Adaptive Threshold-based Scheduling for Real-Time and Non-Real-Time Tasks

By

WENJING ZHU

B.Sc., Peking University, China, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 1991

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date August 27, 1991

Abstract

This thesis documents our study on scheduling mixed real-time and non-real-time tasks with different performance metrics. The work is motivated by the need to provide satisfactory performance trade-offs in a dynamic environment where the arrival rates and proportions of the real-time and non-real-time tasks vary with time. We first examine two threshold-based schemes, Queue Length Threshold and Minimum Laxity Threshold, and propose the corresponding adaptive schemes based on our results from approximate analysis and simulation. The idea is to improve performance by adjusting trade-off points adaptively as the arrival rates change. We further discuss the idea of integrating the two thresholds. The new algorithm, ADP, is evaluated by simulation under various load conditions and compared with other common scheduling disciplines as well as an optimal algorithm. Some implementation issues are also discussed. We conclude that by setting appropriate threshold functions in accordance to the requirements of applications, we can achieve satisfactory bounded loss ratio for real-time tasks and acceptably low average delay for non-real-time tasks in a wide range of workload conditions.

Acknowledgements

I would like to thank Dr. Samuel T. Chanson, my thesis supervisor, for his guidance and encouragement throughout my work on this thesis. The discussion between us improved the content and presentation of this thesis significantly. I would also like to thank Dr. Norman Hutchinson for reading through the draft of this thesis, his comments and his help in system kernels. Many thanks go to the graduate students of the Department of Computer Science, especially Ann Lo, Luping Liang, Ying Zhang and Francis Liang. I thank Scott Ralph and George Phillips for their help in PostScript.

This work was partly supported by a University Graduate Fellowship from the University of British Columbia.

Finally, I wish to thank my parents and my lovely Wei. They have been an endless source of encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This research is motivated by the scheduling problems we encounter in designing an operating system kernel to support real-time communications. The particular applications that this kernel is going to support are distributed multi-media processing in a general purpose timesharing environment, such as voice and digital audio/video, which requires both time-critical and non-time-critical computation and communication. In this chapter, we first describe the problem, then briefly review related work in this area, and finally close this chapter with a thesis overview.

1.1 The Problem and Motivations

This thesis is motivated by the performance problems encountered in supporting both *time sensitive* and other usual activities in a general purpose computing environment. We are interested not only in satisfying the time constraints of the real-time jobs but also in ensuring the overall system performance, especially minimizing the average delay or response time of the regular tasks.

1.1.1 The Problem

Most real-time system research focuses mainly on meeting *strict* time constraints in a *predictable* way [32]. Typical examples of such applications include process control plants, aircraft and space shuttle control systems and robotics. However, with the advent of the new generation of workstations and high-speed networks providing diverse integrated services, more and more *time* *sensitive* applications are run on general purpose systems. This brings increasing importance to studying scheduling policies for computer and communication systems which process/transmit both real-time and non-real-time tasks/data.

A system is called a *real-time system* if its correctness depends on not only its logical output but also the time when the output value becomes available. A real-time task can be characterized by a few time parameters. *Arrival time* of a real-time task is the time when it becomes available to be scheduled for service. Either a *deadline* or *laxity* can be used to express the time urgency of a task. *Deadline* is the latest time a task should be completed before it is considered *lost* , while *laxity* is the time interval from the current time to the latest time a task should be started if it is to meet its deadline. The initial laxity of a task is the time interval from its arrival time to the latest time it should be started. The laxity of a real-time task deceases as the time passes. A task is lost when its laxity becomes negative. A *lost* real-time task may be useless or of little use. In the former case, it can be simply discarded.

The main objective of real-time system is *predicatability* of its temporal behavior, i.e., to ensure that the system will meet the time requirements of its specification, or to ensure it will do so with a *bounded* high probability. The most important technique to achieve timing predicatability is *real-time scheduling* , i.e., assigning system resources to processes taking their timing constraints into account. Two most important performance metrics of real-time system are *loss ratio* and *guarantee ratio* :

$$loss\ ratio = \frac{number\ of\ lost\ jobs}{number\ of\ total\ arrivals}$$

$$guarantee\ ratio = \frac{number\ of\ accepted\ jobs}{number\ of\ acceptable\ jobs}$$

A real-time scheduling algorithm is often called *optimal* in two senses. One is if its guarantee ratio is 1, i.e., it can find a feasible schedule whenever such schedule exists. Algorithms like *rate monotonic* [1] [19], *earliest deadline first* (EDF) [21] and *minimum laxity first* (ML)

[1]Rate monotonic assigns static priority to periodic processes according to their rates. Processes with higher rate get higher priority. It is optimal only among all static priority schemes.

[21] are shown to be optimal in this sense. However, in designing real-time systems, we are particularly interested in minimizing the long-term loss ratio. Another (stronger) definition of an *optimal algorithm* then requires an algorithm to produce a schedule minimizing the loss ratio in any arrival scenario. We will use the latter criterion in this thesis. A scheduling problem is *nonpreemptive* if resources cannot be taken away temporary from the tasks. It is *preemptive* otherwise. A variation of EDF which drops lost jobs has been shown to be optimal if the jobs require equal service time [31]. More recently, Panwar and Towsley [25] proved that ML is optimal among all *nonpreemptive work-conserving*[2] policies with respect to the fraction of jobs beginning service by their laxity for very general system models. They also obtained similar results for EDF allowing preemption. Numerous works have been published in this area. Other than the exact analysis of simple policies, there exists a growing literature on the design and evaluation of heuristic real-time scheduling policies [32] due to the difficulty in modeling complicated practical systems. There is an obvious assumption in conventional real-time theory that only the performance of real-time tasks is of concern. This is partially because most early real-time applications, included few or no time insensitive tasks.

Unfortunately, this does not hold any more when we try to provide real-time support in general purpose systems. There is a dramatically different view of performance optimality in *general purpose* computing environment where the major metrics are *average response time* (or *average waiting time*), *throughput* , *fairness* and *system utilization* [24, 28, 6]. Algorithms such as *Round-Robin, Shortest Job First* and *Shortest Remaining Time First* have been developed to achieve these objectives [6]. But timesharing techniques obviously cannot provide satisfactory real-time support. The two paradigms, *real-time* and *timesharing,* are motivated and developed for different types of applications. It is not easy to bring them together.

Therefore we face a serious problem in incorporating real-time support to timesharing systems. With the emergence of new applications such as multi-media systems, there have been increasing interests for scheduling algorithms to provide good performance for both general and time critical tasks. In such systems, the performance of time sensitive jobs should not be

[2]In a work-conserving policy, a server is never idle when there is a task to process.

achieved by a serious deterioration of the other tasks. A system using either timesharing techniques only or real-time scheduling policies only cannot achieve this. Another problem arises if real-time and non-real-time tasks are semantically related. Poor non-real-time task performance will bring the overall performance down. On the other hand, a scheduler can multiplex the processor between real-time and non-real-time tasks by taking both types of performance metrics into consideration. Instead of completely biasing toward real-time tasks, we can defer real-time tasks in favor of non-real-time activities until the real-time tasks become really urgent. If the real-time tasks have relatively large laxity, we may expect to significantly improve the performance of non-real-time tasks without much loss of real-time tasks. It is also observed that many applications (e.g., digital video) can tolerate certain degree of loss without noticeable performance deterioration. Therefore a scheduler may take advantage of this property to gain overall system performance by using the above strategy. Meanwhile, this multiplexing should be efficient, able to adapt to dynamic environments and inexpensive to implement.

1.1.2 Goals

This thesis presents our development, analysis and evaluation of an adaptive scheduling method which is to provide an effective solution to the above problem. Among the works addressing the design and analysis of scheduling policies for real-time systems, there are two types of approaches: one tends to analyze the exact behavior of some simple policies, while the other tries to develop heuristic strategies and implementation methods and evaluate them through simulation or benchmarks. In our study, analytic, numerical and simulation methods are used to analyze and demonstrate the performance of the various algorithms.

We restrict our attention to two main performance metrics:

- *the loss ratio for real-time jobs* , and
- *the average waiting time for non-real-time jobs*

We have the following goals in mind:

1. To provide real-time jobs some degree of guarantee on maximum loss ratio.

2. To minimize non-real-time performance penalty while achieving (1).
3. Efficient and easy implementation.
4. Stable system behavior in a dynamic environment. In particular, real-time task performance should not be undermined by temporary overall system overrun, or by heavy non-real-time load. The system should also behave normally under an unbalanced work load.
5. To achieve the best performance trade-off between real-time and non-real-time tasks. Further more, this trade-off decision should be explicitly available to the system designer and should be easy to control.

1.2 Related Work

Numerous results on scheduling theory have been published in the early days by the researchers in areas such as operation research, automatic control, combinatorial mathematics and computer science. General problems of scheduling jobs with arbitrary arrivals, varied processing time and laxity (or deadline) to minimize loss ratio or optimize other performance metrics are well-known to be *NP-complete* in the strong sense. However, some restrictions on the problem parameters may often transform them into *pseudopolynomial* or even *polynomial-bounded* problems. We refer readers to [8, 18, 10, 17] for surveys on these classic works.

The development of real-time computer systems has brought new interest to scheduling algorithms that optimalize certain system metrics. In Panwar and Towsley's work [25], the *Minimum laxity first* (ML) policy has been shown to be optimal among all nonpreemptive work preserving policies for the $G/M/c+G$ system[3] where the time constraint is laxity. Similar results are also obtained for the *earliest deadline first* (EDF) policy in $G/M/1+G$ systems when the time constraint is deadline. Hong, Tan and Towsley [12, 13] recently analyzed the

[3]We use Kendall's notation $A/B/m + L + S$ in this thesis, where A describes the inter-arrival process, B describes the service time requirement, and m is the number of servers. L and S represent laxity distribution and scheduling policy respectively.

performance bound of $M/M/1+M$ system when jobs have deadlines for both ML and EDF. They also proposed a class of $ML(n)$ and $EDF(n)$ algorithms for performance bound analysis and more efficient implementation. $ML(n)$ schedules the first n jobs according to the minimum laxity first discipline and leaves the remaining jobs in a secondary queue from where they will be fed to the main queue in a FCFS manner. $EDF(n)$ works similarly. It is observed that $ML(n)$ and $EDF(n)$ provide reasonably good approximations of the original algorithms even when n is very small. Interestingly enough, a similar idea also appears in [35], where a variant of FCFS (called IFCFS) is proposed and is later proved to have identical behavior to $ML(2)$ [23]. Further studies of $ML(n)$ follow in [23, 22, 9]. $ML(n)$ can also model systems with limited resources (e.g., buffers) where overloaded jobs can be left in a secondary storage and be fed into main memory in FCFS order [27]. Among other works, Liu et al studied scheduling problems with various temporal constraints and problems with imprecise computations [14, 3, 30].

Recently, with the emergence of integrated service packet-switched networks and high-performance workstations, more work has been done in scheduling multiple classes of jobs with diverse performance objectives. In [5], Chipalkatti, Kurose and Towsley proposed two algorithms based on thresholds. They considered a two-queue system, one for real-time jobs and another for non-real-time jobs. The *Queue Length Threshold* (QLT) policy employs a threshold for the non-real-time queue length. Priority is given to real-time jobs unless the non-real-time queue length exceeds the threshold which is kept constant. On the other hand, *Minimum Laxity Threshold* (MLT) uses a laxity threshold so that non-real-time jobs will be served until the minimum laxity (remaining time before deadline) is below the constant threshold. This is an example of the general heuristic that gives real-time jobs priority over non-real-time jobs only when their laxities have become small. Their analytic modeling and numerical results show that MLT and QLT have little difference in the performance tradeoffs and consequently they concluded that QLT was more practical because queue length is easier to monitor. However, their study did not provide explicit indication of how the thresholds were related to the achievable performance. More importantly, QLT and MLT cannot provide satisfactory performance for systems with dynamic work loads. We further study these two heuristics in Section 2.1. Peha and Tobagi [26, 27] studied various scheduling algorithms handling multiple classes of network packets. They proposed a cost-based policy which employs a *cost function* to express different performance objectives. These efforts follow the above common idea of holding real-time tasks until they become urgent. But they did not consider how a scheduler could adapt to a dynamic unbalanced environment which is very likely in reality, nor did they deal with issues such as stability and implementation efficiency which will be addressed in this thesis.

