.0001	0.060 ± 0.0001	0.060 ± 0.0001	0.060 ± 0.0001
0.2	0.063 ± 0.0001	0.061 ± 0.0001	0.061 ± 0.0001	0.061 ± 0.0001
0.5	0.063 ± 0.0001	0.061 ± 0.0001	0.061 ± 0.0001	0.061 ± 0.0001
1.0	0.064 ± 0.0001	0.062 ± 0.0001	0.061 ± 0.0001	0.061 ± 0.0001
10.0	0.064 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001

Table 3.1: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 10

conclusions that dynamic tuning provides better performance at higher burstiness and higher loads.

The fixed(1-1), random with local thresholds, and static algorithms diverged for the experiment reported on tables 3.7 and 3.8.

3.3.2 Number of Hosts in the system

In this subsection we evaluate the effect of varying the number of hosts on the performance of the algorithms. We consider only the second group of algorithms **fixed(0-2)**, **dynamic(1)**, **dynamic(5)**, **fixed(0-10)**; since the first group is already degrading with 10 hosts. The environment considered in this subsection was composed of half slow hosts (4 mips) and half fast hosts (40 mips). The main objective here is to assess the following:

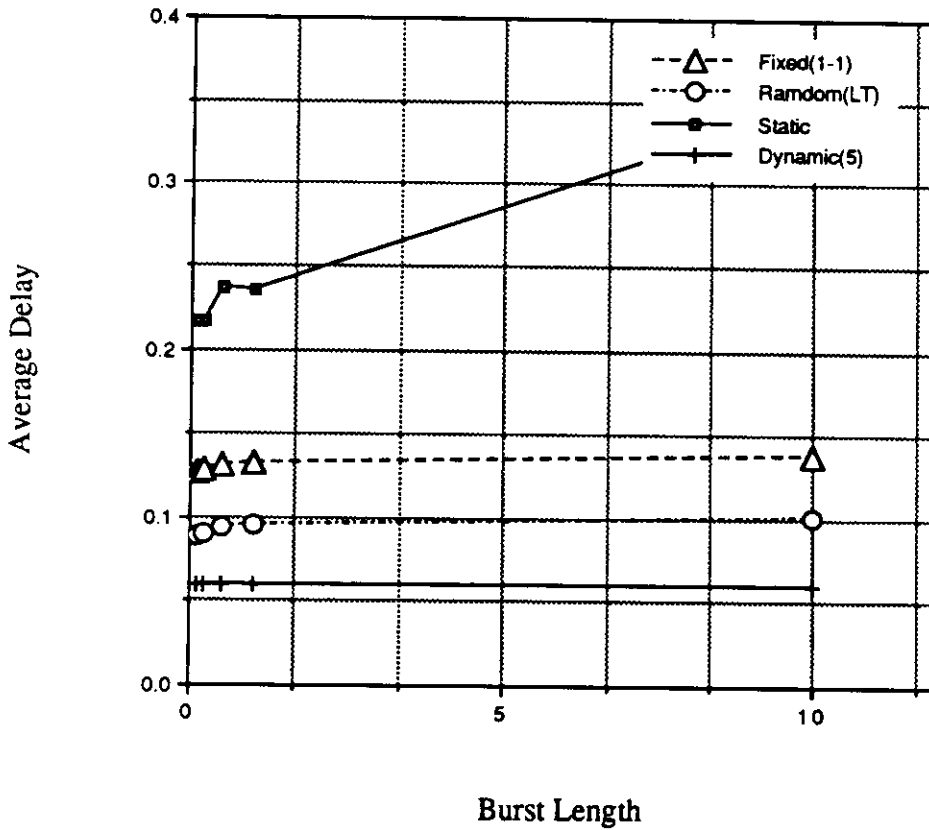


Figure 3.2: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 10

B. L.	Fixed(1-1)	Random(LT)	Static
0.1	0.128 ± 0.0001	0.089 ± 0.0001	0.217 ± 0.0001
0.2	0.129 ± 0.0001	0.091 ± 0.0001	0.217 ± 0.0001
0.5	0.131 ± 0.0001	0.094 ± 0.0001	0.236 ± 0.0001
1.0	0.133 ± 0.0001	0.096 ± 0.0001	0.235 ± 0.0001
10.0	0.138 ± 0.0001	0.101 ± 0.0001	0.348 ± 0.0001

Table 3.2: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 10

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.154 ± 0.0008	0.086 ± 0.0001	0.086 ± 0.0001	0.086 ± 0.0001
0.2	0.194 ± 0.0022	0.090 ± 0.0001	0.090 ± 0.0002	0.090 ± 0.0001
0.5	0.285 ± 0.0001	0.098 ± 0.0001	0.098 ± 0.0001	0.100 ± 0.0001
1.0	0.424 ± 0.0096	0.105 ± 0.0007	0.106 ± 0.0009	0.114 ± 0.0026
10.0	INF	0.211 ± 0.0869	0.219 ± 0.021	INF

Table 3.3: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 22; number of hosts = 10

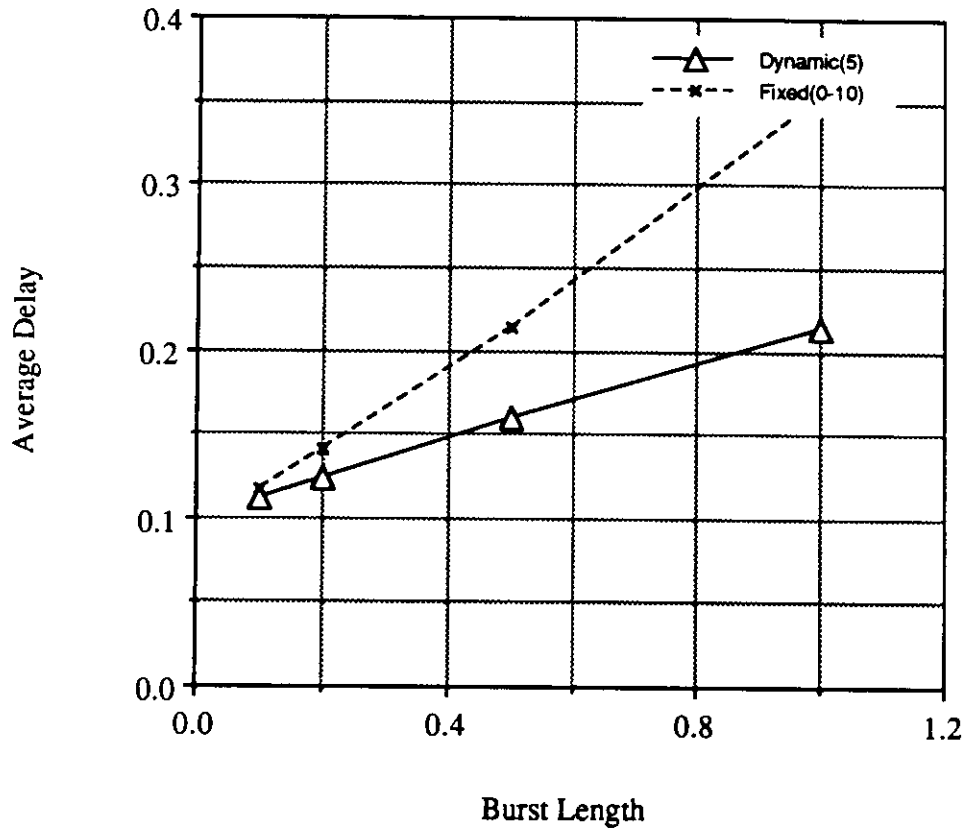


Figure 3.3: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 26; number of hosts = 10

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	INF	0.112 ± 0.0006	0.112 ± 0.0006	0.117 ± 0.0012
0.2	INF	0.124 ± 0.0008	0.124 ± 0.0006	0.141 ± 0.0021
0.5	INF	0.154 ± 0.0042	0.159 ± 0.0034	0.214 ± 0.0037
1.0	INF	0.189 ± 0.0040	0.214 ± 0.0021	0.353 ± 0.0155
10.0	INF	INF	INF	INF

Table 3.4: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 26; number of hosts = 10

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.071 ± 0.0004	0.066 ± 0.0001	0.066 ± 0.0001	0.066 ± 0.0001
0.2	0.075 ± 0.0001	0.069 ± 0.00005	0.069 ± 0.0001	0.069 ± 0.0001
0.5	0.080 ± 0.0001	0.072 ± 0.0002	0.072 ± 0.0002	0.072 ± 0.0002
1.0	0.084 ± 0.0002	0.073 ± 0.0001	0.073 ± 0.0002	0.073 ± 0.0002
10.0	0.128 ± 0.0001	0.076 ± 0.0034	0.075 ± 0.0030	0.077 ± 0.0026

Table 3.5: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 10

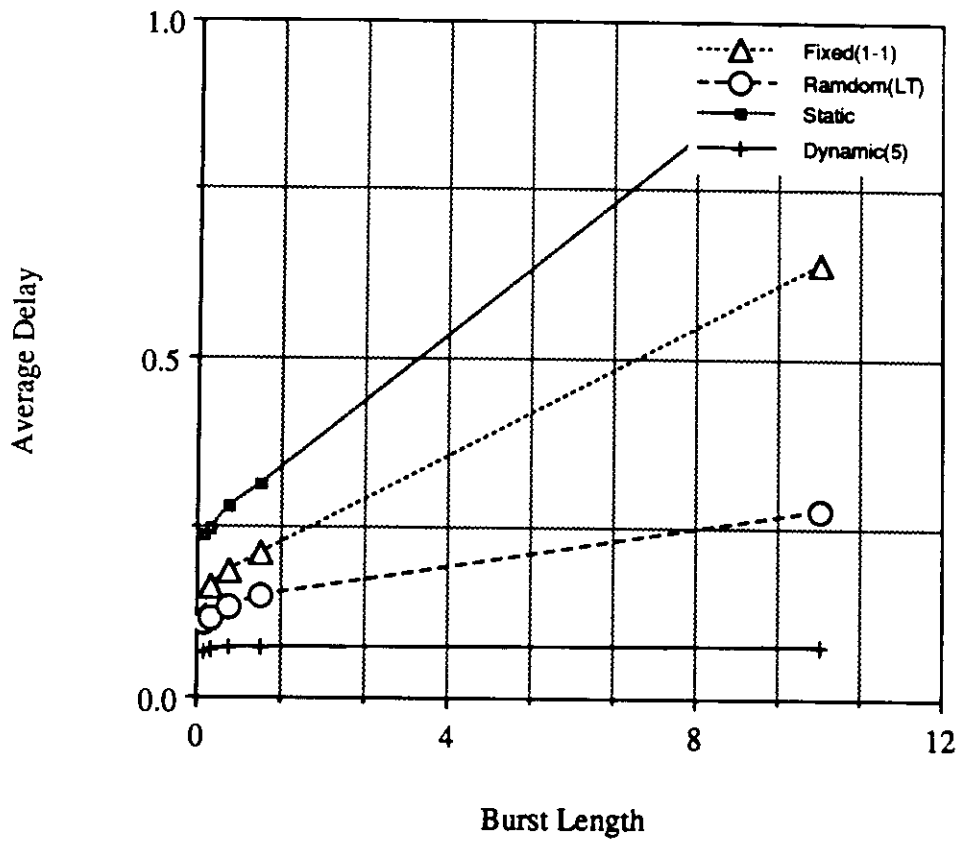


Figure 3.4: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 10

B. L.	Fixed(1-1)	Random(LT)	Static
0.1	0.115 ± 0.0008	0.107 ± 0.0007	0.236 ± 0.0104
0.2	0.160 ± 0.0003	0.116 ± 0.0004	0.247 ± 0.0061
0.5	0.182 ± 0.0008	0.132 ± 0.0007	0.278 ± 0.0186
1.0	0.211 ± 0.0016	0.146 ± 0.0008	0.312 ± 0.0110
10.0	0.638 ± 0.0416	0.276 ± 0.0268	0.975 ± 0.2415

Table 3.6: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 10

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.729 ± 0.0374	0.121 ± 0.0006	0.122 ± 0.0018	0.131 ± 0.0018
0.2	INF	0.156 ± 0.0014	0.161 ± 0.0021	0.202 ± 0.0018
0.5	INF	0.260 ± 0.0114	0.312 ± 0.0109	0.507 ± 0.0371
1.0	INF	0.493 ± 0.0796	INF	INF
10.0	INF	INF	INF	INF

Table 3.7: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 44; number of hosts = 10

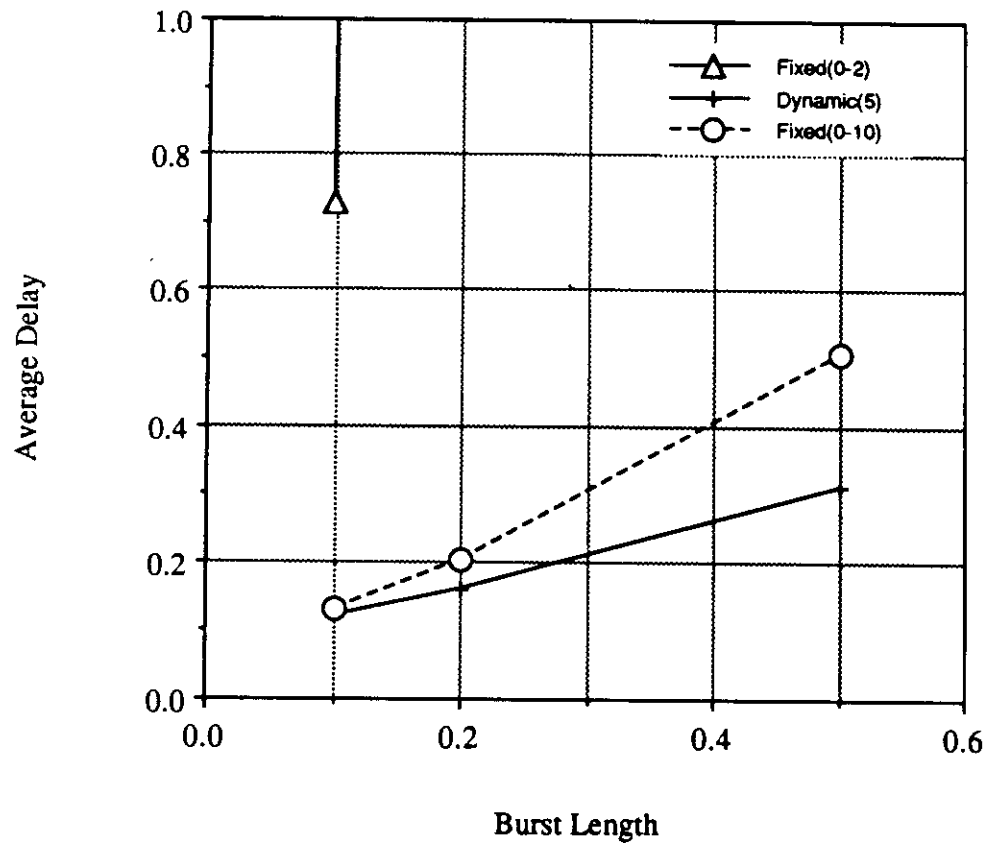


Figure 3.5: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 44;
number of hosts = 10