1.3 Overview

The rest of the thesis is organized as follows:

Chapter 2 describes our further study on the two threshold-based scheduling policies, QLT and MLT. We first capture the major characteristics of the two algorithms by simulation, and then study the relation between system loads, threshold values and system performance. Approximate analysis is presented along with brief discussions.

Chapter 3 explores the idea of adaptive scheduling based on threshold functions to overcome the problems in static QLT and MLT. Two adaptive schemes are proposed and compared with their static counterparts. We also propose integrating the two thresholds and address implementation related problems.

Chapter 4 presents experimental simulation and performance evaluation of our adaptive scheme (ADP) by comparing its various performance metrics to a group of well studied algorithms as well as a proven optimal algorithm. Experiments are carried under various system conditions and job arrival patterns. Our simulation results show that ADP meets our design goals very well.

Chapter 5 concludes this work and looks into future studies.

Chapter 2

Analysis of QLT and MLT Scheduling

In this chapter, we first outline our study on two static threshold-based scheduling policies, QLT and MLT, in section 2.1. This study led us to explore the idea of adaptive threshold functions. Section 2.2 presents our approximate analysis of the dynamic behavior of the two threshold-based algorithms. The analysis is aimed at explaining our observations and providing more insight into the fundamental problems.

2.1 Static Threshold-based Algorithms

In [5], two threshold-based static scheduling algorithms are proposed for scheduling mixed real-time and non-real-time tasks. Assuming geometrical distribution for real-time arrivals, a general form of transition matrices for the corresponding discrete time Markov chains is obtained from which a set of equations can be derived by making further assumptions. Though this gives us the possibility to numerically study the two algorithms, it does not provide explicit information on the relation between the system parameters.

The two policies work as follows:

- **Minimum Laxity Threshold:** In MLT, the real-time task with minimum laxity is scheduled if this laxity is below the constant threshold, T_p; or when the non-real-time queue is empty.

- **Queue Length Threshold:** In QLT, the first non-real-time task in the waiting queue is scheduled if the non-real-time queue length exceeds the constant threshold, T_q; or when the real-time queue is empty.

Figure 2.1 describes the basic model of these two policies. We have a two-queue, one server system. Real-time and non-real-time tasks arrive in a Poisson process with rates λ_r and λ_n respectively. Their service times are exponentially distributed with average $1/\mu$. Each real-time task has a laxity π. We're interested in real-time task loss ratio ϵ (the percentage of total lost tasks over total arrivals) and average non-real-time task delay $\mathcal{E}$ (including waiting time and service time). T_q is the non-real-time queue length threshold associated with QLT, and T_p is the minimum laxity threshold associated with MLT. The scheduler multiplexes the server between the two queues according one of the above two disciplines. Note that there are still free choices for scheduling policies within the individual queues. However, we will use FCFS with the non-real-time queue and an optimal policy ML with the real-time queue throughout our discussion.

Chipalkatti et al [5] studied QLT and MLT under strong assumptions including balanced real-time and non-real-time arrival rates (i.e., constant and equal), constant laxity upon arrival, and normal work load (uncongested). They concluded that QLT and MLT offer similar tradeoffs between the delay of the non-real-time tasks and the loss of the real-time tasks under the above assumptions. However, because of the bursty nature of real applications, we are interested in the performance of algorithm under unbalanced loads, with divergent laxities and in congested peak periods, as well as the effectiveness of the threshold parameters in achieving certain performance objectives. A dynamic environment and the resulting state space explosion make their approach prohibitive. This is mainly due to the difficulty in achieving acceptable accuracy over a large range of system configurations, and high computational complexity. Another inherent problem of analytic and numerical approaches is their inability to describe unstable systems. Therefore the following discussion will be based on simulation results.[1]

[1] However, we do observe that our simulation results match the numerical values well in those cases when the number of states is moderate (e.g. Figures 2.10 and 2.11).

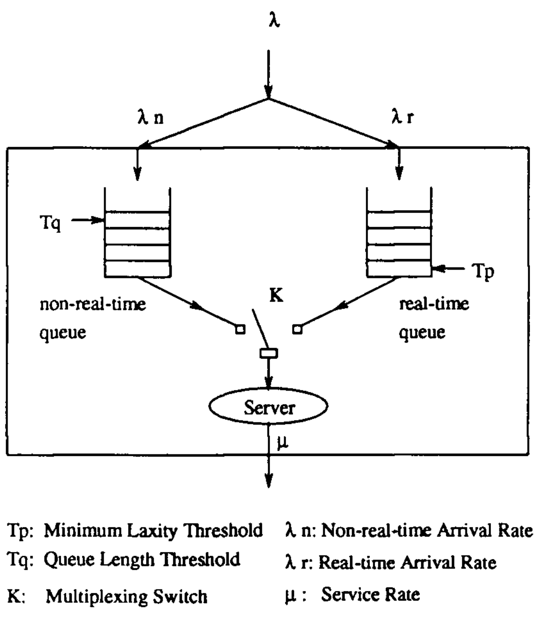
λ
λ n
λ r
Tq
Tp
non-real-time queue
real-time queue
K
Server
μ
Tp: Minimum Laxity Threshold
Tq: Queue Length Threshold
K: Multiplexing Switch
λ n: Non-real-time Arrival Rate
λ r: Real-time Arrival Rate
μ : Service Rate

Figure 2.1: Static Threshold-based Model

Figures 2.2 to 2.9 illustrate the behavior of *static* QLT and MLT policies when the arrival rate of only one type of tasks (either real-time (RT) or non-real-time (NRT)) changes. Figures 2.10[2] and 2.11 show the effect of the queue length threshold for a given work load with $\lambda_r = 0.2$ and $\lambda_n = 0.8$.

We have the following observations:

1. **Static QLT biases towards non-real-time tasks.**

   Figures 2.2 and 2.3 show that, with a fixed λ_n, loss ratio ϵ starts to grow rapidly when the overall work-load (*not real-time work-load !*) approaches saturation point. The delay of non-real-time tasks $\mathcal{E}$, however, tends to remain stable at a value determined by T_q and λ_n. Figures 2.4 and 2.5 provide even stronger evidence. Even with a constant λ_r, the loss ratio grows linearly (with a large slope) as λ_n increases. However, static QLT keeps $\mathcal{E}$ moderate over a large region until λ_n itself (*not the overall work-load !*) approaches 1. Tuning the

[2]The deviation of numerical result from simulation value is caused by accumulated inaccuracy of the program.

threshold T_q changes the relative values, but does not change the above property. Steep curves of loss ratio imply the difficulty in obtaining a T_q value so that the loss ratio can be kept below a reasonably small bound in a dynamic environment. This indicates a most undesirable property that the load of the non-real-time tasks has very strong influence on the performance of the real-time tasks.

2. **Static MLT biases towards real-time tasks.**

   Static MLT has the opposite behavior. Figures 2.7 and 2.9 show a sudden hike of $\mathcal{E}$ when the total work-load approaches the saturation point. A nice property of MLT is that the loss ratio is up-bounded for a given threshold value. The dotted curve in figure 2.6 shows this upper-bound for constant λ_n, and Figure 2.8 shows how ϵ reaches an upper-bound as λ_n increases. However, the problem is that the loss ratio curve grows very fast. This implies that, to ensure low loss ratio in heavy load, T_q has to be raised very high. A high threshold value, in turn, will unnecessarily degrade non-real-time task performance when the load is light. Therefore it is difficult to select a proper threshold to ensure peak time loss ratio without loss of performance under normal conditions.

3. **Static QLT and MLT will perform worse with non-uniform laxity and service time.**

   Minimum laxity is used as an indication of urgency for real-time tasks. It estimates the real situation less accurately when the laxity and service time are non-uniform. The length of the non-real-time queue is used as an indication of the accumulated non-real-time work load. It becomes a poor indication when the service time is non-uniform. Thus both QLT and MLT will not work well when the laxities and service times of tasks are not constant and identical.

4. **QLT and MLT are effective means to adjust performance tradeoffs for fixed input rates, but not for dynamic arrivals.**

   Figures 2.10 and 2.11 show that QLT's effect on non-real-time average delay is, as expected, almost linear. The loss ratio of real-time tasks, however, drops dramatically in the beginning, and then stays at a fairly constant level. Figures 2.10 and 2.11 also suggest that we should look at the *elbow* of the loss ratio curve, where the loss ratio is low with a relatively small penalty in average waiting-time. Unfortunately, this point is a function of both λ_r and λ_n and is not fixed in a dynamic environment. MLT is more effective in controlling the loss ratio, but increases average delay much more dramatically than QLT. We observe a very steep curve for both loss ratio and average delay. This is not good as it offers more *coarse-grained* tradeoff choices to the system designer. It means it is difficult to obtain a proper tradeoff point even for fixed arrival rates, let alone in a dynamic environment.

All of these observations suggest the need of some kind of dynamic scheme to adapt to the changing environment, especially the changing work loads and the changing proportion of real-time and non-real-time tasks. In other words, the threshold value should be a function of the dynamic environment. The performance problem is particularly serious in the *overloaded* periods, when certain discrimination for the different classes of tasks is essential. But consistent discrimination will deteriorate the performance of low priority tasks unnecessarily under moderate system load. This leads us to examine the idea of adaptive schemes. But before that, we will first analyze the dynamic behavior of the two static threshold policies more formally to gain more insights into the problems.

2.2 Approximate Analysis of QLT and MLT

In [5], Chipalkatti, Kurose and Towsley described a discrete time Markov chain model of both QLT and MLT. They used a combined state (x_1, x_2) to describe their system, where x_1 denotes the number of non-real-time tasks in the system, and x_2 denotes the minimum laxity among the queued real-time tasks. Though their combinatorial approach is accurate in modeling the system, it inevitably leads to unmanageably large state transition matrices in practical systems. In fact, even to solve the system numerically is not easy, especially when the load is high (because