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	INF	0.192 ± 0.0025	0.205 ± 0.0039	0.331 ± 0.0062
0.2	INF	0.328 ± 0.0086	0.386 ± 0.0165	0.860 ± 0.0580
0.5	INF	INF	INF	INF

Table 3.8: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 52; number of hosts = 10

A. R.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
5.5	0.059 ± 0.0001	0.058 ± 0.00003	0.058 ± 0.00001	0.058 ± 0.00001
11.0	0.106 ± 0.0003	0.077 ± 0.0001	0.077 ± 0.0002	0.077 ± 0.0001

Table 3.9: Average delay vs. arrival rate ; number of hosts = 10 ; Poisson arrivals

A. R.	Fixed(1-1)	Random(LT)	Static
5.5	0.121 ± 0.0002	0.083 ± 0.0003	0.218 ± 0.0014
11.0	INF	INF	INF

Table 3.10: Average delay vs. arrival rate ; number of hosts = 10 ; Poisson arrivals

Burst Length = 10

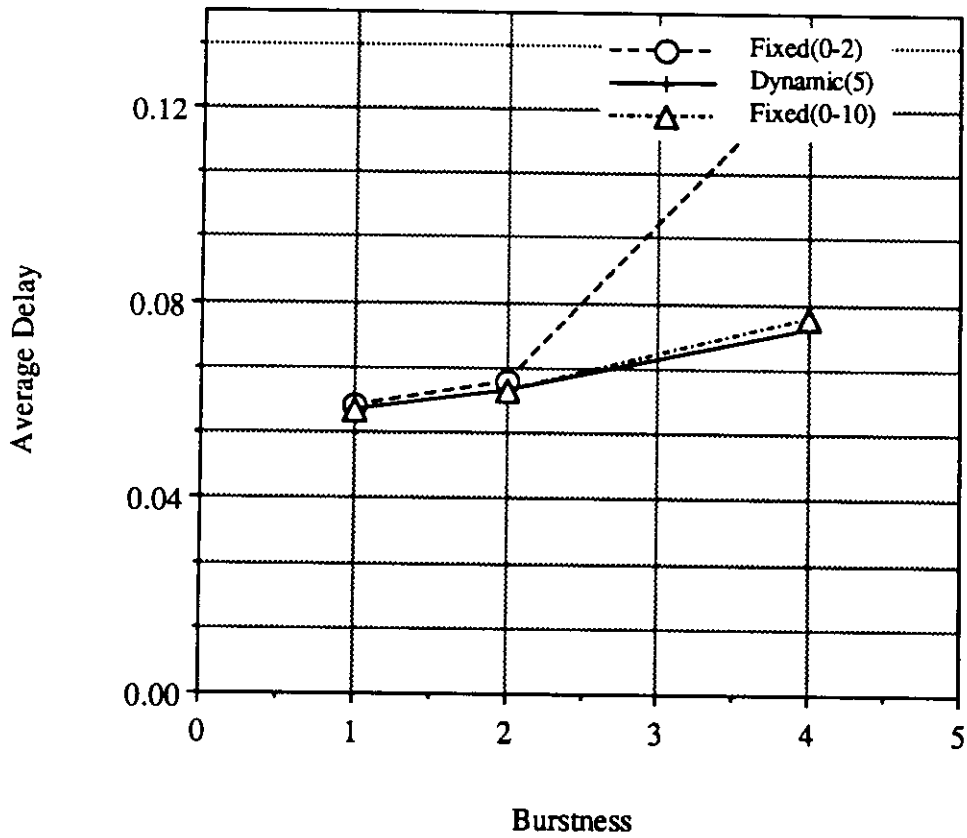


Figure 3.6: Average delay vs. burstiness; burst length = 10; utilization = 25% ; number of hosts = 10

Burst Length = 0.5

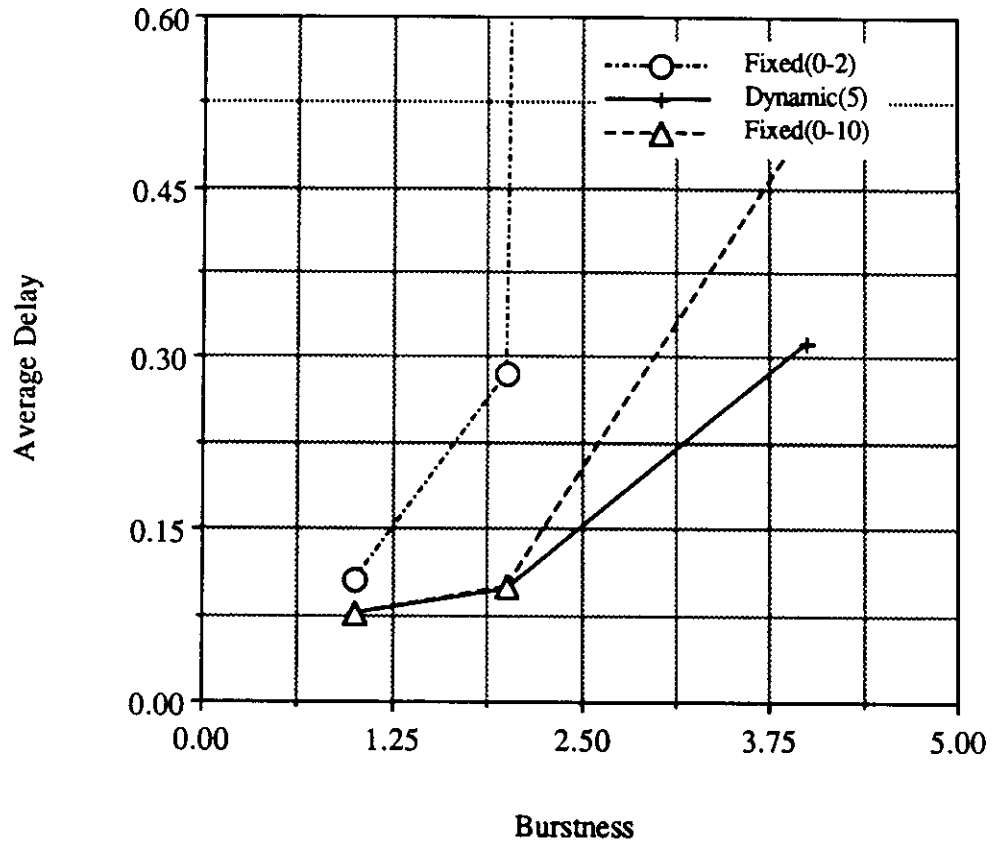


Figure 3.7: Average delay vs. burstiness; burst length = 0.5; utilization = 50% ; number of hosts = 10

- For the parameters given in our system, how many hosts can a central site manage for a given configuration of burstiness, burstlength and arrival rate?
- What is the point beyond which there is no performance advantage in increasing the system size?

For a configuration with burstiness = 2, and arrival rate = 11 we varied the number of hosts from 10 to 50 and the burstlength from 0.1 to 10. From tables 3.1, 3.12, and 3.11 we conclude that at low loads and low burstiness no performance improvement (average delay) is achieved by using larger systems, since the central site has to manage a remote job allocation proportional to system size, and most of the time the fast processors are idle. We observe a minor degradation in the fixed(0-2) algorithm for high burst length, and we attribute that to queueing at the central site due to larger remote traffic.

For a configuration with burstiness = 2, and arrival rate = 22 we varied the number of hosts from 10 to 20 (the 50 hosts example diverged) and the burstlength from 0.1 to 10. In tables 3.3, and 3.13 we are able to observe that for the dynamic tuning algorithms and the fixed(0-10) algorithms there is a major performance improvement at high burst length (10) by doubling the number of hosts. However, for the fixed(0-2) algorithm a major degradation in performance is observed.

For a configuration with burstiness = 4, and arrival rate = 22 we varied the number of hosts from 10 to 50 and the burstlength from 0.1 to 10. In tables 3.5, 3.14, and 3.16 we are able to observe that at low loads and higher burstiness there is no performance improvement by increasing the number of hosts.

For a configuration with burstiness = 4, and arrival rate = 44 we varied the number of hosts from 10 to 20 (the 50 hosts example diverged) and the burstlength from 0.1 to 10. In tables 3.7, and 3.15 we are able to observe that for the dynamic(5)

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.072 ± 0.0002	0.066 ± 0.0001	0.066 ± 0.0001	0.066 ± 0.0001
0.2	0.074 ± 0.0001	0.067 ± 0.0001	0.067 ± 0.0001	0.067 ± 0.0002
0.5	0.077 ± 0.0001	0.068 ± 0.0001	0.068 ± 0.0001	0.068 ± 0.0001
1.0	0.080 ± 0.0001	0.069 ± 0.0002	0.068 ± 0.0002	0.067 ± 0.0002
10.0	0.112 ± 0.0108	0.071 ± 0.0018	0.070 ± 0.0016	0.070 ± 0.0002

Table 3.11: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 50

algorithm and the fixed(0-10) algorithms there is a major performance improvement at medium burst length (1.0) by doubling the number of hosts. However, for the dynamic(1) algorithm a major degradation in performance is observed. This result indicates that for our parameters, high frequency for threshold computation at the central site can result in performance degradation. We will explore this result in more detail in section 3.3.4.

The justification for the optimal cluster size must take into account not only the average delay but also the cost for setting up central sites. Previous work has suggested that an optimal cluster size for homogeneous distributed system is around 30 [Zho87]. Our results seems to indicate that, for our parameters an optimal cluster size would be between 20-30.

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.063 ± 0.0001	0.061 ± 0.00004	0.061 ± 0.0001	0.061 ± 0.0001
0.2	0.063 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001
0.5	0.064 ± 0.00003	0.062 ± 0.00003	0.062 ± 0.0001	0.062 ± 0.0001
1.0	0.064 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001
10.0	0.064 ± 0.0002	0.063 ± 0.0002	0.063 ± 0.0001	0.063 ± 0.0001

Table 3.12: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 11; number of hosts = 20

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.183 ± 0.0070	0.081 ± 0.0001	0.081 ± 0.0002	0.081 ± 0.0002
0.2	0.266 ± 0.0123	0.083 ± 0.0001	0.083 ± 0.0001	0.083 ± 0.0003
0.5	0.534 ± 0.0646	0.087 ± 0.00003	0.087 ± 0.0003	0.087 ± 0.0001
1.0	0.927 ± 0.1592	0.090 ± 0.0003	0.090 ± 0.0009	0.090 ± 0.0003
10.0	INF	0.097 ± 0.0003	0.101 ± 0.0065	0.176 ± 0.0443

Table 3.13: Average delay vs. burst length ; burstiness = 2 ; arrival rate = 22; number of hosts = 20

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.069 ± 0.00003	0.066 ± 0.0001	0.067 ± 0.0001	0.066 ± 0.00004
0.2	0.070 ± 0.0001	0.069 ± 0.0001	0.069 ± 0.0001	0.069 ± 0.0001
0.5	0.072 ± 0.0001	0.071 ± 0.0001	0.071 ± 0.0001	0.071 ± 0.0002
1.0	0.073 ± 0.0001	0.073 ± 0.0003	0.072 ± 0.0002	0.073 ± 0.0002
10.0	0.079 ± 0.0041	0.079 ± 0.0080	0.073 ± 0.0004	0.076 ± 0.0006

Table 3.14: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22;
number of hosts = 20

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	INF	0.1219 ± 0.0003	0.1056 ± 0.0005	0.1280 ± 0.0004
0.2	INF	0.1609 ± 0.0029	0.1249 ± 0.0007	0.1691 ± 0.0016
0.5	INF	0.3233 ± 0.0170	0.1788 ± 0.084	0.2570 ± 0.0034
1.0	INF	0.7257 ± 0.0398	0.3032 ± 0.0240	0.3918 ± 0.0079
10.0	INF	INF	INF	INF

Table 3.15: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 44;
number of hosts = 20

B. L.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
0.1	0.099 ± 0.0011	0.074 ± 0.0001	0.074 ± 0.0002	0.074 ± 0.0002
0.2	0.143 ± 0.0138	0.078 ± 0.0003	0.078 ± 0.0002	0.078 ± 0.0004
0.5	0.347 ± 0.2158	0.085 ± 0.0011	0.084 ± 0.0010	0.085 ± 0.0004
1.0	INF	0.092 ± 0.0017	0.095 ± 0.0023	0.093 ± 0.0028
10.0	INF	0.199 ± 0.0463	0.210 ± 0.0562	0.238 ± 0.0200

Table 3.16: Average delay vs. burst length ; burstiness = 4 ; arrival rate = 22; number of hosts = 50

3.3.3 Speed Ratio between Fast and Slow processors

In this section we evaluate the sensitivity of the algorithms to the speed ratio between fast and slow processors. This is a very important measure since it will indicate how the algorithms behave in face of unpredictable system behavior that affect processor speed. The most common factors that affect processor speed are: varying user workload, operating system versions, memory sizes, different hardware configurations, and bugs. From tables 3.17, 3.18, 3.19, and figure 3.8 we conclude that dynamic tuning outperforms fixed thresholds for all speed ratios. Furthermore, for burstlength of 10, fixed(0-2), and fixed(0-10) diverge for speed ratios of 10, while dynamic tuning still maintain good performance. When the fixed thresholds are matched to the capacities, fixed(0-2) performs well, while it faces major degradation when the thresholds are not tuned. When the fixed thresholds are untuned, dynamic tuning outperforms fixed thresholds by factors of

S. R.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
2	0.105 ± 0.0006	0.083 ± 0.0004	0.079 ± 0.0002	0.099 ± 0.0004
5	0.143 ± 0.0016	0.096 ± 0.0003	0.098 ± 0.00005	0.099 ± 0.0008
10	0.285 ± 0.0045	0.098 ± 0.0004	0.098 ± 0.0005	0.100 ± 0.0008

Table 3.17: Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 0.5; arrival rate = 22; number of hosts = 10

1.5-5.

3.3.4 Sensitivity to threshold computation period (dynamic tuning only)

We have measured the sensitivity of the dynamic tuning algorithm to the threshold computation period, by reporting in all tables values for dynamic(1) and dynamic(5). For almost all results the values are either identical or show minor discrepancies. We have observed anomalies in the case of high load and high burstiness where dynamic(5) outperformed dynamic(1). This indicates that in dynamic environment adapting too fast can be harmful. Tables 3.20, 3.21, 3.22, 3.23, and 3.24 report results when the threshold computation period is varied from 1-20 seconds. Here again, we observe that the delay is almost insensitive to the threshold computation period. Table 3.24 shows the anomaly observed before, also in a situation where the system is stressed.

These results indicate that the main purpose of dynamic tuning is to match thresholds the varying effective capacities, and not to cope with varying loads. The

S. R.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
2	0.110 ± 0.0004	0.088 ± 0.0002	0.083 ± 0.0003	0.108 ± 0.0015
5	0.171 ± 0.0019	0.101 ± 0.0003	0.106 ± 0.0010	0.110 ± 0.0012
10	0.424 ± 0.0096	0.105 ± 0.0007	0.106 ± 0.0009	0.114 ± 0.0026

Table 3.18: Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 1.0 ; arrival rate = 22; number of hosts = 10

S. R.	Fixed(0-2)	Dynamic(1)	Dynamic(5)	Fixed(0-10)
2	0.130 ± 0.0050	0.095 ± 0.0006	0.095 ± 0.0008	0.142 ± 0.0060
5	0.514 ± 0.0511	0.108 ± 0.0010	0.149 ± 0.0104	0.214 ± 0.0206
10	INF	0.211 ± 0.0869	0.219 ± 0.0208	INF

Table 3.19: Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 10.0; arrival rate = 22; number of hosts = 10

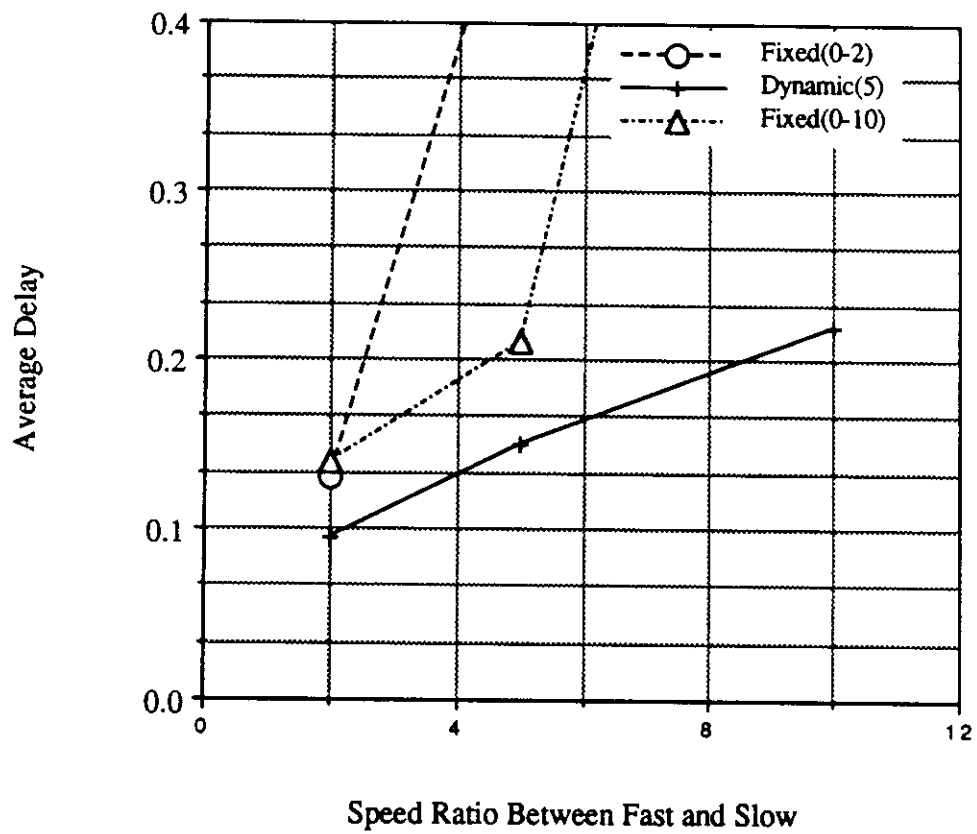


Figure 3.8: Average delay vs. speed ratio between fast and slow processors; burstiness = 2; burst length = 10.0; arrival rate = 22; number of hosts = 10

A. R.	Threshold Computation Period			
	1	5	10	20
11.0	0.062 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0001	0.062 ± 0.0002
22.0	0.211 ± 0.00869	0.219 ± 0.0208	0.258 ± 0.0239	0.231 ± 0.0360

Table 3.20: Average delay vs. threshold computation period ; burstiness = 2 ; number of hosts = 10

A. R.	Threshold Computation Period			
	1	5	10	20
22	0.076 ± 0.0034	0.075 ± 0.0030	0.079 ± 0.0050	0.079 ± 0.0080
44.0	INF	INF	INF	INF

Table 3.21: Average delay vs. threshold computation period ; burstiness = 4 ; number of hosts = 10

varying system loads are taken care of by dynamic load sharing.

3.4 Sensitivity to the system model (dynamic tuning only)

In our evaluations of the dynamic tuning algorithm we have used a $M/M/1$ model in the optimizations. The results indicate that the exact modeling of the arrival process is not critical, since we obtained good results for bursty arrivals, while using a Poisson arrival model.

A. R.	Threshold Computation Period			
	1	5	10	20
5.5	0.06 ± 0.00003	0.06 ± 0.00003	0.06 ± 0.0001	0.058 ± 0.0001
11.0	0.08 ± 0.0001	0.08 ± 0.0002	0.08 ± 0.00005	0.08 ± 0.0001

Table 3.22: Average delay vs. threshold computation period ; number of hosts = 10; Poisson Arrivals

B. L.	Threshold Computation Period			
	1	5	10	20
0.5	0.26 ± 0.0014	0.312 ± 0.0109	0.322 ± 0.0172	0.323 ± 0.0170
0.2	0.156 ± 0.0014	0.161 ± 0.0021	0.161 ± 0.0028	0.161 ± 0.0029
0.1	0.121 ± 0.0006	0.122 ± 0.0018	0.122 ± 0.0006	0.122 ± 0.0003

Table 3.23: Average delay vs. threshold computation period; number of hosts = 10; burstiness = 4; arrival rate = 44

B. L.	Threshold Computation Period			
	1	5	10	20
0.5	INF	INF	1.396 ± 0.1470	1.294 ± 0.1019
0.2	0.328 ± 0.0086	0.386 ± 0.0165	0.377 ± 0.0149	0.205 ± 0.0041
0.1	0.192 ± 0.0025	0.205 ± 0.0039	0.205 ± 0.0041	0.203 ± 0.0028

Table 3.24: Average delay vs. threshold computation period; number of hosts = 10; burstiness = 4; arrival rate = 52

Future research need to be done in this area in order to evaluate what is the real impact of proper modeling in the performance of the algorithm.

3.5 Conclusions and Future Research

The main conclusions of our evaluation are the following:

- The seven algorithms considered can be divided in two groups that exhibit major difference in performance, according to their performance at low burstiness and low utilizations (25 %). The first group is the one that maintains global information about queue length, schedules jobs to the queue with maximum difference between threshold and queue length, and has a threshold of 0 for the slow processors. The second group includes static load sharing, identical threshold of 1 for all sites with shortest queue allocation, and random allocation with local thresholds.

- Random allocation in asymmetric systems produces major degradation in performance as compared to shortest queue allocation.
- Burstiness and burstlength have a major impact on performance on all algorithms.
- The proper tuning of thresholds to the hosts capacities and arrival process is critical. The mismatch between host capacities and thresholds is responsible for a major degradation in the performance of fixed thresholds algorithms, as compared to the same algorithms with tuned thresholds.
- Dynamic tuning of thresholds exhibits the best performance over all algorithms analyzed for range of parameters. The fixed(0-10) exhibits a similar performance at low to medium loads and low burst lengths, since its thresholds are matched to the sites capacities. However, when the system is stressed to its limits we observe that the dynamic tuning is able to improve performance over the fixed(0-10) by factors of 2-2.6.
- We varied the number of host from 10 to 50 and the burstlength from 0.1 to 10. We conclude that at low loads and low burstiness no performance improvement (average delay) is achieved by using larger systems, since the central site has to manage a remote job allocation proportional to system size, and most of the time the fast processors are idle. For medium loads we observed that for the dynamic tuning algorithms, and for the fixed(0-10) algorithms there is a major performance improvement at high burst length (10) by doubling the number of hosts. However, for the fixed(0-2) algorithm a major degradation in performance was observed.

- The justification for the optimal cluster size must take into account not only the average delay but also the cost for setting up central sites. Previous work has suggested that an optimal cluster size for homogeneous distributed system is around 30 [Zho87]. Our results seems to indicate that, for our parameters an optimal cluster size would be between 20-30.
- We evaluated the sensitivity of the algorithms to the speed ratio between fast and slow processors. This is a very important measure since it will indicate how the algorithms behave in face of unpredictable system behaviors that affect processor speed. We concluded that dynamic tuning outperforms fixed thresholds for all speed ratios. Furthermore, for burstlength of 10, fixed(0-2) and fixed(0-10) diverged for speed ratios of 10 while dynamic tuning still maintain bounded performance. When the fixed thresholds are matched to the capacities, fixed(0-2) performs well while it faces major degradation when the thresholds are not tuned. When the fixed thresholds are not tuned dynamic tuning outperforms fixed thresholds by factors of 1.5-5.
- We have measured the sensitivity of the dynamic tuning algorithm to the threshold computation period. The average delay is insensitive to the threshold computation period at low to medium loads. We have observed anomalies in the case of high load and high burstiness were dynamic(5) outperformed dynamic(1). This indicated that in dynamic environments adapting too fast can be harmful. This results indicate that the main purpose of dynamic tuning is to match thresholds the varying effective capacities, and not to cope with varying loads. The varying system loads are taken care of by dynamic load sharing.

We are continuing and extending these studies in the following direction:

Dynamic Tuning Sensitivity to optimization algorithm

In this work we have used a coordinate descent algorithm in order to optimize the system. We propose to use different optimization algorithms with varying degree of complexity and evaluate its impact on the system performance. We propose to investigate further, the impact of system model in the algorithm performance.

Chapter 4

Hard Real Time Systems

“chacham eino medaber bifnei mi sheu gadol mimenu be chochma,
ve eino nichnass le toch divrei chavero,
ve eino nivhal leashiv,
shoel qui inian ve one quealacha,
ve omer al rishon rishon ve al acharon acharon,
ve al ma she lo shama omer: lo shamati,
ve mode al a emet.”

The wise does not talk in front of a wiser man,
And does not interrupt his friend,
And does not answer hastily,
Asks from interest and answers to the point,
And says first first and last last,
And about what he has not heard he says: I have not heard,
And acknowledges the truth.

Chapters of the fathers, chapter 5, par. 7

4.1 Introduction

In this chapter we will address the design of next generation hard real-time systems [Sta88]. First, we exemplify the scenario considered through examples. A computer system controlling a next generation medical environment may be com-

posed of thousands of nodes connected to life monitoring devices. All the nodes will have the full set of tasks that deals with emergency actions built-in. Once an event (e.g. heart failure) is triggered, a hardware interrupt is generated and the proper actions must be taken within a very short period of time (delivery of electrical shock, injection of proper medicine, or emergency attention by the medical staff.) Real time requirements may vary from seconds to microseconds according to the application. If we generalize the use of life monitoring devices into the future, it is conceivable that sophisticated monitoring devices could prevent serious illnesses (e.g. strokes and heart attacks.) The patients won't need to be hospitalized. They should carry monitoring devices that would be connected to regional centers through a packet radio network. The information would be monitored and the proper action to be taken informed to the patient (immediate hospitalization, specific drug ingestion or activation of hardware equipment.) In this case, the real time requirement may well be in the order of microseconds. Another example is the control of spacecrafts . In this case, some tasks must be performed routinely and others only when major events occur. For example, equipment checks for faults must be done periodically. However, prior to takeoff, every equipment needed for launching must be thoroughly checked. Those applications demonstrate a need to combine the execution of periodic tasks with tasks that are generated via hardware interrupts. In this chapter we will model the latter task type by a Poisson process and we will call them Poisson tasks.

The increase in processing speeds available, the decrease in cost of processors, deadline constraints, and fault-tolerance requirements indicate that the solutions for these problems may involve large number of fast processors connected by a fast packet switched network.

In this chapter we evaluate the tradeoffs involved in the design of next genera-

tion hard real-time systems [Sta88], using fast packet switching technology [Jac90], [AD89]. Those systems will be distributed and composed by hundreds of nodes connected through high speed switches based on optical technology. Next generation computer networks, using fast packet switching technology, will be able to deliver hundreds of thousands of packets per second with communication delays on the order of microseconds. Load sharing in this context is very attractive, since we have a situation where scheduling decisions can be made with perfect information. Next generation hard real-time systems will be used in similar environments as in the present, e.g. life monitoring, nuclear power plants, intelligent manufacturing, underseas exploration, air traffic control, spacecrafts, military applications, etc... Failures of such systems occur when a task misses its deadline without running a recover routine in time. This can happen due to hardware failure or task scheduling conflicts. Such failures are often followed by large human and economical losses.

The outline of the remaining of the chapter is as follows. In Section 4.2 we briefly describe the literature reviewed. In Section 4.3 we consider the design of medium size (10 nodes) hard real-time systems, subjected to Poisson arrivals and bursty arrivals. In that section we present the design of a high speed load balancing switch based on fast packet switching techniques, as well as simulation results for the fraction of tasks dropped. In Section 4.4 we consider the design of hard real time systems when the job mix is composed of Poisson tasks and periodic tasks. In section 4.5 we develop an analytical model for the fraction rejected due to deadline constraints. Section 4.6 contains the main conclusions of the chapter.

4.2 Literature Review

The literature in scheduling of tasks under real time constraints is vast [Sta88], [Sta89], [ZRS87], [RSZ89], [SC88], [CL86b], [CKT89a]. Previous work in the area considered either fault-tolerant centralized multiprocessing systems where tasks are run synchronous in replication [AP84] or distributed systems consisting of loosely coupled single processors [Sta89]. In loosely coupled systems the approach has been to use sophisticated local scheduling algorithms, and to schedule in foreign processors only those jobs that cannot be scheduled locally, due to deadline constraints. In next generation systems, the most likely approach will be to use cheap straight forward hardware implemented local scheduling algorithms and schedule tasks remotely whenever a task cannot be successfully scheduled locally.

In [Sta89] the reallocation of hard real-time tasks in a distributed system after a single site failure was considered. The performance metric used was the success ratio, defined as the fraction of tasks that were allocated previously to the failed host and were successfully completed (met their deadline constraints) after reallocation. A decentralized task reallocation algorithm of order $O(n^2)$ was presented and analyzed through simulation. The conclusion was that the decentralized approach achieved a better success ratio than an equivalent centralized algorithm. The intuition behind the improvement was that given a high cost for reallocation $O(n^2)$ and the framework for task reallocation, the distributed algorithm was able to produce a linear speedup for the reallocation of the failed site tasks.

In [ZRS87] the problem of scheduling tasks locally in a node with multiple resource requirements and hard real-time deadlines was considered. An heuristic approach that took into consideration (1) resource requirements, (2) tasks laxity (difference between deadline and computation time) and (3) tasks computation

time, was described. It was shown that policies that considered only one of the three measures performed poorly for the workload simulated. However, if the measures were weighted in a way to tune the scheduler to the offered workload, the heuristic proposed performed well.