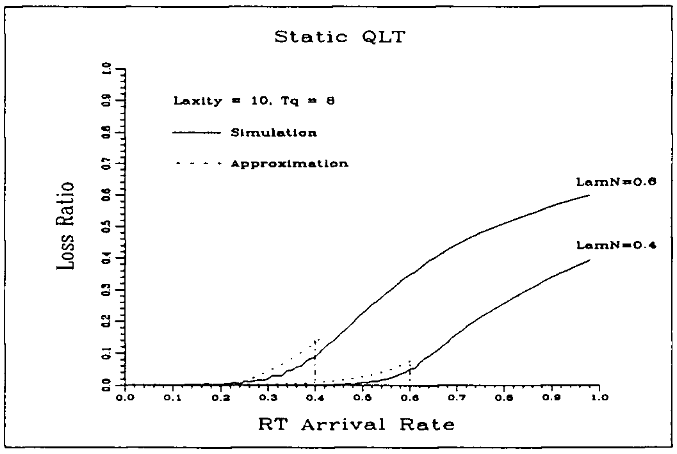

Figure 2.2: QLT Loss Ratio: with Varied RT Arrival Rates

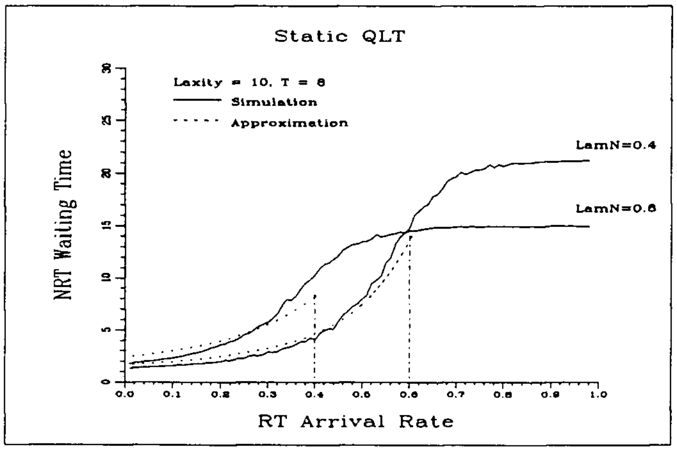

Figure 2.3: QLT Average Delay: with Varied RT Arrival Rates

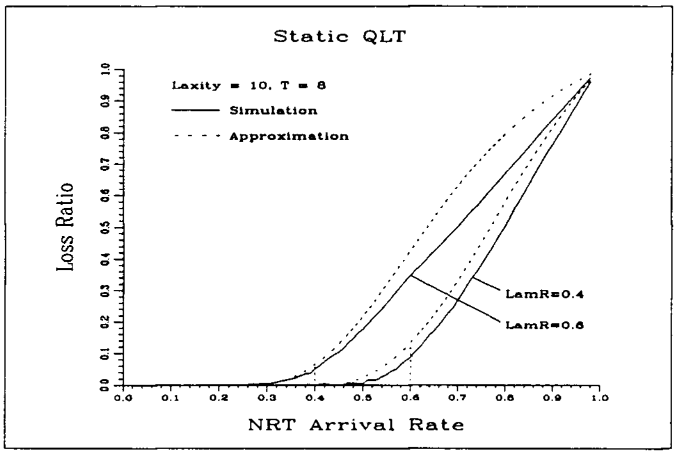

Figure 2.4: QLT Loss Ratio: with Varied NRT Arrival Rates

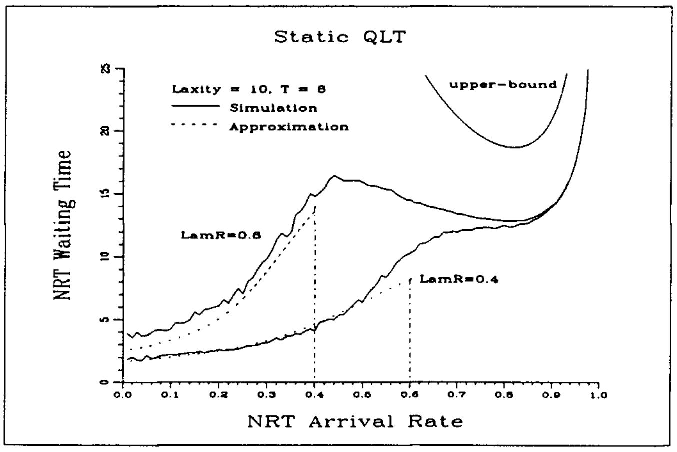

Figure 2.5: QLT Average Delay: with Varied NRT Arrival Rates

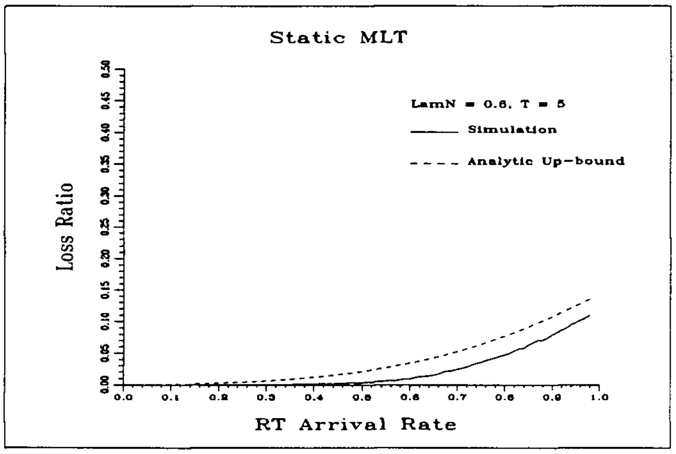

Figure 2.6: MLT Loss Ratio: with Varied RT Arrival Rates

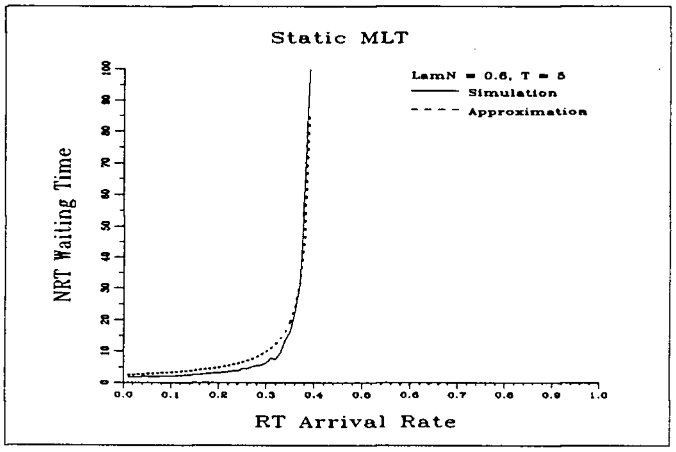

Figure 2.7: MLT Average Delay: with Varied RT Arrival Rates

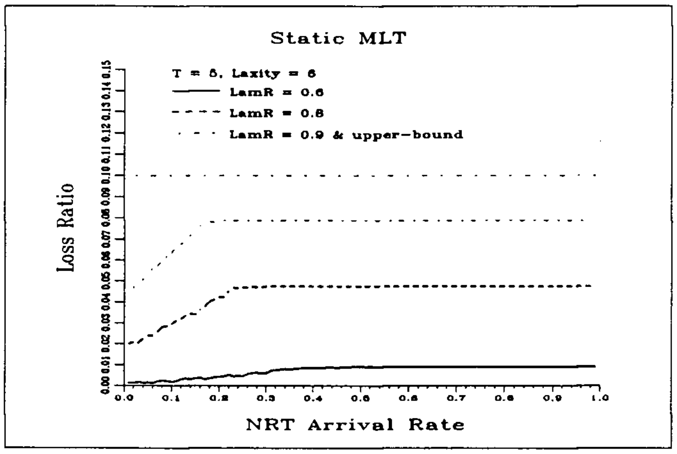

Figure 2.8: MLT Loss Ratio: with Varied NRT Arrival Rates

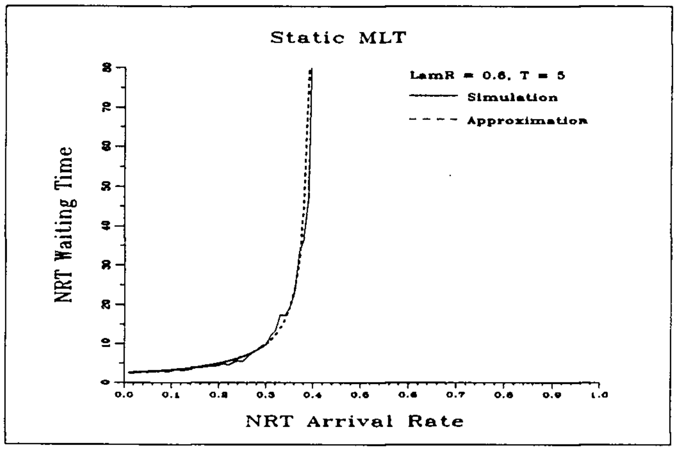

Figure 2.9: MLT Average Delay: with Varied NRT Arrival Rates

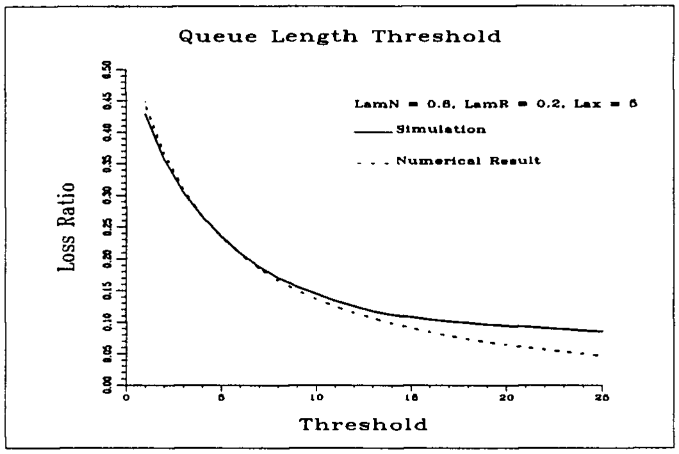

Figure 2.10: Loss Ratio: Effect of Queue Length Threshold

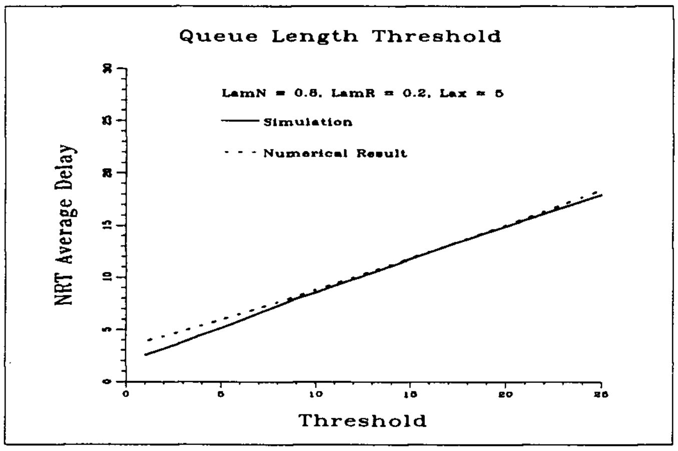

Figure 2.11: Average Delay: Effect of Queue Length Threshold

of limited memory space) and when the load is low (because of cutting errors)[3]. With only a set of general equations without a closed-form solution, their result gives little information on the dynamic behavior of the two policies. Therefore, we developed a simple approximate approach in order to describe general behavior of the two threshold algorithms and explain the observations given earlier. In the following two sections, we derive analytic bounds and approximations for QLT and MLT.

2.2.1 Approximate Analysis of QLT

Recall that in figure 2.1 QLT works as follows:

QLT: schedule a real-time task unless

1. *NRT queue length* $> T_q$; or
2. *RT queue length* $= 0$.

Since we have two queues in the system, a complete description of the system state requires two state variables. Such an approach will inevitably lead to state explosion. So, we decided to separate the two queues as far as we can to avoid the complexity introduced by the correlation between them.

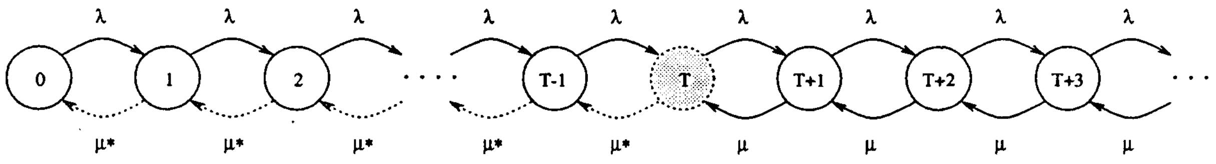

Figure 2.12: QLT: Non-Real-Time state transition rate diagram ($\lambda = \lambda_n$)

Let us first look at the non-real-time queue in QLT [figure 2.12]. By using queue length as the state variable, the non-real-time queue can be described in two stages. When its queue

[3][5] described a solution using matrix geometric methods. However, without special treatment, the accuracy of the solution is limited by space and time. As shown in some figures, our straightforward program has difficulty in producing reasonable results in the cases when the states are large.

length exceeds the threshold T_q non-real-time tasks will always be served, so it is simply an $M/M/1$ system. However, when its queue length is below T_q, a non-real-time task (if any) can be served only if the real-time queue is empty. We make the assumption that its departure process is also *Markovian* when the queue length is below the threshold. We can imagine that the non-real-time queue has its own but *slower* server, so that the service time is scaled *larger* with mean $1/\mu_*$. Let $p_0^r = Pr\{realtime\ \ queue\ \ is\ \ empty\}$, we have:

$$\mu_* = \mu p_0^r \tag{2.1}$$

Let us also denote by ρ_n the ratio λ_n/μ , and by ρ_* the ratio λ_n/μ_*[4]. We have the following *equilibrium equations* :

$$\begin{aligned}
&\lambda_n p_0 = \mu_* p_1 \\
&\lambda_n p_{N-1} + \mu_* p_{N+1} = (\lambda_n + \mu_*) p_N, \qquad (1 \le N \le T_q) \\
&\lambda_n p_{T_q-1} + \mu p_{T_q+1} = (\lambda_n + \mu_*) p_{T_q} \\
&\lambda_n p_{N-1} + \mu p_{N+1} = (\lambda_n + \mu) p_N, \qquad (N > T_q) \\
&\sum_{N=0}^{\infty} p_N = 1
\end{aligned} \tag{2.2}$$

where p_N is the *equilibrium probability* in state N. The solutions to the above equations are :

$$p_N = p_*^N p_0, \qquad (0 \le N \le T_q) \tag{2.3}$$

$$p_{T_q+k} = \rho_n^k p_{T_q} = \rho_n^k \rho_*^{T_q} p_0, \qquad (k \ge 1) \tag{2.4}$$

Substituting them into (2.2), we get

$$p_0 = \left(\frac{1 - \rho_*^{T_q}}{1 - \rho_*} + \frac{\rho_n}{1 - \rho_n} \rho_*^{T_q} \right)^{-1} \tag{2.5}$$

Therefore, the mean queue length is

$$\overline{N_*} = \sum_{n=0}^{\infty} n p_n = \sum_{n=0}^{T_q} n p_n + \sum_{T_q+1}^{\infty} n p_n$$

[4] Though $\rho = \lambda/\mu$ is used as *traffic intensity* in queueing theory, there is no such meaning attached to ρ_*. It is only used to simplify the formula.

$$\begin{aligned}
&= \sum_{n=0}^{T_q} n\rho_*^n p_0 + \sum_{n=T_q+1}^{\infty} n\rho_n^{n-T_q}\rho_*^{T_q} p_0 \\
&= p_0\left(\frac{\rho_*(1-\rho_*^{T_q}) - T_q\rho_*^{T_q+1}(1-\rho_*)}{(1-\rho_*)^2} + \frac{T_q\rho_n(1-\rho_n)+\rho_n}{(1-\rho_n)^2}\rho_*^{T_q}\right) \qquad (2.6)
\end{aligned}$$

This expression seems awkward and complex. However, assuming there is extremely high or low real-time work load, or equivalently letting $\mu_* \to 0^+$ or $\mu_* \to \mu$ respectively, we get the following bounds.