In [RSZ89] a set of heuristic algorithms to schedule tasks with deadline and resource requirements in loosely coupled distributed systems was presented. They considered the allocation of tasks that could not be guaranteed locally, and evaluated the performance of four different algorithms: random allocation, focusing addressing (send to a particular node that has the most chance of satisfying the request), bidding and a flexible algorithm that combines the advantage of bidding and focusing addressing. They performed an extensive simulation study to evaluate the effect of communication delay, task laxity, load differences in the nodes and task computation times on the performance of the algorithms. They concluded that hard real-time algorithms can be effectively implemented on the top of a loosely coupled distributed architecture.

In [SC88] a decentralized, dynamic load sharing algorithm for a real-time distributed system built from uniprocessor nodes was proposed and analyzed. Service times and deadlines were considered constant and identical for all tasks. The state of each node was defined as light, full or heavy according to a comparison of the current workload to three predefined thresholds. Each node used state information about a set of nodes that was maintained by a broadcast of state changes. The optimal number of buddies per node was observed to be between 10-15 nodes. They were able to show that, for the parameters used in their workload, the load sharing method proposed succeeded in reducing the probability of deadline misses to a small number. The overhead incurred in collecting global state information could be controlled by an appropriate tuning of the thresholds.

In [CL86b] a receiver-initiated scheduling algorithm for distributed soft real-time system was presented and analyzed through simulation. The algorithm divided jobs in "blessed" (if the deadline was sure to be met), "to-be-late" (if there was uncertainty about the ability to meet the deadline) and "late" (if the deadline would be missed for sure). The performance of the algorithm was compared with that of a similar sender-initiated algorithm and with a simple load sharing algorithm that did not use any deadline information. The sender-initiated algorithm was found to be unstable at high loads because it could not efficiently gather load information when most processors were busy. The sender-initiated algorithm performed better at low loads due to the ability to provide a better service to the high loaded sites. They concluded that receiver-initiated algorithms should be employed in distributed soft-real time systems since the the benefit of better performance at low loads is negligible in comparison to the damage that can be caused by the instability at high loads demonstrated by the sender-initiated algorithm.

In [CKT89a] routing and flow control of a simple parallel network of $M/M/1$ queues subjected to deadline constraints was considered. They posed an optimization problem where the cost function to minimize was the fraction of messages that were rejected either at the entry point(due to flow control) or at the exit point (due to deadline violation). Using the assumption that the cost function was convex, they showed that the optimal solution used only a subset of the links and that for the active set of links the incremental rejection of messages was the same for all links. They also showed that for a FCFS discipline an admissions policy that rejected messages at the entry point was optimal above some critical load value.

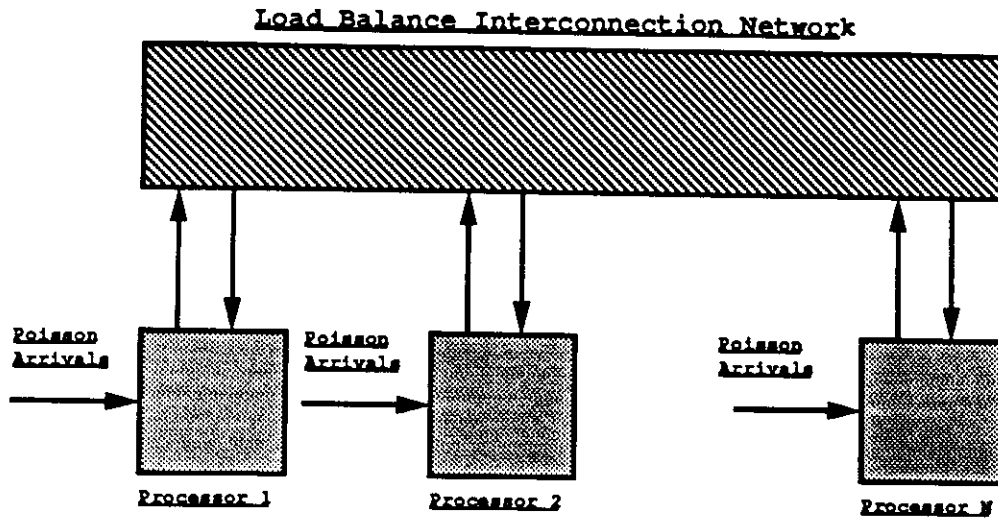


Figure 4.1: Real-time architecture considered

4.3 Evaluation of Real Time Systems Under Poisson Tasks

4.3.1 Model

The model considered for hard real-time systems under Poisson arrivals is described in figure 4.1 and is the following:

1. The system is composed of processing nodes that are connected through a load balancing fast packet switch,
2. the job mix is composed of Poisson tasks that arrive to the local nodes with the following requirements:
 - a truncated exponentially distributed worst case computation time c . The computation time is discretized in slots and ranges from 0 to the maximum number of slots a task can execute. If c is greater than the maximum number of slots the running time is chosen to be the maximum,

- a deadline d that is the sum of c and an exponentially distributed variable l (laxity – maximum time until process start execution)
3. all processors have direct access to the full set of tasks that can be executed.
 4. a task can be scheduled locally or remotely, according to laxity requirements.

4.3.2 Scheduling

We now describe the scheduling mechanics in the system. The time is slotted and maintained globally through a synchronized clock. In a job mix with Poisson tasks only, the scheduler works as follows. At the beginning of a slot, the local scheduler in all nodes accept the first pending Poisson arrival from the external world, and compare the task laxity with the number of busy slots in the local node. A task will be scheduled locally if the laxity is greater or equal to the number of busy slots. At the end of the local acceptance phase each local scheduler will route through the load balancing interconnection network one of two packets as follows (to have a concrete example and to relate to our simulations, we indicate lengths in bits):

- If the local scheduler is scheduling a task for remote execution, it will route a schedule request packet with the following information:
 - Task laxity minus the interconnection network communication delay (4 bits) ,
 - Task identification (8 bits). Since in our architecture we assume that all tasks are previously known to all nodes, the task identification is an index in a task table built-in in every node. The gathering of the data needed to execute the task is responsibility of the own task (probably

done through a data gathering network) and is included in the task computation time,

- If the local scheduler is not requesting a remote task allocation it will route a packet with the following information:
 - Number of busy slots at the node (4 bits),
 - Node address (4 bits).

The interconnection network will either drop the remote request packet (if it couldn't be scheduled properly) or schedule the job to be executed in a node with the number of busy slots less than the laxity. Each local scheduler will end the allocation cycle, as before, by incrementing the number of busy slots by the task requirement allocated to it. The interesting feature in this load sharing algorithm is that remote job allocation is done with perfect knowledge, since the number of busy slots available in each node at the beginning of the interconnection network cycle is the same as when the task is really allocated to run remotely (the slot of the local scheduling algorithm is orders of magnitude larger than the slot of the interconnection network.)

4.3.3 Load Balancing Interconnection Network

In this subsection we describe a version of the interconnection network for the job mix with Poisson tasks only. The load balancing interconnection network is composed of three stages as shown in figure 4.2:

1. a sort network, that sorts all packets according to type (request for allocation or offer of available slots), laxity of the task being scheduled, and number of busy slots in the destination node. Packets will come out of this stage

Load Balance Interconnection Network

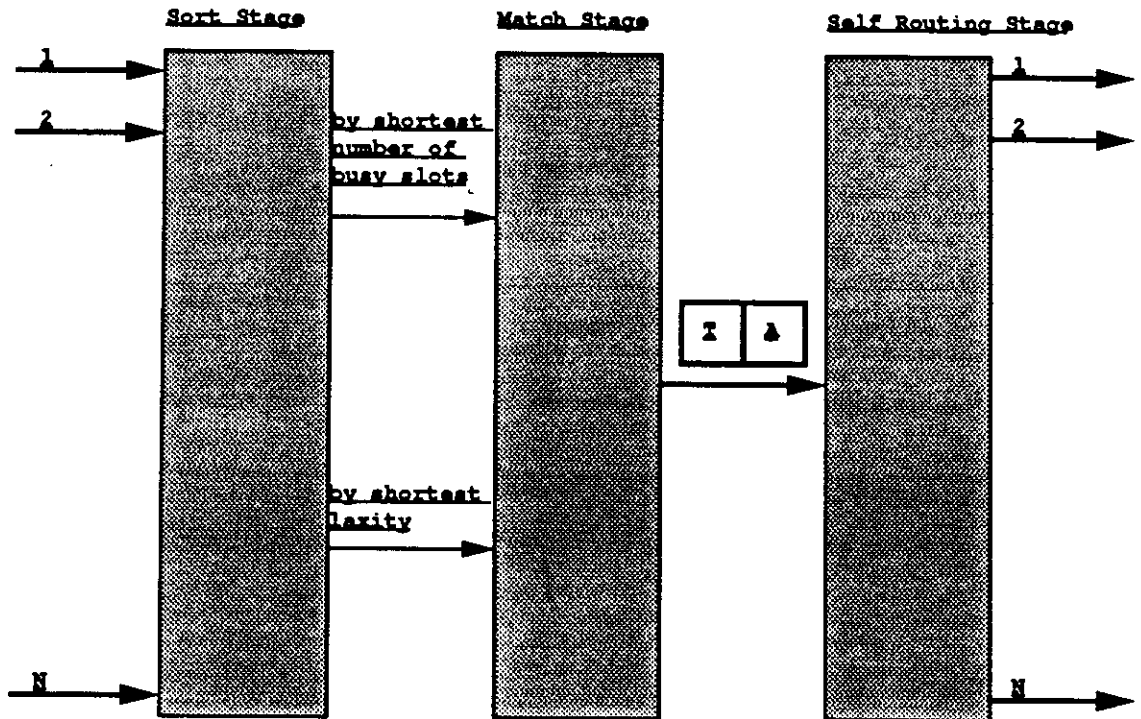


Figure 4.2: Load balancing interconnection network

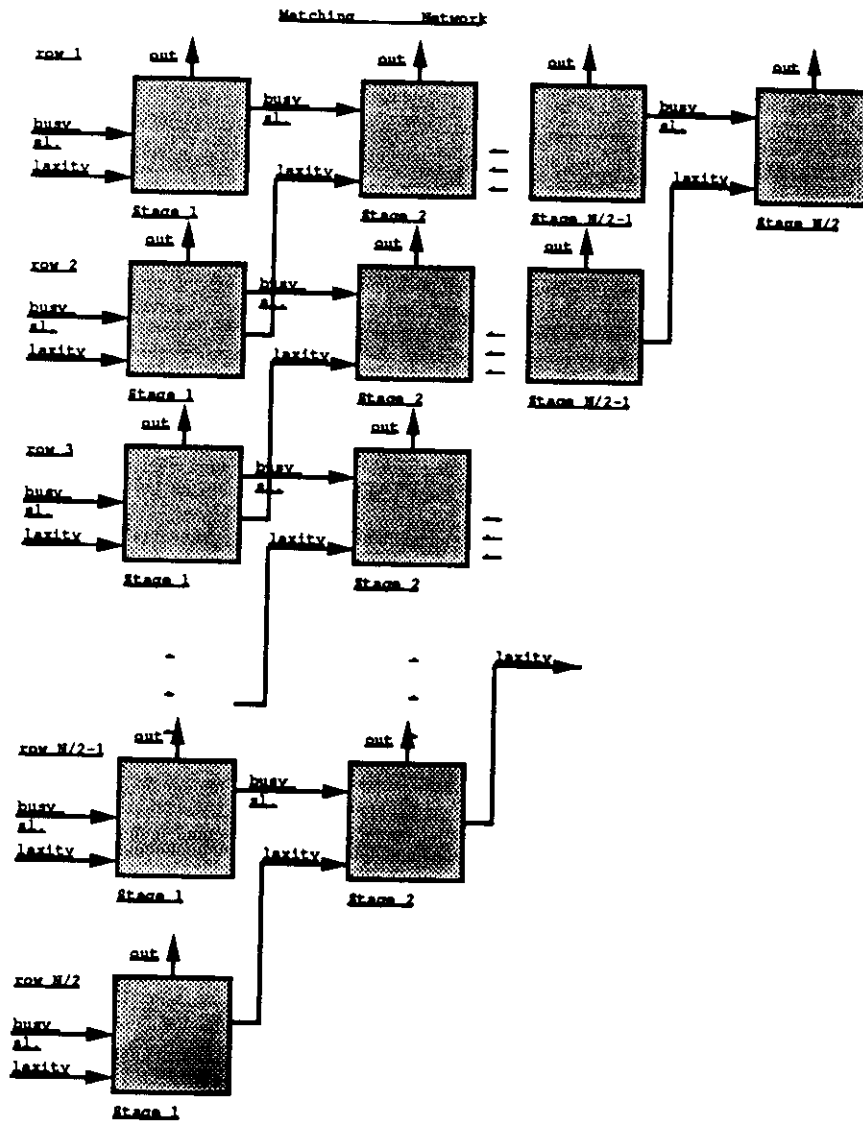


Figure 4.3: Matching Network

sorted by type with a maximum of $N/2$ packets of each type. The first $N/2$ packets will be busy slots packets; the next $N/2$ packets will be laxity packets. Within each group packets will be sorted by shortest busy slots (laxities.)

2. a matching network, that is a multistage network composed of 2×2 elements, $N/2$ stages and $N(N+2)/8$ elements. Each element compares the information in its upper input (number of busy slots) with the information in its lower input (laxity.) If the number of busy slots is less or equal to the laxity a match is found and a packet with the destination address (A) and task identification (T) is routed to the destination through the element output line across the third stage self-routing network. Otherwise, the element copy its input lines to its output lines. The matching network topology is shown in figure 4.3. In the first stage a match is attempted between ordered offers and requests. If the smallest laxity cannot be satisfied by the smallest number of busy slots, it could not be satisfied by any other and is thus dropped. On the other hand, if the larger number of busy slots cannot satisfy the larger laxity it for sure won't be able to satisfy any other task and will also be dropped. In the next stage we have one less element, and the number of busy slots are propagated straight, while the laxities are propagated to the input of the upper element in the next row. The process is repeated until we get to the last stage with only one element, where the largest laxity is compared to the smallest number of busy slots,
3. a self-routing network, that routes the matching pairs to the destination node. At each slot of the matching network the successful packets will be fed into the routing network.

4.3.4 Simulation Results-Scheduling Based on Laxity only

In this section we evaluate the performance of the load sharing algorithm proposed for hard real-time systems through a discrete event simulation. In the following we describe the basic core of the simulation. The events in the simulation are the following:

1. *Poisson task arrival.* A task arrives to a local site. A running time c and a laxity l are selected according to the model described in section 4.3.1. The local scheduling algorithm described in section 4.3.2 is run and the task is classified as local or remote. If the task is local the number of busy slots in the local node is incremented by the number of slots requested by the task; otherwise a remote node that best satisfies the laxity constraint is selected for remote execution (this assumes that a task is scheduled as soon as it arrives; in section 4.3.4.3 we consider the effect of the interconnection network in the system performance.) If no remote node can satisfy the request the task is dropped; otherwise a task arrival event is scheduled to happen at the destination node after the remote scheduling delay,
2. *Remote task arrival.* If the local scheduling algorithm is satisfied the number of busy slots is incremented by the number of slots requested; otherwise the task is dropped,
3. *End of slot.* All busy nodes have the number of busy slots decremented by one.