Theorem 1 *The average delay of non-real-time tasks in QLT is bounded by*

$$\frac{1}{\mu(1-\rho_n)} \le \mathcal{E} \le \frac{1}{\lambda_n}\left(\frac{1}{\rho_n}T_q + \frac{1}{1-\rho_n}\right) \qquad (2.7)$$

Proof: To prove the right part, as we discussed at the beginning, we will give our system a *slower* server whenever its queue length is lower than T_q. We will slow it down so that the average queue length in the new system is no less than the original system.

More explicitly, for any real-time arrival rate, there exists a scale ratio $1/\eta$ $(0 < \eta \le 1)$ such that, if $\mu_* \le \eta\mu$, $\overline{N_*} \ge \overline{N}$.

Using (2.6) and letting $\eta \to 0^+$, we obtain

$$\begin{aligned}
\overline{N} &\le \lim_{\eta\to 0^+}\overline{N_*} = \lim_{\rho_*\to+\infty}\overline{N_*} \\
&= \frac{1}{\rho_n}T_q + \frac{1}{1-\rho_n} \qquad (2.8)
\end{aligned}$$

Applying *Little's Law* , we get the right half. The left half can be derived simply by letting $\eta = 1$. □

Discussions:

1. If $\mu = 1$, the difference between our upper-bound and lower-bound is T_q/λ_n. So, as λ_n approaches 1, our bounds estimate the actual value very well. Meanwhile, with smaller threshold T_q, the difference decreases. In fact, when $T_q \to 0$, our system becomes $M/M/1$.

2. More interestingly, (2.8) can be rewritten as

$$\frac{1}{\rho_n}\left(T_q + \frac{\rho_n}{(1-\rho_n)}\right) \qquad (2.9)$$

Consider ρ_n large (close to 1), the first item is the threshold T_q, and the second item is the mean queue length in an $M/M/1$ queue. This indicates that under heavy load, the non-real-time waiting queue can be viewed as consisting of two pieces. A queue of T_q tasks followed by an $M/M/1$ queueing system. Interpreting (2.7) in the same way, we rewrite (2.7) as

$$\frac{1}{\rho_n}\left(\frac{1}{\lambda_n}T_q + \frac{1}{\mu(1-\rho_n)}\right) \tag{2.10}$$

It indicates that under heavy load the waiting time for non-real-time tasks consists of two periods, one to wait for T_q tasks to arrive ($\frac{1}{\lambda_n}T_q$) and the other to receive service from an $M/M/1$ system ($\frac{1}{\mu(1-\rho_n)}$). This agrees with our intuition.

3. Let $\mu = 1$ and $\rho_n = \frac{\lambda_n}{\mu} = \lambda_n$, (2.8) becomes

$$\frac{1}{1-\lambda_n} \leq N_* \leq \frac{T_q}{\lambda_n} + \frac{1}{1-\lambda_n} \tag{2.11}$$

Consider the system as one consisting of two subsystems: the first is a T_q long waiting device and the second is an $M/M/1$ service center. Our rough bounds represent the uncertainty of the delay time in the first device. If a task can pass through it immediately we get the lower bound. In the worst case when a task has to wait for T_q incoming tasks to push itself through, we get the upper-bound. This is of course a rough approximation. The time that a task has to spend in the waiting device is determined by the behavior of the real-time queue. We will take this into consideration below.

The inaccuracy resides on the extreme values we take in evaluating (2.6). We can use a better estimation for η other than 0 or 1. One reasonable approximation is

$$\begin{aligned} \eta &= p_0^r = Pr\{realtime\ queue\ is\ empty\} \\ &\approx 1 - \rho_r = 1 - \frac{\lambda_r}{\mu} \end{aligned} \tag{2.12}$$

Substituting $\rho_* = \frac{\lambda_n}{\eta\mu} = \frac{\lambda_n}{\mu - \lambda_r}$ into (2.6), we expect to achieve a better estimation. The result is especially good when λ_n is small, because the length of the non-real-time queue is unlikely to exceed T_q to interrupt real-time processing. This makes (2.12) accurate. We show this in figure 2.3 and figure 2.5. In fact, when λ_n is small (and real-time work load is not too high) the chance for the non-real-time queue length to exceed T_q is very small, then we may even use the following simple approximation:

$$\mathcal{E} \approx \frac{1}{\mu_*}(1-\rho_*) = \frac{1}{\mu - \lambda_r - \lambda_n} = \frac{1}{\mu - \lambda} \tag{2.13}$$

where $\lambda = \lambda_r + \lambda_n$, is the overall arrival rate. This indicates that non-real-time tasks have to yield to real-time tasks.

Therefore our conclusion regarding the average delay of the non-real-time tasks is as follows. For light work load and small λ_n, QLT's behavior is similar to $M/M/1$. For heavy work load and large λ_r, QLT first makes the non-real-time tasks wait T_q units of time and then treats them as in a separate $M/M/1$ system without real-time work load. For the cases in between, $\mathcal{E}$ will be moderate and determined by T_q, λ_n and λ_r.

We now turn to study the real-time loss ratio in QLT. Noticing the real-time queue is approximately an $M/M/1+D+ML$ system, we can simply use Hong's result in [13] to estimate the loss ratio:

$$\epsilon = 1 - \frac{(1-p_0^r)\mu}{\lambda_r} \tag{2.14}$$

To get better result, let γ_r be the throughput of real-time tasks, and $P_{\leq T_q} = \sum_{n=0}^{T_q} p_n$, we have

$$\begin{aligned} \gamma_r &= \mu Pr\{realtime\ queue\ is\ not\ empty\ and\ NRT\ queue\ length\ \leq T_q\} \\ &= \mu(1-p_0^r)P_{\leq T_q} \end{aligned} \tag{2.15}$$

Then the loss ratio is

$$\epsilon = \frac{\lambda_r - \gamma_r}{\lambda_r} = 1 - \frac{\mu(1-p_0^r)P_{\leq T_q}}{\lambda_r} \tag{2.16}$$

(2.14) is the special case when $P_{\leq T_q} = 1$. Substituting the p_n's , we have the following result:

$$\epsilon \approx 1 - \frac{1-p_0^r}{\lambda_r}\left(1 + \frac{\rho_n}{1-\rho_n}\left(1 - \frac{1}{p_0^r}\rho_n\right)\frac{\left(\frac{1}{p_0^r}\right)^{T_q}\rho_n^{T_q}}{1-\left(\frac{1}{p_0^r}\right)^{T_q}\rho_n^{T_q}}\right)^{-1} \tag{2.17}$$

The factor p_0^r in (2.17) can be approximated depending on the value of λ_n. If λ_n is relatively large, then p_0^r can be estimated using $1 - \frac{\lambda_r + \delta\lambda_n}{\mu}$, where δ is a small factor. If λ_n is very small, $1 - \frac{\lambda_r}{\mu}$ may serve the purpose well. We show this in figure 2.2 and figure 2.4.

We conclude that QLT offers little protection to the real-time tasks during periods of high work load. Though rather complicated, ϵ is strongly dependent on the non-real-time load. Actually from (2.17), we have $\lim_{\lambda_n \to 1} \epsilon = 1$.

2.2.2 Approximate Analysis of MLT

Recall from figure 2.1 that MLT works as follows:
MLT: schedule a non-real-time task unless

1. *minimum laxity* $\leq T_p$; or

2. *NRT queue length* $= 0$.

Again, the complexity is introduced by the correlation between the two queues. However, MLT policy does not use any non-real-time queue information except whether it is empty or not. To reduce the complexity and obtain a clean solution, we make the following assumption:
Assumption: *For large* λ_n*, Pr{ NRT queue is empty } is negligible in MLT.*
Note that based on this assumption, the loss ratio we may obtain is an upper-bound of the actual value, because we actually give up some chances when real-time tasks may be served. In other words, as far as real-time tasks are concerned, the server can be considered unavailable when all real-time tasks have laxities larger than T_p.
So, our version of MLT is equivalent to: [figure 2.13]

- if task t_i arrives at time r_i with initial laxity π_i, then the execution time e_i must be:

$$r_i + \pi_i - T_p \leq e_i \leq r_i + \pi_i, \quad \pi_i \geq T_p$$

We further assume all tasks have identical initial laxity:

$$\pi_i = \pi, \quad 0 \leq i < \infty$$

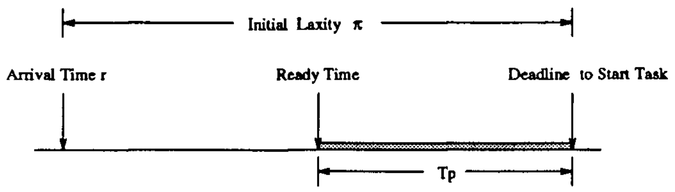

Figure 2.13: Model $\mathcal{M}_1$

We will call this model system $\mathcal{M}_1$. Now we describe another system $\mathcal{M}_2$ as follows:

- $\mathcal{M}_2$ is same as $\mathcal{M}_1$, except:

  1. task t_i arrives at time $r_i' = r_i + \pi - T_p$;
  2. task t_i has initial laxity $\pi' = T_p$;
  3. tasks are scheduled FCFS.

We have the following result:

Lemma 1 *System $\mathcal{M}_1$ and $\mathcal{M}_2$ are equivalent with regard to the fraction of lost tasks:*[5]

$$\epsilon_1 = \epsilon_2$$

Proof: We prove the lemma in two steps:

1. It is obvious that the FCFS scheduling policy in $\mathcal{M}_2$ is equivalent to ML, because all tasks have identical initial laxity value π. At any point in time, a task that has arrived earlier has a smaller laxity then those that come after it. Therefore, it is sufficient to prove that $\mathcal{M}_1$ is equivalent to a system, say $\mathcal{M}_2'$, which is the same as $\mathcal{M}_2$ but scheduled in ML instead of FCFS.

[5]We actually obtain a stronger result in the proof that follows.

2. We achieve this by proving $\mathcal{M}_2'$ *almost* exactly simulates $\mathcal{M}_1$:

   Suppose we purposely start the $\mathcal{M}_2'$ server $(\pi - T_p)$ later than that of $\mathcal{M}_1$. Then, at instant $(\pi - T_p)$, $\mathcal{M}_1$ and $\mathcal{M}_2'$ will have identical tasks in their waiting queues and none has processed any task. We claim that $\mathcal{M}_1$ and $\mathcal{M}_2'$ will have completely identical states. We show this by considering $\mathcal{M}_2'$ as a pipeline of two systems. The first (i.e., left) is just a delay device with identical input and output, while its right is $\mathcal{M}_1$. Furthermore, the first two conditions of $\mathcal{M}_2$ ensure that the output of the left part is the same as the input of $\mathcal{M}_1$.