The simulation results for the fraction of tasks dropped are shown in the following subsections.

Unless stated differently in the section that reports the result, the parameters used were the following:

1. The arrival rates are constant and equal to 0.075 tasks/node/slot.
2. The communication delay for starting a remote task is varied from 0 slots to 1.0 slot.
3. The Poisson tasks laxity is varied from 1 slot to 100 slots.
4. The service time distribution for the Poisson tasks is truncated to maximum of 20 slots exponentially distributed with 10 slots average. This gives roughly 50 % total utilization (the utilization will depend on the fraction dropped.)
5. The system is composed of 10 nodes.
6. Local and remote scheduling is done immediately based on global information.

4.3.4.1 Immediate Scheduling with Perfect Information

Table 4.1 shows the results for the fraction dropped for the model described in section 4.3.4. From the table we learn that communication delay and laxity have a major impact on performance. It demonstrates that, for the environment considered, simple local scheduling algorithms can be used effectively if average laxities are bigger than average processing times, or if communication delay for remote task initiation is less than 1 % the average task processing time.

4.3.4.2 Service times

In this section we investigate the influence of the processor speed in the fraction of tasks dropped. Tasks service times were exponentially distributed with 1 slot

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.046	0.025	0.015	0.0025	0.001053
0.01	0.052	0.027	0.016	0.0027	0.001053
0.1	0.089	0.038	0.022	0.0043	0.001857
0.5	0.21	0.082	0.049	0.011	0.0058
1.0	0.29	0.12	0.078	0.020	0.010

Table 4.1: Fraction dropped vs. average laxity and communication delay; immediate scheduling based only on laxity

Comm. Delay	Laxity		
	1	10	100
0	0.00014	0.00	0.00
0.01	0.00097	0.000027	0.00
0.1	0.0087	0.0010	0.000014
1.0	0.05	0.0085	0.00094

Table 4.2: Fraction dropped vs. average laxity and communication delay; immediate scheduling based only on laxity; 1 slot average processing per task

average (table 4.2), or 5 slots average (table 4.3) truncated to a maximum of 20 slots. Assuming that we can upgrade the network speed by a factor of 10 and the processor speed by the same factor, which will result in the best improvement in the fraction of tasks dropped? A first look at the tables 4.1, 4.2, and 4.3 could indicate that the investment in processing power would produce a better improvement than the investment in the network. This will be true only if the cost/upgrade is the same in both technologies. The true answer for the question will depend on the stage of technology development and on which technology is being used to its limit. Today, using optical technology and fast packet switching, we can achieve an improvement of 1000 in the network speeds. To achieve that same factor in the speeds of processors we would have to resort to very fancy technology that wouldn't be cost effective.

Comm. Delay	Laxity		
	1	10	100
0	0.003	0.0003	0.00
0.01	0.006	0.0008	0.000014
0.1	0.04	0.004	0.0005
1.0	0.20	0.04	0.004

Table 4.3: Fraction dropped vs. average laxity and communication delay; immediate scheduling based only on laxity; 5 slot average processing per task

4.3.4.3 Load Balancing Interconnection Network Evaluation

In this section we evaluate the performance of the load balancing interconnection network proposed in section 4.3. The approach used was to simulate the influence of the interconnection network in the fraction of tasks that were dropped. We did that by delaying the remote scheduling to the boundary of the interconnection network slot. We defined the communication network slot to be equal to the network communication delay, and we varied it while maintaining the processor and scheduler slots constant. Table 4.4 show the results for the fraction dropped. Comparing table 4.4 with table 4.1 we can see that for very small communication delays (0.01 slot) the performance of the interconnection network approaches the perfect knowledge system asymptotically. However, as the communication delay increases, the fraction dropped is increased due to the fact that remote tasks wait, on the average, half a slot plus the communication delay until they get sched-

Comm. Delay	Laxity				
	1	5	10	50	100
0.01	0.053	0.027	0.016	0.0024	0.0010
0.1	0.11	0.043	0.026	0.0050	0.0020
0.5	0.26	0.10	0.062	0.015	0.007
1.0	0.33	0.16	0.10	0.026	0.014

Table 4.4: Fraction dropped vs. average laxity and communication delay; immediate local scheduling based only on laxity ; interconnection network effect included; 10 slot average processing per task

Laxity/ Com. Del.	Number of Nodes							
	1	2	5	10	15	20	25	30
10	0.37	0.26	0.14	0.089	0.074	0.066	0.064	0.062
50	0.29	0.18	0.081	0.037	0.025	0.020	0.018	0.016

Table 4.5: Fraction dropped vs. Number of Nodes; immediate scheduling

uled. If laxities are small, the extra wait prevents successful allocations that were possible before in the system with immediate remote allocation. This result indicates that the approach proposed is very effective in special purpose hard real time architectures, where we can control the delay incurred for remote task allocation.

4.3.4.4 Number of nodes

In this section we evaluate the effect system size (number of nodes) has in the fraction of tasks that are dropped. The number of nodes was varied from 1 to 30. The task fraction dropped vs. the number of nodes is shown in table 4.5. Two ratios of laxity and communication delay were considered. The scheduling algorithm used was the immediate local and remote scheduling.

From the table 4.5 we can see that the performance improvement due to increase in system size levels off around 10 nodes.

4.3.4.5 Local Scheduling Algorithms

Two local scheduling algorithm were considered. The first is the one described in section 4.3 and reported in table 4.1. The second, that we will call delayed

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.057	0.050	0.040	0.022	0.019
0.01	0.061	0.050	0.040	0.021	0.019
0.1	0.098	0.061	0.047	0.024	0.020
0.5	0.22	0.10	0.072	0.034	0.028
1.0	0.34	0.25	0.22	0.17	0.16

Table 4.6: Fraction dropped vs. average laxity and communication delay; delayed scheduling

scheduling, schedule immediately tasks with laxities that are within one slot of the number of busy slots found in the local node; tasks with larger laxities are delayed in a local queue and scheduled when their laxities gets to within one slot of the number of busy slots found in the system. Table 4.6 show the fraction dropped for delayed local scheduling.

The delayed scheduling algorithm, shows worse performance because at 50 % utilization the processors are half the time idle when a task arrives. Since the rational behind the policy is to avoid contention with short laxity tasks, this is achieved by the immediate scheduling algorithm as well. However, large laxity tasks don't take advantage of their laxity (because they are delayed.) In table 4.6 we observe major degradation for high laxity. For large communication delays the

performance is degraded as well since load sharing is not effectively used by the algorithm.

4.3.5 Simulation Results-Scheduling Based on Laxity and Threshold

In this section we evaluate the utilization of a local threshold in order to schedule tasks in the architecture evaluated. Each node has a threshold that is the maximum number of slots that will ever be allocated in that node. The only event in the simulation that is different from section 4.3.4 is the following:

- *Poisson task arrival.* A task arrives to a local site. A running time c and a laxity l are selected according to the model described in section 4.3. The local scheduling algorithm is the following: If the number of busy slots in the local node plus the number of slots required to run the task is less or equal to the threshold and the task laxity is greater than the number of busy slots the task is classified as local; otherwise the task is classified as remote. If the task is local the number of busy slots in the local node is incremented by the number of slots requested by the task; otherwise a remote node that best satisfies the threshold constraint and satisfies the laxity constraint is selected for remote execution. If no remote node can satisfy the request the task is dropped; otherwise a task arrival event is scheduled to happen at the destination node after the remote scheduling delay.

4.3.5.1 Local Scheduling Algorithms

Two local scheduling algorithm were considered as in Section 4.3.4.5.

Table 4.7 and 4.8 show the fraction dropped for delayed and immediate scheduling respectively, when the threshold was 20 for all nodes. The tables show that

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.116	0.103	0.0962	0.084	0.082
0.01	0.121	0.105	0.098	0.084	0.082
0.1	0.150	0.111	0.101	0.086	0.084
0.5	0.24	0.14	0.12	0.094	0.091
1.0	0.38	0.27	0.24	0.21	0.20

Table 4.7: Fraction dropped vs. average laxity and communication delay; delayed scheduling ; threshold = 20

using thresholds, immediate scheduling performs much better than delayed scheduling as in section 4.3.4.5. Comparing table 4.8 with table 4.1, we observe that for the working range (fraction dropped less than 0.1) the use of thresholds degrades performance 1.6 times for communication delay equal to 0.01, laxity equal to 1. For communication delay equal to 0.01, laxity equal to 100, we observed a degradation in performance of 2 orders of magnitude.

In systems with periodic tasks the number of slots that can be allocated in a window is limited. This result indicates that in systems where Poisson tasks constitute a significant fraction of the workload, the system designer may consider allocating them to a load sharing cluster separate from the periodic tasks.

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.083	0.080	0.079	0.078	0.078
0.01	0.087	0.081	0.079	0.078	0.078
0.1	0.12	0.089	0.084	0.079	0.079
0.5	0.21	0.12	0.10	0.085	0.083
1.0	0.28	0.15	0.12	0.091	0.086

Table 4.8: Fraction dropped vs. average laxity and communication delay; immediate scheduling; threshold = 20

Laxity/ Com. Del.	Number of Nodes							
	1	2	5	10	15	20	25	30
10	0.34	0.23	0.14	0.118	0.114	0.114	0.114	0.113
50	0.28	0.18	0.11	0.089	0.087	0.087	0.086	0.086

Table 4.9: Fraction dropped vs. Number of Nodes; immediate scheduling

4.3.5.2 Number of nodes

The task fraction dropped vs. the number of nodes is shown in table 4.9. Two ratios of laxity and communication delay were considered. The scheduling algorithm used was the immediate local scheduling.

From the table 4.9 we can see that the performance improvement due to increase in system size levels off around 10 nodes.

4.3.6 Simulation Results - Bursty Arrivals

In this section we consider that the arrival process is bursty. The arrival process can be in one of two states: active or passive. In the active phase, tasks arrive according to a Poisson process, as in the previous section. We call the length of the active phase: the burst length. In the passive phase the arrival process is blocked, i.e. no tasks arrive. The length of the active and passive phase are exponentially distributed. The arrival process period is defined as the sum of the average active phase length and of average the passive phase length. We evaluate the performance as a function of the arrival process burstiness, defined as the ratio of the the total period to the active phase; and as a function of the burst length. We scale the

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.08	0.05	0.03	0.007	0.003
0.01	0.09	0.05	0.03	0.007	0.004
0.1	0.12	0.06	0.04	0.009	0.004
0.5	0.23	0.10	0.07	0.018	0.009
1.0	0.31	0.15	0.09	0.025	0.014

Table 4.10: Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 1

rate of the Poisson process accordingly, in order to maintain the offered load to the system constant.

4.3.6.1 Burst Length

In this section we consider a system with 10 nodes, subjected to 0.150 Poisson tasks arrival per slot per node. Local allocation is based on laxity only. The burstiness of the process is 2. Tables 4.10, and 4.11 show the results for burst lengths of 1 and 10 respectively. From the tables we can see that for constant burstiness, the burst length has a major impact on performance. The impact of the burst is roughly proportional to the ratio between laxity and communication delay. The ratio between the main diagonal in tables 4.11 and 4.10 ranges between

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.21	0.15	0.11	0.04	0.020
0.01	0.20	0.15	0.11	0.035	0.020
0.1	0.23	0.16	0.12	0.04	0.020
0.5	0.31	0.18	0.14	0.05	0.025
1.0	0.37	0.22	0.16	0.06	0.03

Table 4.11: Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 10

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.11	0.096	0.092	0.087	0.085
0.01	0.12	0.098	0.090	0.088	0.087
0.1	0.15	0.10	0.097	0.089	0.089
0.5	0.25	0.14	0.12	0.093	0.091
1.0	0.31	0.18	0.14	0.10	0.096

Table 4.12: Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 1; threshold = 20

2.14 to 3.0. For the other diagonals the ratio varies in a small range decreasing from 6.6 (for communication delay equal 0, laxity equal 100) to 1.2 (for communication delay equal to 1.0 , laxity equal to 1.0.)

4.3.6.2 Burst Length and Thresholds

In this section we consider that the local allocation is based on laxity and thresholds as in section 4.3.5.

The system modeled is the same as in the previous section. The burstiness of the process is 2. Tables 4.12, 4.13, and 4.14 show the results for burst lengths of 1, 10, and 100 respectively. Comparing table 4.13 with table 4.11, we reinforce the

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.20	0.16	0.14	0.12	0.11
0.01	0.20	0.16	0.14	0.12	0.11
0.1	0.22	0.16	0.15	0.12	0.11
0.5	0.31	0.20	0.17	0.13	0.12
1.0	0.38	0.23	0.19	0.14	0.13

Table 4.13: Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 10; threshold = 20

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.26	0.21	0.19	0.16	0.15
0.01	0.26	0.21	0.19	0.16	0.15
0.1	0.28	0.22	0.20	0.16	0.15
0.5	0.37	0.25	0.22	0.17	0.16
1.0	0.45	0.29	0.24	0.18	0.17

Table 4.14: Fraction dropped vs. average laxity and communication delay; immediate local and remote scheduling; bursty arrivals, burst length = 100; threshold = 20

Comm. Delay	Burstiness			
	8	4	2	4/3
0	0.32	0.17	0.08	0.07
0.01	0.33	0.18	0.09	0.07
0.1	0.34	0.20	0.12	0.10
0.5	0.39	0.29	0.23	0.22
1.0	0.44	0.35	0.31	0.30

Table 4.15: Fraction dropped vs. burstiness and communication delay; bursty arrivals, burst length = 1, laxity = 1; no thresholds

conclusion that in real time systems thresholds degrade performance.

4.3.6.3 Burstiness

Table 4.15 show the effect of burstiness in the fraction dropped for the algorithm without thresholds. The arrival rate was scaled accordingly, in order to maintain the system utilization (if no tasks were dropped) constant. Burstiness degrades performance for low(0.32 vs. 0.07) and high communication delays (0.44 vs. 0.30.) However, for low communication delays the impact is bigger.

Comm. Delay	Laxity				
	1	5	10	50	100
0	0.095	0.073	0.057	0.03	0.026
0.01	0.10	0.072	0.058	0.03	0.024
0.1	0.13	0.085	0.064	0.03	0.028
0.5	0.24	0.12	0.088	0.04	0.04
1.0	0.35	0.26	0.23	0.18	0.17

Table 4.16: Fraction dropped vs. average laxity and communication delay; delayed scheduling; bursty arrivals, burst length = 1, burstiness = 2, no thresholds

4.3.6.4 Local Scheduling Algorithms

Table 4.16 show the fraction dropped for delayed scheduling, for a 10 node system, subjected to 0.150 bursty Poisson tasks arrival per slot per node. The burst length was 1 and the burstiness was 2. No thresholds were used. We observe here degradation in performance due to delayed scheduling as well.

4.4 Evaluation of Real Time Systems Under Periodic Tasks

4.4.1 Model

The model considered for hard real-time systems under periodic tasks is the same as in Section 4.3, with the following additions:

1. The job mix is composed from two job types :periodic jobs that must be executed at a constant rate, and dynamically assigned jobs that arrive according to a Poisson process. The Poisson rates are identical for all nodes,
2. when considering a job mix composed from periodic tasks and Poisson tasks we need to define a window for the execution of the periodic tasks. Each node has a local window that is the maximum number of busy slots that will ever be allocated in that node. The window can be different from node to node. In this section we are considering the window to be constant and identical for all nodes,
3. the job mix with periodic tasks requires a more complex scheduler. In this case each node has a local scheduler for Poisson tasks, that based on a local threshold (the window minus the number of slots required by periodic tasks) and the number of busy slots scheduled for the node, decides if the arriving task will be scheduled locally or remotely.
4. each periodic task arrives at the local nodes with requests for constant service time. The task is always guaranteed to be executed in the next window.

4.4.2 Scheduling

In a job mix with periodic and Poisson tasks, the scheduler works as follows. At the beginning of a slot, the local scheduler in all nodes accept the first pending Poisson arrival or Periodic task from the external world . If the task is periodic it is scheduled to run after the already scheduled tasks in the node, by incrementing the number of local busy slots by the task requirement. If the task is Poisson then the scheduler takes the following action:

1. compare the number of slots requested by the task with the number of slots available in the local node,
2. compare the task laxity with the number of busy slots in the local node.

A task will be scheduled locally if:

1. it satisfies the load balancing constraint, the threshold minus the number of busy slots is greater or equal to the number of slots the task needs to run to completion, and,
2. it satisfies the laxity constraint, the laxity is greater or equal to the number of busy slots.

At the end of the local acceptance phase each local scheduler will route through the load balancing interconnection network one of two packets:

- If the local scheduler is scheduling a task for remote execution, it will route a schedule request packet with the following information:
 - Number of slots requested (4 bits),
 - Task laxity (4 bits) ,
 - Task identification (8 bits).
- If the local scheduler is not requesting a remote task allocation it will route a packet with the following information:
 - Number of slots available at the node (4 bits),
 - Number of busy slots at the node (4 bits),
 - Node address (4 bits).

The interconnection network will either drop the packet or schedule it to be executed in a node with the largest difference between threshold and number of busy slots that satisfies the laxity requirement. At the end of the remote task scheduling in the interconnection network, each local scheduler that had a task allocated to it, will increment the number of busy slots by the task requirement allocated to it.

4.4.3 Load balancing interconnection network – Periodic Tasks

In this subsection we describe a version of the interconnection network for the job mix with Poisson tasks and periodic tasks. The load balance interconnection network will be the same as in section 4.3.3. The only difference here is that the 2 x 2 element in the matching network will have to compare two pairs of information : (available number of slots, required number of slots) and (number of busy slots, task laxity.)

The simulation results for the fraction of tasks dropped are shown in the following subsections. The parameters used were the following:

1. The window was considered constant for all nodes and equal to 20.
2. The thresholds are equal to 15.
3. The arrival rates are constant and equal to 0.075 tasks/node/slot for the Poisson tasks. The service time distribution for the Poisson tasks is truncated to maximum of 15 slots exponentially distributed with 5 slots average. This gives a half the previous utilization for the Poisson tasks.
4. The periodic tasks arrives once in every window (20 slots) and use 5 slots, accounting for another 25 % utilization.

Comm. Delay	Laxity		
	1	10	100
0	0.16	0.073	0.047
0.1	0.19	0.077	0.048
1.0	0.31	0.11	0.048

Table 4.17: Poisson fraction dropped vs. average laxity and communication delay; immediate scheduling with periodic tasks

5. The communication delay for starting a remote task is varied from 0 slots to 1.0 slot.
6. The Poisson tasks laxity is varied from 1 slot to 100 slots.

4.4.4 Periodic and Poisson tasks mix

In this subsection we consider a system with periodic tasks. The service time distribution for the Poisson tasks is truncated to a maximum of 15 slots exponentially distributed with 5 slots average. This gives a 25 % utilization for the Poisson tasks. The periodic tasks arrives once in every window (20 slots) and use 5 slots, accounting for another 25 % utilization.

From the table 4.17 we can observe two main effects due to the introduction of the D/D/1 periodic tasks:

1. large laxity jobs don't get hurt by the smaller window, since they still have enough time to be scheduled successfully. On the contrary, since they account

only for half the total utilization, they are competing with less large laxity jobs with high variability.

2. small laxity jobs are hurt because the periodic tasks are likely to get a fraction of the window it needs.

4.5 Performance Modeling

In this section we will describe three different analytical approaches used in order to model the local and global scheduling algorithms. The first approach uses a global state Markov chain in order to model the system. The main challenge here as in section 2.4 is the explosion of the number of states of the Markov Chain when the system grows large. The second approach treats the system as a single server discrete time bulk arrival system in order to obtain an upper bound in the performance. The third approach computes a lower bound by modeling the system as a modified M/D/c.

4.5.1 Markov Chain Modeling

Our system could be modeled as a Markov chain with state

$S = (N_0, N_1, \dots, N_k, \dots)$ where,

$N_k =$ number of processors with k busy slots.

Note that S completely defines the system, since the scheduling of arrivals depends only on the difference between the task laxity and the number of busy slots. We have a discrete time Markov chain with transitions occurring at the service time boundary. The problem with this approach is that the state space is huge if we maintain the state for any k with reasonable size. As we learned in the

previous sections, a threshold of 20 produces major degradation in performance. So in order to achieve a good approximation we shall keep track of $k \gg 20$. This exceeds the capacity of most available computers since for $k = 20$, $n = 10$ we already have 10 million states in the Markov chain. We present in the following sections simple approximations that enable us to model the system in a very cost effective way.

4.5.2 Bulk Arrivals

In this section we will present an analytical approach in order to compute an upper bound on the fraction of tasks that are dropped. The upper bound results from the fact that we have only one processor. We compute the fraction dropped in a discrete time, single server system subjected to bulk arrivals. The model is the same as in section 4.3.1 with the exception that service times are exponentially distributed (in section 4.3.1 service times were truncated exponential.)

We make the following assumptions:

- The probability of no arrivals in a slot is $e^{-\lambda}$.
- With probability $1 - e^{-\lambda}$ we have at least one arrival in a slot. All of those arrivals will be batched together, and the length of the whole batch will be exponentially distributed with average $1/\mu$. A single laxity will be chosen for the whole batch with average $1/D$.
- The whole batch will be accepted or rejected as a bulk. The probability that a bulk is discarded given that there are k busy slots when it arrives is $1 - e^{-Dk}$

Let's define the following notation:

k - is the number of busy slots in the system,

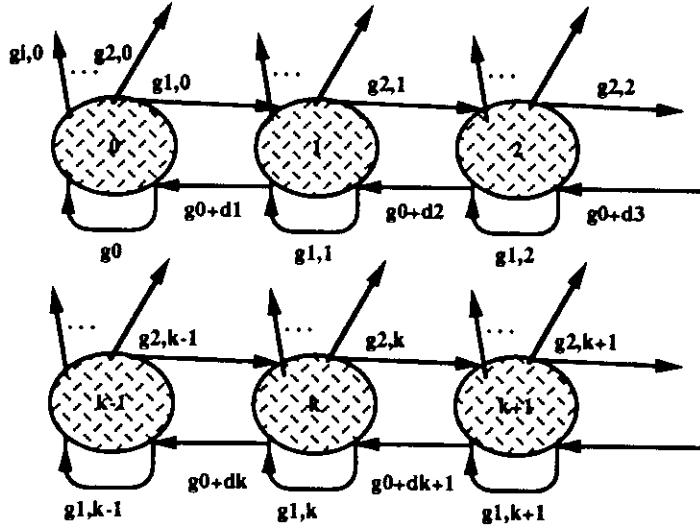


Figure 4.4: Markov chain for bulk arrivals system

p_k - is the steady state probability of having k busy slots in the system,

g_0 - is the probability that there are no arrivals in a slot,

d_k - is the probability that a bulk is discarded given that there are k busy slots in the system when it arrives,

$g_{i,k}$ - is probability that a bulk of i slots is accepted given that there are k busy slots in the system when it arrives.

$$g_{i,k} = e^{-Dk}(e^{-\mu(i-1)} - e^{-\mu(i)})(1 - e^{-\lambda})$$

A bulk of arrivals is accepted if it has a laxity greater or equal to k . It has length i with probability $e^{-\mu(i-1)} - e^{-\mu i}$ and arrives with probability $1 - e^{-\lambda}$.

Under those assumptions we have a discrete time Markov chain as shown in figure 4.4. The system is a bulk arrival single service discrete time system. The set of equations to be solved is:

	Laxity				
	1	5	10	50	100
Simulation	0.41	0.35	0.30	0.17	0.12
Analysis	0.42	0.35	0.30	0.17	0.12

Table 4.18: Fraction dropped vs. average laxity; analysis and simulation

$$p_0 = p_1(g_0 + d_1) + p_0g_0$$

$$p_n = p_{n+1}(g_0 + d_{n+1}) + p_0g_{n,0} + \sum_{k=1}^n p_k g_{n-k+1,k}, n > 0$$

We solve this system of equations numerically. The standard method of Z-transforms would lead in this case to a two dimension transform in (i,k) . Table 4.18 shows the results obtained through analysis and simulation. The agreement is very good demonstrating that the assumptions we made are reasonable (discarding or accepting the whole set of arrivals in a slot is reasonable if the arrival rate/slot is less than 1.)

4.5.3 $M^k/D/c$

In this section we generalize the approach in section 4.5.2 in order to analyze a system with c processors. We will model our system as a $M^k/D/c$. That is, we assume that we have a global queue where we accumulate all arrivals, and that different slots of the same task may be processed in different processors. At the

beginning of a processor slot, the first c slots in the global queue are allocated to be run in the c processors.

We make the following approximations in order to be able to carry the analysis through:

- The probability of no arrivals in a slot is $e^{-c\lambda}$.
- With probability $1 - e^{-c\lambda}$ we have at least one arrival in a slot. All of those arrivals will be batched together, and the length of the whole batch will be exponentially distributed with average $1/\mu$. A single laxity will be chosen for the whole batch with average $1/D$.
- The whole batch will be accepted or rejected as a bulk. The probability that a bulk is discarded given that there are k busy slots when it arrives is $1 - e^{kD/c}$

We use the same notation as in section 4.5.2.

$$g_{i,k} = e^{-kD/c}(e^{-\mu(i-1)} - e^{-\mu i})(1 - e^{-c\lambda})$$

A bulk of arrivals is accepted if it has a laxity greater or equal to k/c . It has length i with probability $e^{-\mu(i-1)} - e^{-\mu i}$ and arrives with probability $1 - e^{-c\lambda}$.

Under those assumptions we have a discrete time Markov chain as shown in figure 4.5. The state of the Markov chain is the number of busy slots in the system.

The set of equations to be solved is:

$$p_0 = p_0 g_0 + \sum_{i=1}^c (g_0 + d_i) p_i$$

$$p_n = p_{n+c} (g_0 + d_{n+c}) + \sum_{i=0}^c p_i g_{n,i} + \sum_{i=1}^{n-1} g_{i,n-i+c} p_{n-i+c}, n > 0$$

Transitions into state $n, n > 0$ occur:

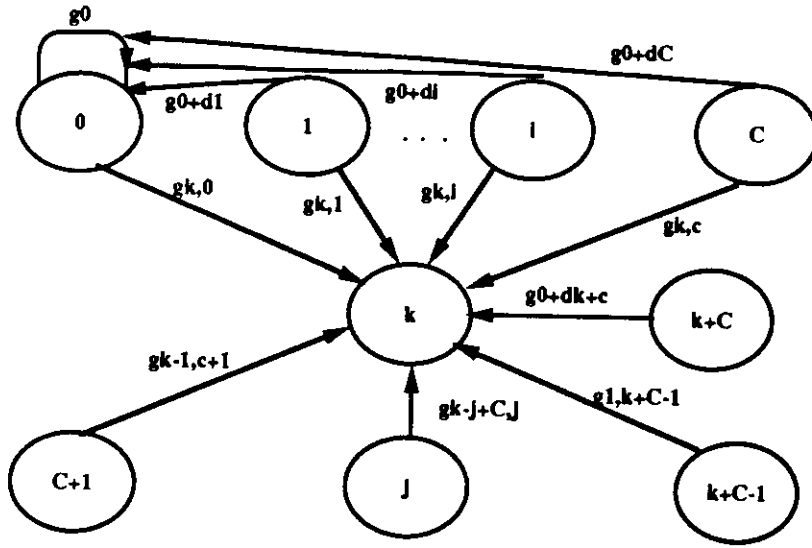


Figure 4.5: Markov chain for $M^k/D/C$ system

1. From state $n + c$, if the arrival bulk is rejected, or if there are no arrivals, with probability $(g_0 + d_{c+n})$;
2. From a state $0 \leq i \leq c$, serve i and n arrive;
3. From a state $c + 1 \leq i \leq c + n$, serve c and arrive $n+c-i$.

Transitions into state 0 occur:

1. From state $i \leq c$, if the arrival bulk is rejected, or if there are no arrivals, with probability $(g_0 + d_i)$;
2. From state 0 if there are no arrivals with probability g_0 .

We solve this system of equations numerically as in section 4.5.2.

Table 4.19 shows the results obtained through analysis and simulation. The agreement is very good demonstrating that the model captures the most important aspects of the physics of the problem. The simulation results shown are for a

	Laxity				
	1	5	10	50	100
Simulation	0.29	0.12	0.078	0.020	0.010
Analysis	0.31	0.14	0.09	0.022	0.012

Table 4.19: Fraction dropped vs. average laxity; analysis and simulation, 10 nodes system with communication delay of 1 slot for remote allocation, and 0 slot for local allocation. In our system arrivals see, on the average, a delay for allocation of 1/2 a slot, since all allocations are done at the boundary of the slot.

4.6 Conclusions and Future Research

The problem of optimal local scheduling in hard real time systems is NP-complete [RSZ89]. The traditional approach for scheduling in loosely coupled hard real time systems has been to use good heuristics for local scheduling [RSZ89]. We have proposed an alternative approach for scheduling in hard real-time systems. We propose to use simple local scheduling algorithms and enforce deadline constraints by using a fast load sharing interconnection network that can schedule tasks with communication delays on the order of microseconds. We have evaluated our approach under Poisson and bursty arrivals assumptions. We have simulated the effect of system size, different local scheduling algorithms, burstiness, burst length, thresholds, synchronous allocation due to the interconnection network, and of a job mix composed of Poisson and periodic tasks. The main conclusions of our evaluation are the following:

- Load sharing can be used effectively in order to schedule jobs in hard real time systems. A simple inexpensive hardware based algorithm for local scheduling can be used together with a fast interconnection network. However, the burstiness of the process must be carefully studied, since it has a major impact on performance.
- The improvement in the performance levels off around 10-15 nodes. This is similar to results obtained for load sharing in homogeneous systems [Zho87] and real time systems [SC88].
- Delayed scheduling of jobs degrades performance. This is because real time systems are designed to be run at low utilizations. The immediate scheduling algorithm should be used.
- In systems with periodic tasks the number of slots that can be allocated in a window is limited. This result indicates that in systems where Poisson tasks constitute a significant fraction of the workload, the system designer may consider allocating them to a load sharing cluster separate from the periodic tasks.
- Laxity and number of busy slots is a good load measure in load sharing in hard real-time systems. Thresholds on queue length degrade performance and shall be avoided.
- Synchronous remote allocation can be used effectively to reduce the fraction of tasks that are dropped. The current approach to improve performance has been to reduce processor utilization. We have shown that reducing communication delay is an effective alternative.