   Since the initial state has no influence on the final stationary state, we conclude that, when $t \to \infty$, $\mathcal{M}_1$ has identical stationary state as $\mathcal{M}_2'$. Thus $\mathcal{M}_1$ and $\mathcal{M}_2$ have equivalent loss ratio. □

Based on Lemma 1, we consider an $M/M/1+D+FCFS$ with arrival rate λ and departure rate μ, where D denotes uniform distribution. Let us define $F(\omega, t)$ to be the distribution function of waiting tasks and $F(\omega)$ to be its steady state distribution.

$$F(\omega, t) = Pr\{number\ of\ waiting\ tasks\ at\ time\ t < \omega\}$$

$$F(\omega) = \lim_{t \to \infty} F(\omega, t)$$

Let $f(\omega)$ be the density function of $F(\omega)$. The task loss ratio can be computed by:

$$R = R^+(1 - F(0^+)) \tag{2.18}$$

where $R^+ = Pr\{a\ task\ is\ lost \mid server\ is\ busy\}$ and $F(0^+)$ is the probability that the server is idle.

Following the approaches in [15, 35, 16], we can derive the following equation for $F(\omega, t+\Delta t)$. In general, we have:

$$\begin{aligned} F(\omega, t + \Delta t) &= (1 - \lambda\Delta t)F(\omega + \Delta t, t) + \\ &\quad \lambda\Delta t \int_0^\omega (1 - L(x))B(\omega - x)d_xF(x,t) + \\ &\quad \lambda\Delta t \int_0^\omega L(x)d_xF(x,t) \end{aligned} \tag{2.19}$$

where $L(x)$ and $B(x)$ are the task laxity and service time distributions respectively. In our case, $B(x)$ is exponential and $L(x)$ is a uniform distribution:

$$L(x) = \begin{cases} 0 & \text{if } x \leq T_p \\ 1 & \text{otherwise} \end{cases}$$

On the right hand side of equation (2.19), the first term is for the case when there is no new arrival from time t to $t + \Delta t$. The second term is for the case when there is a new task arrival and the task is scheduled. The last term is for the case when the newly arrived task is lost.

If $0 < \omega \leq T_p$, (2.19) becomes:

$$\begin{aligned} F(\omega, t + \Delta t) &= (1 - \lambda \Delta t) F(\omega + \Delta t, t) + \\ & \lambda \Delta t \int_0^{\omega} B(\omega - x) d_x F(x, t) \end{aligned} \tag{2.20}$$

or,

$$\frac{F(\omega, t + \Delta t) - F(\omega + \Delta t, t)}{\Delta t} = -\lambda \left(F(\omega + \Delta t, t) - \int_0^{\omega} B(\omega - x) d_x F(x, t) \right).$$

Let $\Delta t \to 0$, we get:

$$\frac{\partial F(\omega, t)}{\partial t} - \frac{\partial F(\omega, t)}{\partial \omega} = -\lambda \left(F(\omega, t) - \int_0^{\omega} B(\omega - x) d_x F(x, t) \right)$$

Let $t \to 0$, since $\lim_{t \to \infty} \frac{\partial F(\omega, t)}{\partial \omega} = 0$ and $\lim_{t \to \infty} \frac{\partial F(\omega, t)}{\partial t} = f(\omega)$, we obtain:

$$f(\omega) = -\lambda \left(F(\omega) - \int_0^{\omega} B(\omega - x) f(x) dx \right) \tag{2.21}$$

Considering the fact that $B(x) = 1 - e^{-\mu x}$ and $B'(x) = \mu e^{-\mu x}$, and differentiating both sides of (2.21), we have:

$$\frac{df(\omega)}{d\omega} = (\lambda - \mu) dw \tag{2.22}$$

Therefore the solution is, for $0 < \omega \leq T_p$,

$$f(\omega) = A e^{(\lambda - \mu)\omega} \tag{2.23}$$

where A is a constant to be determined.

When $\omega > T_p$, (2.19) becomes,

$$\begin{aligned} F(\omega, t+\Delta t) &= (1-\lambda\Delta t)F(\omega+\Delta t, t) + \\ &\quad \lambda\Delta t \int_0^{T_p} B(\omega - x) d_x F(x,t) + \\ &\quad \lambda\Delta t \int_{T_p}^{\omega} d_x F(x,t) \end{aligned} \tag{2.24}$$

Following similar steps, we have:

$$f(\omega) = \lambda\left(F(\omega) - \int_0^{T_p} B(\omega - x) f(x)dx - \int_{T_p}^{\omega} f(x)dx\right) \tag{2.25}$$

and,

$$\begin{aligned} \frac{df(\omega)}{d\omega} &= \lambda\left(f(\omega) - \int_0^{T_p} \mu e^{-\mu(\omega-x)} f(x)dx - f(\omega)\right) \\ &= -\mu\lambda \int_0^{T_p} e^{-\mu(\omega-x)} f(x)dx \\ &= -\mu f(\omega) \end{aligned} \tag{2.26}$$

Therefore the solution is, for $\omega > T_p$,

$$f(\omega) = A' e^{-\mu\omega} \tag{2.27}$$

We need three more equations to determine the constants $F(0^+)$, A and A'. The first will be the continuity condition at point $\omega = T_p$:

$$\lim_{\omega\to T_p+} f(\omega) = \lim_{\omega\to T_p-} f(\omega)$$

which leads to:

$$f(\omega) = Ae^{-\mu\omega+\lambda T_p}, \quad \omega > T_p \tag{2.28}$$

The following boundary condition can then be used to solve for A:

$$\begin{aligned} \int_0^{\infty} f(\omega) &= F(0^+) + \int_0^{T_p} f(\omega)d\omega + \int_{T_p}^{\infty} f(\omega)d\omega \\ &= F(0^+) + A\left(\frac{1}{\mu-\lambda} - \frac{\rho}{\mu-\lambda} e^{-(\mu-\lambda)T_p}\right) \\ &= 1 \end{aligned} \tag{2.29}$$

Hence,

$$A = \frac{1 - F(0^+)}{\frac{1}{\mu-\lambda}\left(1 - \rho e^{-(\mu-\lambda)T_p}\right)} \tag{2.30}$$

The third condition is the flow conservation condition:

$$\lambda = \lambda R + \mu(1 - F(0^+)) \tag{2.31}$$

We then get,

$$F(0^+) = \frac{1 - \rho(1 - R^+)}{1 + \rho R^+} \tag{2.32}$$

Now we are able to compute R^+ and R as follows:

$$\begin{aligned} R^+ &= \frac{\int_{0^+}^{\infty} L(x) f(x)dx}{\int_{0^+}^{\infty} f(x)dx} \\ &= \frac{\int_{T_p}^{\infty} f(x)dx}{\int_{0^+}^{T_p} f(x)dx + \int_{T_p}^{\infty} f(x)dx} \\ &= \frac{\frac{1}{\mu} e^{-(\mu-\lambda)T_p}}{\frac{1}{\mu-\lambda} - \frac{\rho}{\mu-\lambda} e^{-(\mu-\lambda)T_p}} \end{aligned} \tag{2.33}$$

Combining (2.18) and (2.32), we have

$$\begin{aligned} R &= \rho \frac{R^+}{1 + \rho R^+} \\ &= \frac{\rho(1-\rho)e^{-(\mu-\lambda)T_p}}{1 - \rho^2 e^{-(\mu-\lambda)T_p}} \end{aligned} \tag{2.34}$$

The above results are summarized in the following theorem.

Theorem 2 *For given minimum laxity threshold T_p, the real-time task loss ratio in $M/M/1 + D + MLT^6$ is upper-bounded by*

$$\epsilon \le R = \frac{\lambda_r(1-\lambda_r)e^{-(1-\lambda_r)T_p}}{1 - {\lambda_r}^2 e^{-(1-\lambda_r)T_p}} \tag{2.35}$$

[6] This is not a common usage of Kendall's notation. It denotes that the real-time queue is an $M/M/1$ queue. All tasks have a constant laxity, and are scheduled by MLT.

Proof: Substitute $\mu = 1$ and $\rho = \lambda_r$ into (2.34). The conclusion immediately follows from Lemma 1 and the above discussion. □

Figure 2.6 shows an example of how this upper-bound fits the actual value.

From (2.35), we can derive the following result. Let $\lambda_r \rightarrow 1^-$,

$$\begin{aligned}\lim_{\lambda_r \rightarrow 1^-} R &= \lim_{\lambda_r \rightarrow 1^-} \frac{\lambda_r(1-\lambda_r)e^{-(1-\lambda_r)T_p}}{1-{\lambda_r}^2 e^{-(1-\lambda_r)T_p}} \\ &= \lim_{\lambda_r \rightarrow 1^-} 1 - \frac{1-\lambda_r e^{(1-\lambda_r)T_p}}{1-{\lambda_r}^2 e^{(1-\lambda_r)T_p}} \\ &= \frac{1}{T_p - 2} \qquad (2.36)\end{aligned}$$

Corollary 1 *The real-time task loss ratio in $M/M/1 + D + MLT$ is bounded by $\frac{1}{T_p-2}$, independent of the arrival rates (when $\lambda_r < 1$).*

This result, though not fitting the actual values very accurately, explicitly gives system designer a rough idea of the worst case performance bound. In figure 2.8, we can see an example of how this estimates the actual bound.

The major shortcoming of this analysis is that we do not take the non-real-time tasks explicitly into account, and the upper-bound fits the true value well only when λ_n is large. This also prevents us from getting any estimates of the non-real-time task performance. To circumvent this, we take a similar approach to that for QLT.

$\mathcal{E}$ is expected to be small when λ_r is small. We are more interested in how $\mathcal{E}$ grows when λ_r is very large. Recall that $F(0^+)$ in $\mathcal{M}_2$ equals the probability that the minimum laxity is below T_p in $\mathcal{M}_1$. Therefore $F(0^+)$ represents the probability that the non-real-time tasks will be served. Following an approach similar to that which we used to analyze QLT, we define $\mu_* = F(0^+)\mu$. The average delay can be estimated by

$$\mathcal{E} \approx \frac{1}{\mu_*(1-\rho_*)} = \frac{1}{F(0^+)\mu - \lambda_n} \qquad (2.37)$$

Combining (2.33) and (2.32), we have

$$F(0^+) = \frac{1-\rho}{1-\rho^2 e^{-(\mu-\lambda)T_p}} \qquad (2.38)$$

then we obtain the following approximation for $\mathcal{E}$.

$$\mathcal{E} \approx \frac{1}{\frac{\mu-\lambda_r}{1-\rho^2 e^{-(\mu-\lambda)T_p}} - \lambda_n} \tag{2.39}$$

Note that when T_p is large, the term $\rho^2 e^{-(\mu-\lambda)T_p}$ approaches 0, so that (2.39) becomes $\frac{1}{(\mu-\lambda_r)-\lambda_n}$. This indicates that non-real-time tasks have to wait until the real-time queue is empty (note the same result in (2.13)).

Before ending this section, we briefly discuss the granularity of control that MLT offers. We first rewrite (2.34) as follows

$$R = \frac{\rho(1-\rho)}{e^{(\mu-\lambda)T_p} - \rho^2}. \tag{2.40}$$

For a given work load, this equation suggests that increasing T_p will exponentially decrease R up to a point. Since T_p can only be integral, this implies that MLT offers relatively coarse-grain control in adjusting the loss ratio. Another limitation of MLT is that the maximum value of T_p will be no larger than the maximum laxity. In the case that all tasks have constant laxity, this limits T_p to be below this constant. For the case when laxity is exponentially distributed, this means that increasing T_p will have little effect on the loss ratio when T_p is large.

2.3 Summary

In this chapter, the performance of two threshold-based algorithms (QLT and MLT) were analyzed. Though the two thresholds achieve roughly similar performance tradeoffs for constant and identical real-time and non-real-time work loads, they are quite different in the dynamic environment. It has been shown that when the work load is high and the arrival rates change, QLT provides bounded average delay (determined by the arrival rate of the non-real-time tasks and the threshold value) for non-real-time tasks at the cost of real-time task loss. On the other hand, MLT ensures bounded loss ratio (independent of the arrival rates) for real-time tasks at the cost of higher average delay for non-real-time tasks under heavy loads. For both QLT and MLT, we have derived analytical bounds and approximate results of the two major performance metrics, i.e., average delay for non-real-time tasks and loss ratio for real-time tasks, for different arrival rates and threshold values. These results were shown to estimate the actual behavior well under most of the interesting conditions.

Chapter 3

Adaptive Schemes

The analysis of the previous chapter shows that the static schemes QLT and MLT cannot provide satisfactory performance in a dynamic environment. This chapter presents our exploration of the idea of adaptive scheduling based on threshold functions. We first describe two adaptive algorithms and examine how they may be integrated with other heuristics in section 3.1. Then, in section 3.2, we discuss implementation issues relating to threshold functions, monitoring and waiting queues.