We are continuing and extending these studies in the following directions:

- *Large Systems Topology*

In order to design for large systems we have following alternatives:

1. To expand only the interconnection network in order to manage a larger system, thus constructing a flat structure,
2. To structure the whole system hierarchically and to redesign the load balance interconnection network,
3. To have clusters of small systems that are autonomous and are connected by routers.

We propose to evaluate those alternatives in terms of their cost/performance.

- *Modeling*

We have only touched the performance evaluation of those systems. This is a rich and difficult area. In particular we must account for the proper delay in the interconnection network and the various distributions of deadlines. We propose to use a mini-slot approach in order to model different interconnection network delays. The idea is to express the all units in the system in terms of the interconnection network mini-slot. The state of the Markov chain will be the number of minislots in the system.

- *Fault-tolerance*

We have not addressed the issue of fault-tolerance. One approach should be to have all jobs come in with a priority that indicates its criticality. If the priority is high the job will be started with backups in other nodes with different parent. At high loads those backups may have lower priorities and will use only idle cycles. On node failures, or when they get to a point where

if a node fails they will miss the deadline, their priorities will be increased.
The point here is that many idle cycles can occur due to the fact that jobs
are scheduled based on worst case computation times.

Chapter 5

Conclusions and Future Research

“Se procuro, estou achando. Se acho, ainda estou procurando?”

If I search, I am finding. If I find, am I still searching?

João Guimarães Rosa In: Terceiras Estórias - Tutameia, pp 74

First we summarize the portion of our work applicable to asymmetric distributed computing systems. After formulating the basic load sharing strategies, we performed measurements (Chapter 2) on our testbed under Poisson arrivals in order to obtain the main parameters that characterize the load sharing algorithms under study. The performance evaluation of the dynamic tuning algorithm was done by comparing it to an algorithm where all thresholds were fixed and identical to one. The example reflects a situation where a large number of homogeneous workstations is sharing their residual capacity, defined as the fraction of the processing power not used by the workstation owner. In this case thresholds can not be set a priori to be proportional to the (unknown) available processing power. Our conclusions from the evaluation were the following:

- The self tuning algorithm is able to improve performance for a wide range of arrival rates. The dynamic threshold algorithm is able to reduce the average delay to more than 50% of the value obtained when the fixed threshold of 1 is used. Furthermore, it can support an arrival rate 20% larger than the fixed threshold algorithm.

- Load sharing implementation at the user level incurs high overhead (0.3 secs cpu overhead/job initiation and 20 ms communication delay.) Load Sharing is still very attractive in the environment considered: asymmetric systems with many cpus speeds running jobs with 1M instructions average.
- Fixed and uniform thresholds algorithms in asymmetric environments are highly sensitive to tuning. The thresholds must be well matched to the service demands and capacities of the site in order to achieve good performance. This is the main drawback of this kind of algorithm since workloads must be very well specified in advance. Furthermore the algorithm fails to adapt to varying workloads. The dynamic computation of thresholds solves this problem.
- The dynamic computation of thresholds leads to a better utilization of the more expensive resources since it tends to make better use of the higher capacity servers.
- The simulation model gives very good performance prediction for the dynamic tuning algorithm but disagrees in about 30% for the fixed thresholds algorithm before the knee of the curve. The reason for the disagreement in the fixed threshold algorithm is that it exhibits a larger statistical variation, since the slower hosts are run at very high utilizations.

We proceeded in Chapter 2 to present a new approach for the modeling of dynamic load sharing algorithms in asymmetric systems that employed a global state Markov Chain. The model was validated through simulation and a very good agreement was found at low to medium loads. We managed to analyze large systems as well, by bounding the performance measure (long run average delay.) The lower bound obtained is tight (in the range of arrival rates between 0.2-1.0),

since the states discarded have very low probabilities. The upper bound proposed is tight as well in the range (0.2-1.0).

We evaluated a system with half 4mips processors (slow) and half 40 mips processors (fast) (Chapter 3). The arrival process considered was bursty, and the parameters were scaled from the testbed measurement results. We considered 7 typical algorithms all using centralized routing, but with different amount of information used in the scheduling process.

The main conclusions of our evaluation there were the following:

- The seven algorithms considered can be divided in two groups that exhibit major difference in performance, according to their performance at low burstiness and low utilizations (25 %). The first group is the one that maintain global information about queue length, schedule jobs to the queue with maximum difference between threshold and queue length, and has a threshold of 0 for the slow processors. The second group include static load sharing, identical threshold of 1 for all sites with shortest queue allocation, and random allocation with local thresholds.
- Random allocation in asymmetric systems produces major degradation in performance as compared to shortest queue allocation.
- Burstiness and burstlength have a major impact on performance on all algorithms.
- The proper tuning of thresholds to the hosts capacities and arrival process is critical. The mismatch between host capacities and thresholds is responsible for a major degradation in the performance of fixed thresholds algorithms, as compared to the same algorithms with tuned thresholds.

- Dynamic tuning of thresholds exhibits the best performance over all algorithms analyzed for range of parameters. The fixed(0-10) exhibits a similar performance at low to medium loads and low burst lengths, since its thresholds are matched to the sites capacities. However, when the system is stressed to its limits we observe that the dynamic tuning is able to improve performance over the fixed(0-10) by factors of 2-2.6.
- We varied the number of host from 10 to 50 and the burstlength from 0.1 to 10. We conclude that at low loads and low burstiness there is no performance improvement (average delay) in using larger systems since the central site has to manage a remote job allocation proportional to system size, and most of the time the fast processor are idle. For medium to arrival rates we are able to observe that for the dynamic tuning algorithms and the fixed(0-10) algorithms there is a major performance improvement at high burst length (10) by doubling the number of hosts. However, for the fixed(0-2) algorithm a major degradation in performance was observed. we are able to observe that at low loads and higher burstiness there is no performance improvement by increasing the number of hosts.
- The justification for the optimal cluster size must take into account not only the average delay but also the cost of setting up central sites. Previous work has suggested that an optimal cluster size for homogeneous distributed system is around 30 [Zho88]. Our results seems to indicate that, for our parameters an optimal cluster size would be between 20-30.
- We evaluated the sensitivity of the algorithms to the speed ratio between fast and slow processor. This is a very important measure since it will indicate how the algorithms behave in face of unpredictable system behaviors that

affect processor speed. We concluded that dynamic tuning outperforms fixed thresholds for all speed ratios. Furthermore, for burstlength of 10, fixed(0-2) and fixed(0-10) diverged for speed ratios of 10 while dynamic tuning still maintain bounded performance. When the fixed thresholds are matched to the capacities, fixed(0-2) performs well while it faces major degradation when the thresholds are not tuned. When the fixed thresholds are not tuned dynamic tuning outperforms fixed thresholds by factors of 1.5-5.

- We have measured the sensitivity of the dynamic tuning algorithm to the threshold computation period. The average delay is insensitive to the threshold computation period at low to medium loads. We have observed anomalies in the case of high load and high burstiness were dynamic(5) outperformed dynamic(1). This indicated that in dynamic environments adapting too fast can be harmful. This results indicate that the main purpose of dynamic tuning is to match thresholds the varying effective capacities, and not to cope with varying loads. The varying system loads are taken care of by dynamic load sharing.

We turn now to a summary of our work in real-time systems. The problem of optimal local scheduling in hard real time systems is NP-complete [RSZ89]. The traditional approach for scheduling in loosely coupled hard real time systems has been to use good heuristics for local scheduling [RSZ89]. We have proposed an alternative approach for scheduling in hard real-time systems. We propose to use simple local scheduling algorithms and enforce deadline constraints by using a fast load sharing interconnection network that can schedule tasks with communication delays on the order of microseconds. We have evaluated our approach under Poisson and bursty arrivals assumptions. We have simulated the effect of

system size, different local scheduling algorithms, burstiness, burst length, thresholds, synchronous allocation due to the interconnection network, and of a job mix composed of Poisson and periodic tasks. The main conclusions of our evaluation were the following:

- Load sharing can be used effectively in order to schedule jobs in hard real time systems. A simple inexpensive hardware base algorithm for local scheduling can be used together with a fast interconnection network. However, the burstiness of the process must be carefully studied, since it has a major impact on performance.
- The improvement in the performance levels off around 10-15 nodes. This is similar to results obtained for load sharing in homogeneous[Zho87] systems and real time systems [SC88].
- Delayed scheduling of jobs degrades performance. This is because real time systems are designed to be run at low utilizations. The immediate scheduling algorithm should be used.
- In systems with periodic tasks the number of slots that can be allocated in a window is limited. This result indicates that in systems where Poisson tasks constitute a significant fraction of the workload, the system designer may consider allocating them to a load sharing cluster separate from the periodic tasks.
- Laxity and number of busy slots is a good load measure in load sharing in hard real-time systems. Thresholds on queue length degrade performance and shall be avoided.

- Synchronous remote allocation can be used effectively to reduce the fraction of tasks that are dropped. The current approach to improve performance has been to reduce processor utilization. We have shown that reducing communication delay is an effective alternative.

This research has opened many areas for continuation, some of which were mentioned in their respective Chapters, and which we summarize here.

We have proposed the following as future research in the area of modeling:

- *Optimal Job Mix*

The optimal way to mix different classes of jobs(e.g., Short and long cpu intensive and Interactive) in order to minimize average delay. This question involves the modeling of multiple class load sharing asymmetric systems and the problem of workload characterization since we usually do not know what are the job demands for resources a priori.

- *Configuration Design*

The configuration design of such systems under load sharing control requires an efficient performance modeling algorithm. We intend to study the suitability of our performance bounds to the configuration design of such systems.

We have proposed to continue the sensitivity analysis as follows:

Dynamic Tuning Sensitivity to optimization algorithm

In this work we have used a coordinate descent algorithm in order to optimize the system. We propose use try different optimization algorithms with varying degree of complexity and evaluate its impact on the system performance. We propose to investigate further, the impact of system model in the algorithm performance.

As future research in hard real-time systems we have proposed the following:

- *Large Systems Topology*

In order to design for large systems we have following alternatives:

1. To expand only the interconnection network in order to manage a larger system, thus constructing a flat structure,
2. To structure the whole system hierarchically and to redesign the load balance interconnection network,
3. To have clusters of small systems that are autonomous and are connected by routers.

We propose to evaluate those alternatives in terms of their cost/performance.

- *Modeling*

We have only touched the performance evaluation of those systems. This is a rich and difficult area. In particular we must account for the proper delay in interconnection network and various distributions of deadlines.

- *Fault-tolerance*

We have not addressed the issue of fault-tolerance. One approach should be to have all jobs come in with a priority that indicates its criticality. If the priority is high the job will be started with backups in other nodes with different parent. At high loads those backups may have lower priorities and will use only idle cycles. On node failures, or when they get to a point where if a node fails they will miss the deadline, their priorities will be increased. The point here is that many idle cycles can occur due to the fact that jobs are scheduled based on worst case computation times.

”Isto é conto ou não é conto?”

Discute a crítica, acesa.

E o contista, meio tonto,
é que tem menos certeza.”

Is that a short story or not?

Debate the critics, in heat.

An the author, half dizzy,

is the most unsure.

Carlos Drummond de Andrade, In: Viola de Bolso

Bibliography

- [AC88] R. Alonso and L. Cova. Sharing jobs among independently owned processors. In *8th Int'l Conf. on Distributed Computing Systems*, pages 282–288, January 1988.
- [AD89] H. Ahamadi and W.E. Denzel. A survey of modern high-performance switching techniques. *IEEE Journal on Selected Areas in Communications*, 7(7):1091–1103, september 1989.
- [AGCK89] A. Avritzer, M. Gerla, J. W. Carlyle, and W. J. Karplus. Load balancing in a distributed transaction system. In *IEEE SICON 89*, July 1989.
- [AGR+89] A. Avritzer, M. Gerla, B. A. Ribeiro, J. W. Carlyle, and W. J. Karplus. Analytical modeling of dynamic load sharing algorithms in distributed asymmetric systems. In *IASTED 89*, November 1989.
- [AGR+90] A. Avritzer, M. Gerla, B. A. Ribeiro, J. W. Carlyle, and W. J. Karplus. The advantage of dynamic tuning in distributed asymmetric systems. In *INFOCOM 90*, June 1990.
- [Alo86] Rafael Alonso. *Query Optimization in Distributed Databases Through Load Balancing*. PhD thesis, UC , Berkeley, 1986.
- [AP84] A. Avizienis and J. P. Kelly. Fault tolerance by design diversity: concepts and experiments. *IEEE Computer*, 17(8):67–80, 1984.
- [AR89] I.F. Akyildiz and H.E. Asdtudillo R. Heterogeneous queueing systems with load dependent behavior and different scheduling disciplines. Technical report, School of Information and CS - Georgia Inst. of Tech., Atlanta , GA 30332, February 1989.
- [Avi67] A. Avizienis. Design of fault-tolerant computers. In *1967 Fall Joint Comput. Conf., Afips Conf. Proc.*, pages 733–743, Washington, D.C., 1967. Thompson.
- [BACW89a] J. Betser, A. Avritzer, J.W. Carlyle, and W.J. Karplus. Configuration synthesis for a heterogeneous backbone cluster and a pc-interface

- network. In *Acm Computer Science Conference*, Louisville, Kentucky, Feb 89.
- [BACW89b] J. Betser, A. Avritzer, J.W. Carlyle, and W.J.Karplus. Performance modeling and analysis for a large heterogeneous distributed system:ucla-seasnet. In *IEEE Infocom*, Ottawa, Canada, April 89.
- [Bet88] J. Betser. *Performance Evaluation and Prediction for Large Heterogeneous Distributed Systems*. PhD thesis, UCLA, Los Angeles , CA 90024, November 1988.
- [BF81] Raymond M. Bryant and Raphael A. Finkel. A stable distributed scheduling algorithm. In *2nd Int'l Conf. on Distributed Computing Systems*, pages 314–323. IEEE, July 1981.
- [BK88] F. Bonomi and A. Kumar. Adaptive optimal load balancing in a heterogeneous multiserver system with a central job scheduler. In *8th Int'l Conf. on Distributed Computing Systems*, pages 500–507, 1988.
- [BW89] K. M. Baumgartner and B. W. Wah. Gammon: A load balancing strategy for local computer systems with multiaccess networks. *IEEE Transactions on Computers*, 38(8):1098–1109, August 1989.
- [CDIY87] B. Ciciani, D. M. Dias, B.R. Iyer, and P.S. Yu. On hybrid distributed centralized systems. RC 12698, IBM, T.J. Watson Research Center, April 1987.
- [CDP88] B. Ciciani, D.M. Dias, and P.S.Yu. Load sharing in hybrid distributed/centralized database systems. In *IEEE DCS Conference*, January 1988.
- [CDY86] D. W. Cornell, D.M. Dias, and P.S. Yu. On multisystem coupling through function request shipping. *IEEE Transactions on Software Engineering*, 12(10), October 1986.
- [CDYA87] B. Ciciani, D. M. Dias, P. Yu, and A. Avritzer. Analysis of protocols for hybrid distributed-centralized database systems. RC 58858, IBM, T.J. Watson Research Center, September 1987.
- [CG89] A. E. Conway and N. D. Georganas. *Queueing Networks – Exact Computational Algorithms: A Unified Theory Based on Decomposition and Aggregation*. Computer Systems Series. MIT Press, 1989.