3.1 The Algorithms

Recall that our main objective is to achieve some degree of guarantee with respect to the loss ratio for real-time tasks even under overloaded system condition, and to minimize the non-real-time task performance penalty especially under normal load. We have seen that static QLT and MLT do not support our objective. However, two thresholds, *non-real-time queue length* and *real-time minimum laxity* do provide effective means of tuning the performance tradeoffs for given work load conditions. The problem is that there is no fixed tradeoff point providing acceptable performance under different load conditions. From our observations and analysis in the last chapter, we know how the two thresholds play their roles in determining the performance tradeoffs. Generally, both loss ratio ϵ and average delay $\mathcal{E}$ are functions of arrival rates λ_r, λ_n and the threshold, T_q or T_p. Therefore, by adjusting the thresholds to correspond to varying arrival rates, we may be able to control the system performance at different work loads.

3.1.1 Adaptive QLT

Let us look at QLT first. Rewriting (2.17), we get T_q in terms of λ_r, λ_n and ϵ:

$$T_q = \log_{\rho_*} \left[\frac{\frac{\rho_n}{1-\rho_n}(1-\rho_*)}{\frac{1-p_0^r}{(1-\epsilon)\lambda_r} - 1} + 1 \right]^{-1} \tag{3.1}$$

where $\rho_* = \lambda_n / p_0^r$. This of course only gives a rough estimation for T_q within the region that (2.17) is valid. But nevertheless, it expresses T_q as a function of the work load conditions and the desired performance metric ϵ. Let us denote this function by $\mathcal{T}(\lambda_r, \lambda_n, \epsilon)$. Note that $\mathcal{T}$ only remains finite over a limited area. For small constant ϵ,

1. $\mathcal{T}(\lambda_r, 0, \epsilon) = 0$;
2. $\mathcal{T}(0, \lambda_n, \epsilon) = 0$;
3. $\mathcal{T}$ is *non-decreasing* over λ_n ;
4. $\mathcal{T}$ is *non-decreasing* over λ_r .

Therefore, a simple *zero-crossing* simulation program is able to compute $\mathcal{T}$. A comparison of our analytic approximation with simulation results is given in figure 3.1.

The problem arises when $\mathcal{T}$ is infinite. Since the ML scheduling policy used in real-time is *optimal*, this implies the loss-ratio ϵ requirement is unreachable. A trivial solution is just to set $\mathcal{T}$ to be a pre-defined maximum. A more sophisticated approach is to compute the maximum achievable ϵ for different arrival rates, i.e., $\epsilon(\lambda_r, \quad \lambda_n)$ = *loss-ratio of the optimal algorithm for arrival rates* $\lambda_r, \quad \lambda_n$. A system designer may then use this information and specify the required loss-ratio bounds for different load conditions. Finally, $\mathcal{T}$ can be computed to satisfy the specification.

The adaptive version of QLT works as follows:

AQLT: schedule a real-time task unless

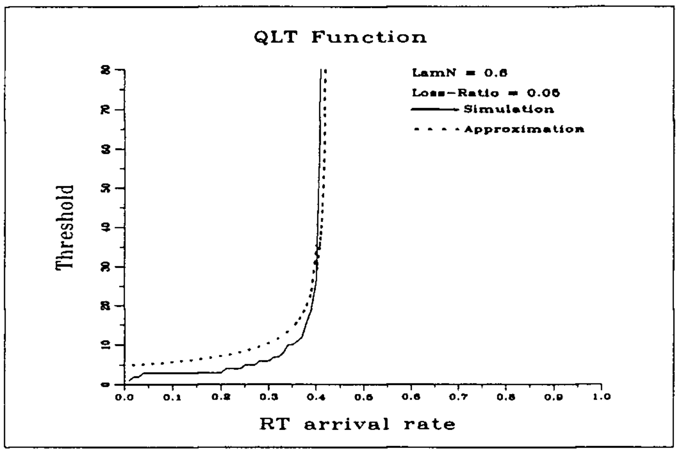

Figure 3.1: QLT Threshold Function

1. *NRT queue length* $> \mathcal{T}$; or

2. *RT queue length* $= 0$.

According to our analysis in the last chapter, the average delay of non-real-time tasks will be approximately T_q/λ_n larger than the delay in the FCFS queue without real-time tasks, as long as the system remains stationary ($\rho_n < 1$). For most of the work load range, $\mathcal{E}$ will be moderate. A rough estimation can be obtained by substituting (3.1) into (2.6).

Finally, we compare the performance of AQLT to QLT in figure 3.2 and 3.3. Note that, in all of our comparisons, the threshold function $\mathcal{T}$ can still be tuned better. Meanwhile $\mathcal{T}$ can be adjusted to achieve different loss ratio bounds and average delay for different applications.

3.1.2 Adaptive MLT

Similarly, Adaptive MLT works as follows:

AMLT: schedule a non-real-time task unless

1. *RT minimum laxity* $\leq \mathcal{T}$; or

2. *NRT queue length* $= 0$.

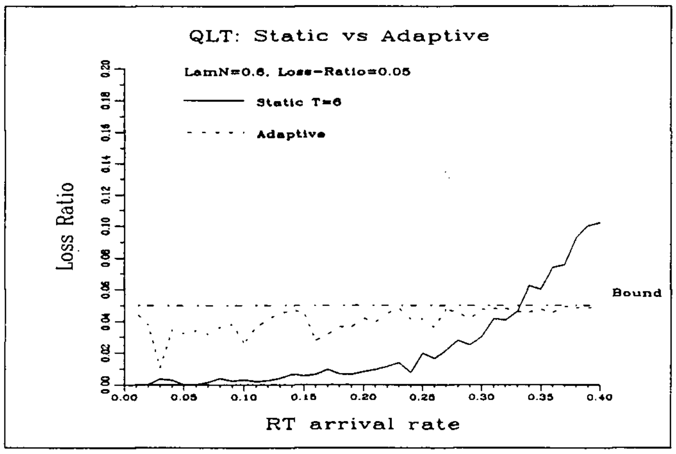

Figure 3.2: Loss Ratio: AQLT vs QLT

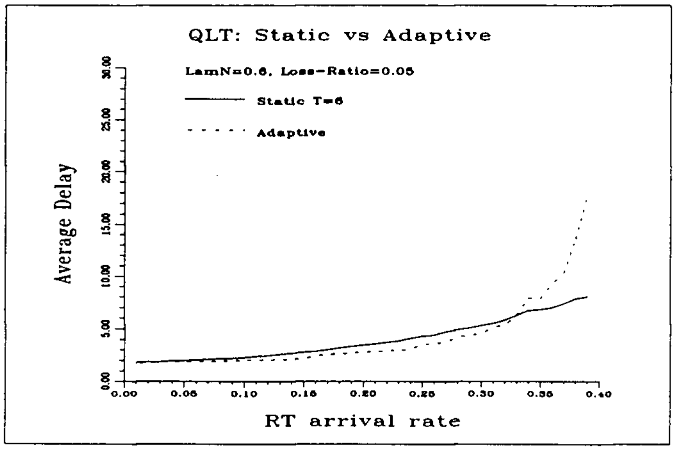

Figure 3.3: Average Delay: AQLT vs QLT

Rewriting (2.35), we get $\mathcal{T}$ for AMLT:

$$\mathcal{T}_p = \frac{1}{1-\lambda_r} \ln \left(\frac{\lambda_r(1-(1-\epsilon)\lambda_r)}{\epsilon} \right) \tag{3.2}$$

Note that this gives an upper-bound for $\mathcal{T}$. The actual value of $\mathcal{T}$ can also be computed by simulation. We show this in figure 3.4.

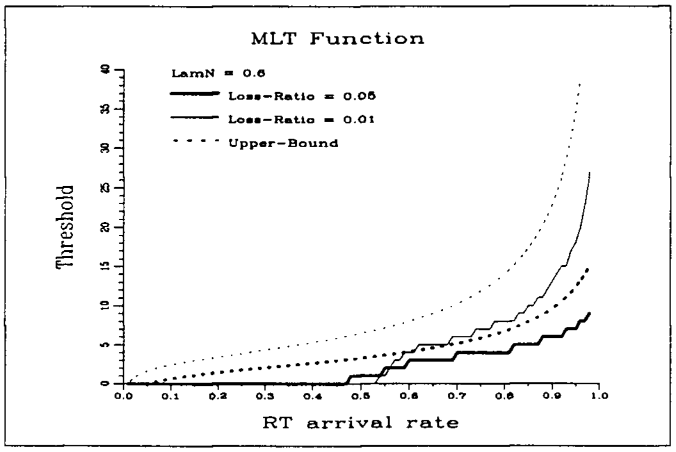

Figure 3.4: MLT Threshold Function

Substituting $\mathcal{T}$ into (2.39), we can estimate the average delay for the non-real-time tasks.

$$\mathcal{E} \approx \frac{1}{\frac{\mu-\lambda_r}{1-\rho_n^2\left(\frac{\epsilon}{\lambda_r(1-(1-\epsilon)\lambda_r)}\right)^{\mu-\lambda_r}} - \lambda_n} \tag{3.3}$$

Our result shows that $\mathcal{E}$ remains small until the system load approaches saturation.

Figures 3.5 and 3.6 compare the performance of AMLT and MLT.

3.1.3 Integration and Other Heuristics

Recall that QLT generally gives bounded average delay for the non-real-time tasks but fails to ensure good real-time task performance in the presence of high non-real-time load. On the contrary, MLT ensures bounded loss-ratio for the real-time tasks, but to do this, its threshold

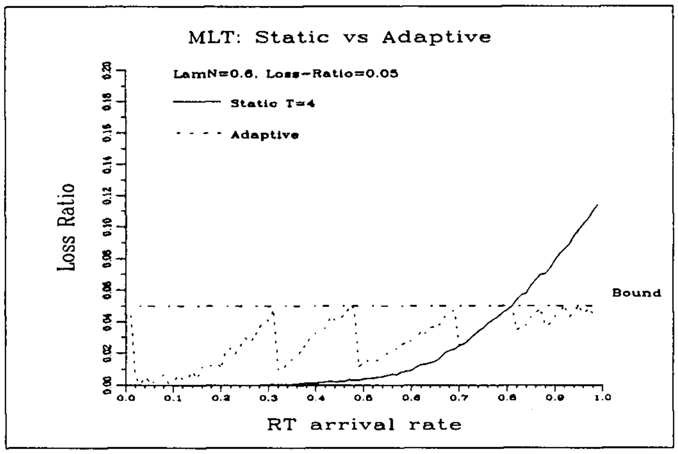

Figure 3.5: Loss Ratio: AMLT vs MLT

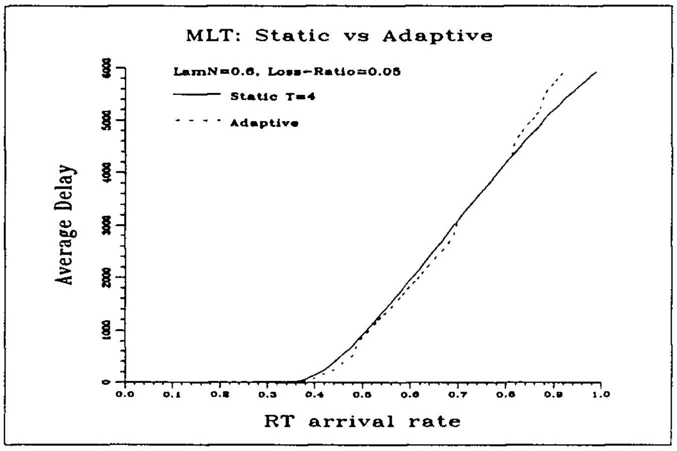

Figure 3.6: Average Delay: AMLT vs MLT

has to be raised so high that it will unnecessarily impose a high delay penalty on the non-real-time tasks. Besides the problem associated with changing arrival rates, another problem causing difficulty in making suitable performance tradeoff is that each of the two thresholds is directly related to only one of the two waiting queues. It is difficult to select the proper value of the threshold to achieve desired performance for both queues. To tackle this issue, we observe that QLT provides satisfactory non-real-time performance under normal work loads, and MLT can provide loss-ratio bound for the real-time tasks under heavy work loads. Therefore, by combining the two algorithms it may be possible to derive a new algorithm which will behave like QLT under light loads and like MLT under heavy loads.

A simple way to do this is as follows. We employ two thresholds, T_p and T_q. Both are actually functions of λ_r and λ_n. The algorithm will normally run as QLT but it will process a real-time task if the minimum laxity in the real-time queue is below the threshold T_p, i.e. MLT takes precedence over QLT. Note that the values of the two thresholds obviously will not be the same as they are used individually. This integrated version of the threshold based scheduling policy will have good performance for non-real-time tasks like QLT at normal loads but still manage to ensure reasonable loss-ratio level as does MLT.

To understand how this works, let us look at some extreme cases. When $T_p = 0$, it schedules like QLT except if a real-time task is due immediately. It is obvious that this integrated version will be better than pure QLT. If λ_r is relatively large, we may expect $T_p = 1$ serving the same purpose better. Therefore T_p can be viewed as representing the urgency of the real-time tasks. The basic idea behind this is to multiplex limited resources between the real-time and non-real-time tasks more wisely. Resources should be devoted to the real-time tasks only when they reach a certain urgency. The minimum laxity of the real-time tasks is one indication of such urgency. Our result shows that this simple heuristic works well. Another indication of urgency is a long continuous real-time task train. Because of the bursty nature of most computing activity, this type of traffic is not unusual. A laxity threshold, however, is not good at dealing with such situations. It requires certain *global* information and *look-ahead* capability to solve this kind of problem. In the extreme case, if we have full information about the future, we can achieve optimal scheduling. In realty, all we know is the information on tasks in the waiting queue and what we can use is limited to a small portion of it due to the run time overhead. However, the following two simple heuristics may help:

1. Instead of using the minimum laxity of all real-time tasks, we take the sum of the two smallest laxities of the real-time tasks. We will service real-time tasks if this sum is below twice T_p. If $T_p = 1$, one of the two tasks that arrived at the same time would be lost in the original algorithm. This simple heuristic prevents such a loss because these two tasks will be detected two units of time ahead of their deadline. Generally, we can take the sum of k minimum laxities. If the queue is organized in order of increasing minimum laxity first, we only need to take the sum of the laxities of the first k tasks.

2. Similar to the idea of queue length threshold for the non-real-time tasks, we may organize the real-time queue into two subqueues. Incoming tasks will enter the first queue in the order of laxity. The first task in the first queue will be placed at the tail of the second queue when its laxity goes below some threshold T_1. We then compare the queue length of the second queue to another threshold T_2. If the length exceeds T_2, the urgency of real-time tasks is indicated. This heuristic works similarly to the previous one except we now count in another domain, queue length instead of laxity.

Note that these heuristics will ease the problem of continuous loss. These two strategies work similarly in general cases, but the first one is simpler and more efficient to implement. In the next chapter, we will evaluate the performance of the integrated adaptive scheme without using any of these heuristics.

A rough estimation of the performance achieved by this integrated policy (ADP) can be obtained by applying the results of chapter 2. When λ_n is large, ADP is similar to MLT. Then ϵ is bounded as (2.35) indicates and $\mathcal{E}$ can be estimated by (2.39). Note that (2.35) fits the true value well for large λ_n. Similarly, when λ_r is small, ADP resembles QLT. The discussion in section 2.2.1 provides equations for a rough estimation. So ADP ensures bounded loss ratio for the real-time tasks even when the non-real-time load is high, and provides similar average delay for the non-real-time tasks as QLT when the real-time load is light. Unfortunately, the values of the two thresholds cannot be derived independently. In fact, for given a performance requirement, the corresponding threshold values may not be unique. We will use simulation results in the next chapter for performance evaluation.

3.2 Implementation Related Issues

3.2.1 Threshold Functions

Given a constant ϵ that we wish to achieve, $\mathcal{T}$ can be plotted as a plane over $(\lambda_r,\ \lambda_n)$. $\mathcal{T}$ is almost a simple step function of small values when $\lambda_r + \lambda_n < 1$. Depending on the requirements of the applications, the following methods may be used to implement the threshold function:

1. Analytic Approximation: The estimation or bounding results derived in the last chapter may satisfy some applications. Of course, $\mathcal{T}$ can be pre-computed to avoid the time required to compute those functions in real time.

2. Numerical Approximation: We can also estimate $\mathcal{T}$ by numerical fitting methods. Let us take QLT as an example. Let $\epsilon = 0.05$. After computing $\mathcal{T}$ numerically or by simulation, we can plot a series of *isograms* for different T's [figure 3.7].

   From these isograms, we may conjecture that there is a function $f(T, \lambda_r)$ so that $\lambda_r + \lambda_n + f(T, \lambda_r) = 1$. Our further study shows that the following formula may provide satisfactory estimation:
   $$T = \frac{1}{K}\left(\frac{(1-\lambda_r)^2}{1-\lambda_r-\lambda_n-c} - K_0\right) \tag{3.4}$$
   where K_0,K and c are parameters to be determined experimentally.

3. Look-up Tables: A more practical method is to compute $\mathcal{T}$ by simulation or actual measurement. The result then can be stored in a look-up table. Since $\mathcal{T}$ is roughly a step function, it can be stored and retrieved very efficiently.

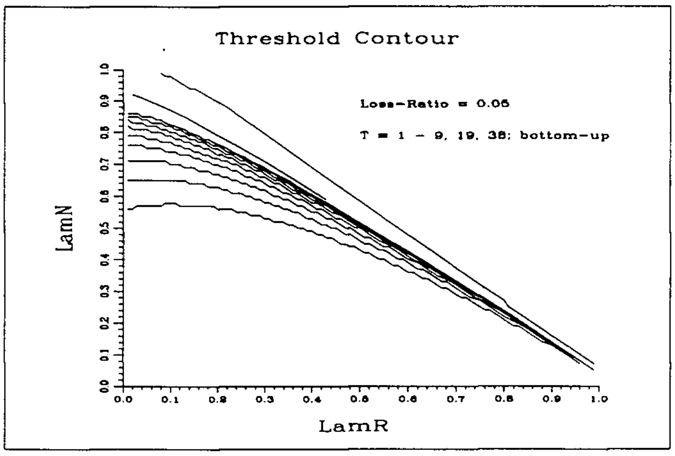

Figure 3.7: QLT Threshold Isogram

3.2.2 Run-time Monitoring

Assuming the threshold function has been obtained, the next problem is how to monitor or measure the system work load. Noting that our system is a predicting control system [figure 3.8], we face a trade-off between *accuracy* and *stability* . This can be explained as follows.

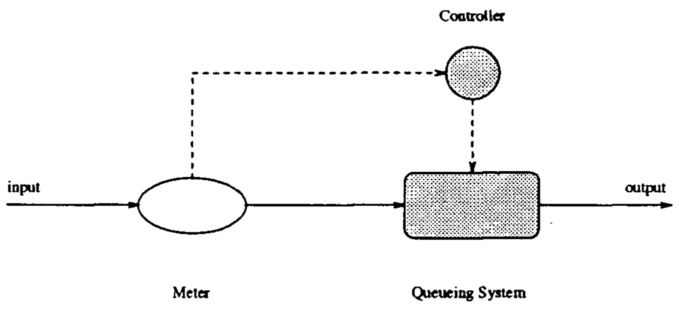

Figure 3.8: Monitor and Controller

In a discrete and time-sharing environment the controller has to share the only processor with the server and the monitor, thus arrival rate monitoring can only be done periodically. Consequently control decisions are also made periodically. This period has to be relatively large to make the scheduling process feasible. Note the following formula defines all *rate* quantities,

$$\lambda = \frac{W}{T}$$

where W is the number of events that occurred during time interval T. Now we have two possible approaches to measure λ. If we set T constant, then the meter will count events and reset itself at the end of each time interval T. If the system has a low resolution timer but high rate of events, this *event frequency* approach produces better accuracy. On the other hand, we may set W to be constant. The meter then reports the *time interval* for every W events. Time interval monitoring is better when event frequency is low and the system has a relatively high resolution timer. [20] discussed a similar problem arising in a software feedback adaptive scheduling system. This also raises the problem of choosing a predicting or a feedback scheme. A predicting system applies input information to control, while a feedback system uses output information for control. In our case, we may also monitor loss ratio or average delay and compare them to some threshold to make scheduling decisions. It is however more difficult to analyze such a system. Furthermore, the feedback system is more vulnerable to the lagging effect discussed below.

Accurate measurement of arrival rates is not sufficient. Another problem to be considered is the *Lagging Effect* existing in the system. Since the queueing system has a non-uniform time latency, the monitored load condition at the input stream does not necessarily match the situation faced by the server. If control is applied out of phase, it can actually magnify the effect it was designed to reduce [2]. The worst case occurs when the arrivals are wavelike and the monitor happens to lag behind a half period [figure 3.9]. More losses will occur than when there is no control at all. This is a disastrous situation that has to be prevented.

The solution is to sacrifice *accuracy* to achieve better *stability* . If we set the monitoring period (W or T, whichever is applicable) long, we lose information on small variations but obtain better estimation in the global term. The worst case scenario will have a much smaller chance of occurring. On the other hand, if this period is too long, we will lose control significantly. Therefore, this is a tradeoff we have to make according to the nature of the application.

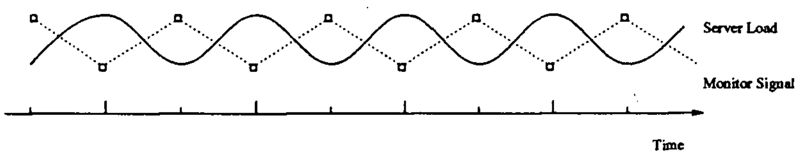

Figure 3.9: Lagging Effect

Generally, this tradeoff should be made according to the pattern of λ_r. Let us assume that λ_r changes moderately at most times but dramatically in short periods from time to time. By considering specific applications, the system designer may know what is the shortest peak period that the system is designed to resist. Let the length of such period (either in time unit or in number of events) be t_0, and the length of the monitoring period be T_0. The worst case is when each of two continuous monitoring periods covers a half of a small pulse [figure3.10]. Then the measured value is $\overline{b} = \frac{1}{T_0}\int_0^{\frac{1}{2}t_0} f(x-x_0)dx$. This $\overline{b}$ is used to match the left side of the threshold function look-up table $(\lambda_r[i],\ T[i])$, where T is T_p or T_q. A simple rule to determine T_0 is: $\overline{b}$ should be large enough so that it matches the proper table item. This also directly depends on the fine-graininess of the threshold function table. If this table has many records, T_0 should be smaller, and vice versa. More explicitly, let the error be

$$\delta = \left| \frac{1}{t_0}\int_0^{t_0} f(x-x_0)dx - \frac{1}{T_0}\int_0^{\frac{1}{2}t_0} f(x-x_0)dx \right|$$

then the condition of accurate measurement is, for all i,

$$\delta < \frac{1}{2}\left|\lambda_r[i+1] - \lambda_r[i]\right| \tag{3.5}$$

Since δ is a function of T_0, this gives us a means to select the proper value of T_0 that satisfies (3.5). A rough estimation following this principle may be sufficient in practice.

Note that this problem is just an instance of a common existing problem in many areas, e.g. to determine the size of a signal filter. This is an *ill-posed* problem because there is no clear definition of what is best. Another example is the filter used to eliminate noise in an image. There is no explicit difference between noise and data. If the filter size is too large, a lot of

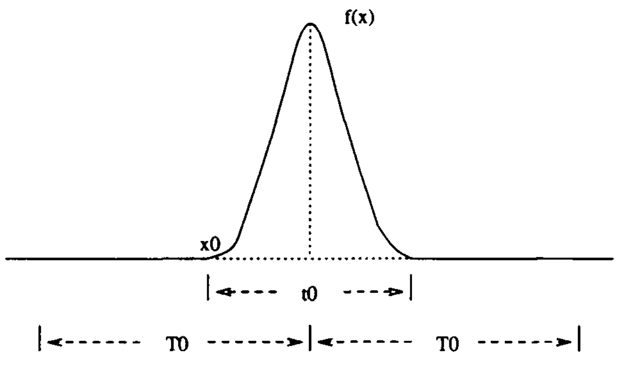

Figure 3.10: Monitoring Period

noise will remain. If the filter is too small, we take the risk of losing data along with the noise. Since we can only determine what is noise case by case, the solution to this problem relies on the specific semantics of the individual applications.