- [CK79] Y. Chow and W. H. Kohler. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Transactions on Computers*, c-28(5):354–361, May 1979.
- [CK86] T. L. Casavant and J. G. Kuhl. A formal model of distributed decision-making and its application to distributed load balancing. In *6th Int'l Conf. on Distributed Computing Systems*, pages 232–291. IEEE, September 1986.
- [CKT89a] C. S. Cassandras, M.H. Kallmes, and D. Towsley. Optimal routing and flow control in networks with real-time traffic. In *INFOCOM 89*, pages 784–791, April 1989.
- [CKT89b] R. Chipalkatti, J. F. Kurose, and D. Towsley. Scheduling policies for real-time and non-real-time traffic in a statistical multiplexer. In *IEEE INFOCOM 89*, pages 774–783, 1989.
- [CL86a] M. J. Carey and H. Lu. Load balancing in a locally distributed database system. In *Int'l Conf. On Management of Data*, pages 108–119, Washington, D.C., May 1986. ACM SIGMOD.
- [CL86b] H-Y Chang and M. Livny. Distrib. scheduling under deadline constraints, a comparison of sender-initiated and receiver-initiated approaches. In *5th Real-Time Systems Symp.*, 1986.
- [CLL85] M. J. Carey, M. Livny, and H. Lu. Dynamic task allocation in a distributed database system. In *5th Int'l Conf. on Distributed Computing Systems*, pages 282–291. IEEE, March 1985.
- [Cou77] P. J. Courtois. *Decomposability, Queueing and Computer System Applications*. ACM Monograph Series. Academic Press, 1977.
- [Cou82] P. J. Courtois. Error minimization in decomposable stochastic models. In E. Coffman, R. Graham, and Kuck D, editors, *Applied Probability - Computer Science The Interface*. Birkhäuser, 1982.
- [CS84] P. J. Courtois and P. Semal. Bounds for the positive eigenvectors of nonnegative matrixes and for their approximation by decomposition. *Journal of the ACM*, 31(4):804–825, 1984.
- [CS86] P. J. Courtois and P. Semal. Computable bounds for conditional steady-state probabilities in large markov chains and queueing models. *IEEE Transactions on Selected Areas in Communications*, 4(6):926–937, September 1986.

- [DIRY87a] D. M. Dias, B. R. Iyer, J. T. Robinson, and P. S. Yu. Design and analysis of integrated concurrency-coherency controls. In *VLDB 87*, September 1987.
- [DIRY87b] D. M. Dias, B. R. Iyer, J. T. Robinson, and P. S. Yu. Trade-offs between coupling small and large processors for transaction processing. Technical report, IBM, T.J. Watson Research Center, 1987.
- [dSeSG84] E. de Souza e Silva and Mario Gerla. Load balancing in distributed systems with multiple classes and site constraints. In E. Gelenbe, editor, *Performance 84*, pages 17–33, Amsterdam, 1984. North Holland.
- [dSeSG87] E. de Souza e Silva and Mario Gerla. Queueing network models for load balance in distributed systems. November 1987.
- [DYB87] D. M. Dias, P. S. Yu, and B.T. Bennett. On centralized versus geographically distributed database systems. Technical report, IBM, T.J. Watson Research Center, 1987.
- [ELZ86a] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed system. *IEEE Transactions on Software Engineering*, SE-12(5):662–675, May 1986.
- [ELZ86b] D. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Evaluation*, 6(1):53–68, March 1986.
- [ELZ88] D. Eager, E. Lazowska, and J. Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *Sigmetrics 88*, pages 63–72, May 1988.
- [ET87] A. M. Eikeboom and H. C. Tijms. Waiting-time percentiles in the multi-server $m^X/g/c$ queue with batch arrivals. *Probability in Engineering and Informational Sciences*, 1:97–115, 1987.
- [Fer86] D. Ferrari. A study of load indices for load balancing schemes. UCB/CSD 86/262, UCB, Berkeley, CA 94720, 1986.
- [FGK73] L. Fratta, M. Gerla, and L. Kleinrock. The flow deviation method - an approach to store-and-forward communication network design. *Networks*, 3, 1973.

- [Fox66] B. Fox. Discrete optimization via marginal analysis. *Management Science*, November 1966.
- [Fra69] H. Frank. Design of economical offshore natural gas pipeline networks. Technical Report R-1, Office of Emergency Preparedness, Washington, DC, January 1969.
- [GNT88] L. Georgiadis, C. Nikolaou, and A. Thomasian. A fair workload allocation policy for heterogeneous systems. RC 14323, IBM T.J. Watson Research Center, P.O.Box 218 Yorktown Heights, NY 10598, August 1988.
- [Gre88] J. J. Green. *Load Balance Algorithms for Computer Networks*. PhD thesis, UCLA, Los Angeles, CA 90024, 1988.
- [Haj85] B. Hajek. Extremal splittings of point processes. *Mathematics of Operations Research*, 10(4):543-556, 1985.
- [HJ86] A. Hac and T.J. Johnson. A study of dynamic load balancing in a distributed system. In G. Serrazi, editor, *ACM SIGCOMM*, pages 348-356, Amsterdam, Netherlands, August 1986.
- [HL86] C. H. Hsu and J. W. Liu. Dynamic load balancing algorithms in homogeneous distributed systems. In *6th Int'l Conf. on Distributed Computing Systems*, pages 216-223, 1986.
- [HL87] C. Hsieh and S.S. Lam. Two classes of performance bounds for closed queueing networks. *Performance Evaluation*, 7:3-30, 1987.
- [HS82] D. P. Heyman and M. J. Sobel. *Stochastic Models in Operations Research: Stochastic Processes and Operating Characteristics*, volume I. MacGraw Hill, 1982.
- [HTT89] J. Hong, X. Tan, and D. Towsley. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Transactions on Computers*, 38(12):1736-1744, 1989.
- [Jac90] A.R. Jacob. A survey of fast packet switches. *Computer Communication Review*, 20(1):54-64, January 1990.
- [Kam86] Hisao Kameda. Effects of job loading policies for multiprogramming systems in processing a job stream. *ACM Transactions on Computer Systems*, 4(1):71-106, February 1986.

- [KL87] P. Krueger and M. Livny. Load balancing, load sharing and performance in distributed systems. Technical report, Computer Sciences Dep. - Univ. of Wisconsin- Madison, August 1987.
- [Kle75] L. Kleinrock. *Queueing Systems, Volume I: Theory*. Wiley-Interscience, New York, 1975.
- [Kle76] L. Kleinrock. *Queueing Systems, Volume II: Computer Applications*. Wiley-Interscience, New York, 1976.
- [KS89] J. F. Kurose and R. Simha. A microeconomical approach to optimal resource allocation in distributed computer systems. *IEEE Transactions on Computers*, 38(5):705–717, May 1989.
- [Lav83] S. S. Lavemberg, editor. *Computer Performance Modeling Handbook*. Notes and Reports in Computer Science and Applied Mathematics. Academic Press, 1983.
- [Lav89] S. S. Lavemberg. A perspective on queueing models of computer performance. *Performance Evaluation*, 10:53–77, 1989. Originally appeared in: *Queueing Theory and Its Applications - Liber Amicorum for J. W. Cohen*.
- [LM82] M. Livny and M. Melman. Load balancing in homogeneous broadcast distributed systems. In *ACM Computer Network Performance Symposium*, pages 47–55, April 1982.
- [LO86] W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *Sigmetrics 86*, pages 54–69, 1986.
- [LT86a] K. J. Lee and D. Towsley. A comparison of priority based decentralized load balancing policies. In *Sigmetrics 86*, pages 70–77, 1986.
- [LT86b] K. J. Lee and D. Towsley. On the advantage of decentralized distributed load balance policies in heterogeneous distributed systems. 1986.
- [LT86c] K. J. Lee and D. Towsley. Quasi-static decentralized load balancing with site constraints. 1986.
- [LY89] M. Lee and J. R. Yee. An efficient near-optimal algorithm for the joint traffic an trunk routing problem in self-planning networks. In *IEEE INFOCOM 89*, pages 127–135, 1989.

- [MdSeSG89] R. R. Muntz, E. de Souza e Silva, and A. Goyal. Bounding availability of repairable computer systems. In *Performance 89 and Sigmetrics 89 Joint Conference*, May 1989. also, *IEEE Trans. on Comp.* December 1989.
- [MTS89] R. Mirchandaney, D. Towsley, and J.A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. March 1989.
- [Neu81] M. Neuts. *Matrix-Geometric Solutions in Stochastic Models*. Johns Hopkins Univ. Press, 1981.
- [NT85] R. Nelson and D. Towsley. On maximizing the number of departures before a deadline on multiple processors. RC 11255, IBM T.J. Watson Research Center, P.O.Box 218 Yorktown Heights, NY 10598, July 1985.
- [Ort87] J. M. Ortega. *Matrix Theory: A Second Course*. Plenum Press, 1987.
- [PFTV86] W. H. Press, B. P. Flannery, S.A. Teukolsky, and W. T. Vetterling. *The Art of Scientific Computing*. Cambridge University Press, 1986, 1986.
- [Ros89] S. M. Ross. *Introduction to Probability Models*. Academic Press, 4 edition, 1989.
- [RP88] Z. Rosberg and P.Kermany. Customer routing to different servers with complete information. RC 61744, IBM, T.J. Watson Research Center, May 1988.
- [RS84] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Software*, 7:65–74, July 1984.
- [RSZ89] K. Ramamritham, J. A. Stankovic, and W. Zhao. Distributed scheduling of tasks with deadlines and resources requirements. *IEEE Transactions on Computers*, 38(8):1110–1123, August 1989.
- [Rud76] H. Rudin. On routing and delta routing: A taxonomy and performance comparison of techniques for packet-switched networks. *IEEE Transactions on Communications*, COM-24(1):43–59, January 1976.
- [RY89] K. W. Ross and D. D. Yao. Optimal dynamic scheduling in jackson networks. *IEEE Transactions on Automatic Control*, 34(1):47–53, January 1989.

- [SC88] K.G. Shin and Y.C. Chang. Load sharing in distributed real-time systems with broadcast of state changes. Technical report, International Computer Science Institute, 1947 Center Street, Suite 600 Berkeley, CA 94704-1105, October 1988. Also, *IEEE Trans. Comp.* August 1989.
- [Sch87] M. Schwartz. *Telecommunication Networks*. Addison Wesley, 1987.
- [SRC85] J. A. Stankovic, K. Ramamrithan, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, 34(12):1130–1145, December 1985.
- [Sta88] J.A. Stankovic. Real-time computing systems: The next generation. COINS 88-06, Dep. of Computer and Information Science, Univ. of Massachusetts Amherst, MA 01003, January 1988.
- [Sta89] J.A. Stankovic. Decentralized decision making for task reallocation in a hard real-time system. *IEEE Transactions on Computers*, 38(3):341–355, March 1989.
- [Sti85] S. Stidham. Optimal control of admission to a queueing system. *IEEE Transactions on Automatic Control*, 30(8):705–713, August 1985.
- [SW88] S. Shenker and A. Weinrib. Asymptotic analysis of large heterogeneous queueing systems. In *Sigmetrics 88*, pages 56–62, May 1988.
- [SW89] S. Shenker and A. Weinrib. The optimal control of heterogeneous queueing systems: A paradigm for load sharing and routing. *IEEE Transactions on Computers*, 38(12):1724–1735, December 1989.
- [Tho87] A. Thomasian. A performance study of dynamic load balancing in distributed systems. In *7th Int'l Conf on Distributed Computing Systems*, September 1987.
- [Tij86] H. C. Tijms. *Stochastic Modelling and Analysis: A Computational Approach*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, 1986.
- [TRTN89] W.T. Tsai, C. V. Ramamoorthy, W. K. Tsai, and O. Nishigushi. An adaptive hierarchical routing protocol. *IEEE Transactions on Computers*, 38(8):1059–1073, August 1989.

- [TT85] A.N. Tantawi and D. Towsley. Optimal static load balancing in distributed computer systems. *Journal of the ACM*, 32(2):445–465, April 1985.
- [TTJ88] A.N. Tantawi, D. Towsley, and J. Wolf. Optimal allocation of multiple class resources in computer systems. In *Sigmetrics 88*, pages 253–260. May 1988.
- [WM85] Y-T. Wang and R. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, May 1985.
- [WS88] A. Weinrib and S. Shenker. Greed is not enough: Adaptive load sharing in large heterogeneous systems. In *IEEE INFOCOM 88*, 1988.
- [YBL87] P. S. Yu, S. Balsamo, and Y. H. Lee. On dynamic load sharing and transaction routing. Technical report, IBM, T.J. Watson Research Center, 1987.
- [YCDI87] P.S. Yu, D. W. Cornell, D.M. Dias, and Balakrishna R. Iyer. Analysis of affinity based routing in multi-system data sharing. *Performance Evaluation*, 7:87–109, 1987.
- [Zho87] S. Zhou. *Performance Studies of Dynamic Load Balancing in Distributed Systems*. PhD thesis, UCB, Berkeley, September 1987. also Tech. Report 87.6 EECS-UCB.
- [Zho88] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.
- [ZR86] M. Zukerman and I. Rubin. On multi-channel queueing systems with fluctuating parameters. In *IEEE Infocom 86*, 1986.
- [ZR87] M. Zukerman and I. Rubin. Queueing performance of a multi-channel system under bursty traffic conditions and state dependent service rates. *Journal of Australian Telecomm Research*, 21(2):3–16, 1987.
- [ZRS87] W. Zhao, K. Ramamrithan, and J. A. Stankovic. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Transactions on Software Engineering*, 13(5):564–577, May 1987.

- [Zuk85] M. Zukerman. *Performance of Queuing and Telecommunication Systems under Mode and Time Dependent Traffic Models*. PhD thesis. UCLA, Los Angeles, 1985.