3.2.3 Real-Time Queue Implementation

The complexity of the waiting queue implementation is another major issue. A sorted queue for ML requires $O(n)$ time complexity for practical use[1]. In [23, 9], an efficient approximate scheme ML(n) is studied. ML(n) separates the waiting queue into two parts. Tasks enter an FCFS queue first and then enter a sorted queue with constant length n for service. By setting the constant n appropriately, this effectively reduces the time complexity to a small constant while still retaining performance competitive with ML. This scheme will work well if we can assume that later arriving tasks have larger laxity, otherwise we risk the possibility that an urgent task in the FCFS queue will be lost.

Another way to reduce the complexity is to organize the waiting queue into an array of n subqueues. The i^{th} subqueue holds tasks with laxities ranging from L_{i-1} to L_i. L_i should be defined so that the average length of each subqueue will be approximately equal. Within each subqueue, a simple sorting algorithm can be used. If n is large enough so that the number of

[1]$O(\log(n))$ can be achieved by using a tree data structure. However, this does not appear to be efficient when n is small.

tasks in each subqueue is very small, a nearly constant time is sufficient to sort the subqueue. In some processors, like the MC68020, only one instruction is needed to select the first non-empty subqueue. This reduces the total complexity to nearly constant. Note that this is actually a simple case of using a tree data structure. Though not exactly constant, it may be good enough in practise.

3.3 Summary

In this chapter, two adaptive threshold-based schemes (AQLT and AMLT) were proposed. They are different from their static counterparts in that the thresholds in the adaptive schemes change as the arrival rates vary. We have shown that by setting the threshold functions properly the adaptive schemes can achieve bounded loss ratio for real-time tasks and lower average delay for non-real-times than the static schemes in the dynamic environment. The threshold values can be calculated by approximate formulas or by simulation. A new scheme (ADP) integrating AQLT and AMLT was then proposed to achieve better performance tradeoffs. ADP has good performance for non-real-time tasks like QLT at normal loads but still manage to ensure reasonable loss ratio levels as MLT does. Issues related to the implementation of the threshold functions, load monitor and waiting queue were also discussed.

Chapter 4

Performance Evaluation

In this chapter, we evaluate the performance of our new scheduling algorithm (ADP) described in chapter 3 by comparing it with some other standard algorithms. The performance metrics that will be used are defined in section 4.1, the various algorithms to be compared are described in section 4.2. Section 4.3 briefly introduces our simulator. Section 4.4 presents the evaluation results along with brief discussions and section 4.5 summarizes the chapter.

4.1 Performance Characteristics

Two performance metrics are of interest for systems that handle both real-time and non-real-time tasks. The major metric for real-time tasks is *loss ratio* defined as the fraction of lost tasks, i.e. those that do not meet their deadlines, over the total number of tasks. Non-real-time tasks will not be 'lost' but may not receive immediate service when they arrive. For these tasks, the main concern is the *average delay* . This delay includes the queue waiting time and the processing time. To evaluate the various scheduling algorithms, we need to define other workload and system parameters of the queueing system. These parameters include the distribution of incoming tasks, the distribution of service times, and the distribution of laxity values for real-time tasks. A common assumption is to let all of these be exponentially distributed. We will compare the performance of the scheduling algorithms under different assumptions. Other distributions like uniform, geometric, combination of uniform and exponential, and train model will also be used to evaluate the *robustness* of the various algorithms. Besides the stationary behavior of the system, we would also like to investigate the system performance in the transient state when it is congested.

4.2 Algorithms Under Comparison

The algorithms that are going to be compared to our adaptive algorithm are introduced in this section. These algorithms include First Come First Serve (FCFS), Static Priority (SP), Minimum Laxity First/Earliest Deadline First (ML/EDF), and Stanford Optimal (OPT). We describe each of the above algorithms in this section.

- **FCFS**: This policy serves tasks in the order of their arrivals without using any timing information.

- **SP**: This policy schedules tasks in the order of their *priorities* . Among tasks with the same priority, FCFS will be used. There are two levels of priorities in our case. Real-time tasks have higher priority than non-real-time tasks. Besides priority, SP knows no other timing information.

- **ML**: Each real-time task will be associated with a *latest time to start* , i.e. *laxity.* The task with minimum laxity will be scheduled first. If a task cannot be scheduled by this time, it is *lost* and will not be serviced at all. This algorithm has been shown to be optimal in reducing loss ratio under certain conditions [25].

- **EDF**: EDF is similar to ML. The difference is that here we have a *latest time to finish,* i.e., *deadline* , associated with each real-time task. Tasks are scheduled in order of their deadlines. A task unable to finish by the deadline is considered lost. Since the scheduler can determine whether a task can make its deadline right before it is about to start the task, lost tasks can be simply dropped without receiving any service. This version of EDF is also proven to be optimal in minimizing the loss ratio [25]. Also note that EDF is equivalent to ML if all tasks have identical laxity.

- **OPT**: OPT is an optimal algorithm introduced in [26]. It schedules both real-time and non-real-time tasks. It is optimal in the sense that it minimizes the loss ratio of real-time tasks and at the same time, among all such schedules, it minimizes the average delay of non-real-time tasks. Therefore this algorithm distinguishes between real-time and non-real-time tasks. It first guarantees real-time task performance, then provides the best possible delay performance for non-real-time tasks. The algorithm assumes complete knowledge of all future tasks. Therefore, it is not motivated for practical use. However, it provides a bound for best possible performance and serves as a standard for comparison. We use a special case of the original general algorithm, with two classes of tasks and equal weight within the same class. The readers are referred to [26] for a detailed description. Though its complexity has been reduced to $O(n^2)$, the algorithm is still extremely slow especially for congested arrival and large laxities. It can take hours on a fast workstation to produce a single point on the performance graph.

4.3 The Simulator

The study is carried out by simulation. Our simulator is of the type whose structure resembles that of the real system to be evaluated. Figure 4.1 shows the structure of the simulator. The arrival processes, and all timing properties of the tasks are modeled using independent random number generators. Different random distributions are generated using the UNIX utility `random()`, which uses a non-linear additive feedback random number generator producing pseudo-random numbers with a period approximately $16 \times (2^{31} - 1)$. A logical timer is used in the simulator. Incoming tasks enter a waiting queue if they are not immediately scheduled upon arrival. This waiting queue is organized into two different subqueues for some algorithms. For example, in ADP, there is a real-time subqueue sorted in the order of laxities and a subqueue for the non-real-time tasks organized in FCFS. At each time unit, the scheduler looks at the two queues and schedules or drops tasks according to its scheduling policy. All tasks that leave the server, with or without receiving service, will pass through an output module that collects and reports needed information.

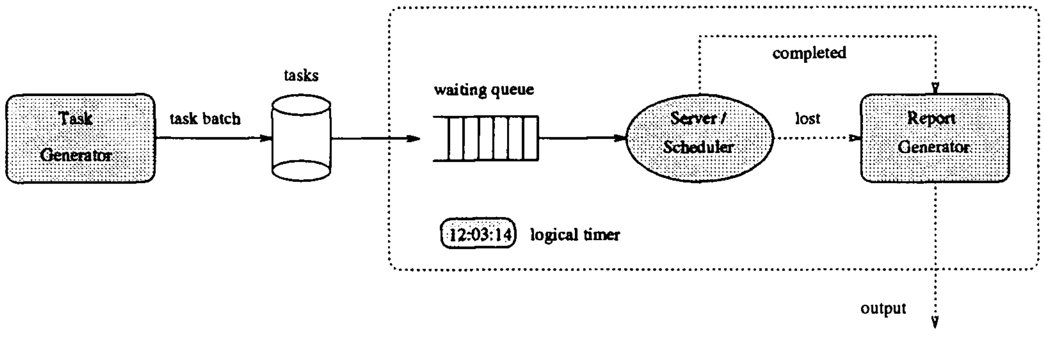

Figure 4.1: The Simulator

Each point in the graph was obtained from a simulation experiment with a duration equal to 10,000 units of time. All tasks are considered *non-preemptive* .

The simulator was validated by checking through complete output lists of some simple examples, and by comparing to other published results. The main part of the simulator (except for the scheduling policies) is also validated by comparing the simulation outputs to the exact analytic results for some simple scheduling disciplines, e.g. FCFS and SP.

4.4 Performance Comparisons

Our first set of comparisons evaluate the performance of the above scheduling policies under the following conditions. All tasks are of constant service time equal to one unit. Real and non-real-time task arrivals constitute two Poisson processes of rates λ_r and λ_n respectively, normalized by service time, i.e., in tasks/unit. We assume a task's laxity upon arrival to be $s - B$, where s is a constant and B is an exponentially distributed random variable, conditioned on $s - B > 0$.

Figures 4.2 and 4.3 compare performance of the various algorithms under a constant total work load, $\rho = \lambda_r + \lambda_n = 0.9$. A very simple T_q function (a function of λ_r only) and constant $T_p = 7$ are used in ADP. We observe that both the loss ratio and mean delay increase as the percentage of real-time tasks grows, with the exception that the average delay in OPT and ADP actually decrease slightly with respect to λ_r when λ_r is small (< 0.2). This is because when a small number of real-time tasks are not near their deadlines, there is a chance that the real-time tasks can be delayed in favor of the non-real-time tasks. Using the simple settings of the threshold values, ADP achieves satisfactory loss ratio (< 0.05) and better mean delay than ML. Note that ML is optimal with regard to loss ratio in this case. We set the laxity of non-real-time tasks to be infinity for ML. While ADP does well in loss ratio performance, the mean delay in ADP is closer to that of SP and ML than that of OPT or FCFS. Of course ADP can achieve better mean delay by adjusting the threshold values, but a more important feature of ADP shown in figure 4.4 promises much better mean delay performance for larger laxities. Figure 4.4 shows that the mean delay achieved by OPT and ADP decreases significantly as the laxity increases. This is because with large laxity, OPT and ADP have greater flexibility in multiplexing the processor between real-time and non-real-time tasks. Under that circumstance, non-real-time tasks have more opportunity to get priority over real-time tasks. However, laxity is irrelevant for SP and FCFS since they do not use this information at all. ML is even worse in that mean delay slightly increases (though not significantly) with laxity because fewer real-time tasks are dropped with larger laxity. As laxity goes to infinity, ML has the same mean delay as SP because no real-time tasks are dropped. By deferring service for real-time tasks until their laxities have become small, we can greatly improve the performance of non-real-time tasks. We conclude that, for larger laxity tasks, the mean delay produced by ADP is close to that of OPT and much better than that of ML. This is shown in figure 4.5. The average delay achieved by ADP is clearly close to OPT and 3 to 5 times better than that of SP and ML until the arrival rate of the real-time tasks becomes high (> 0.75). Even with high real-time load, ADP still achieves much lower average delay than SP and ML. More importantly, for most regions when the real-time load is not very high, ADP performs significantly better than FCFS in average delay. This is again because ADP is able to speed up non-real-time tasks by holding real-time tasks until their laxities become small. Also note that the loss ratio for ADP is bounded below $\epsilon = 0.05$, which is much lower than that of SP and FCFS. This is a desirable property and one that we would like to see from a good scheduling algorithm, i.e. to ensure an upper bound for loss ratio even under high load and keep average delay down for most work load conditions.

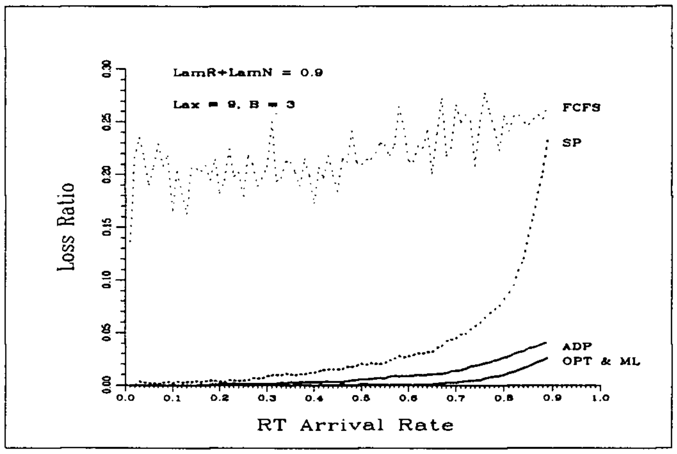

Figure 4.2: Loss Ratio: with constant total work load

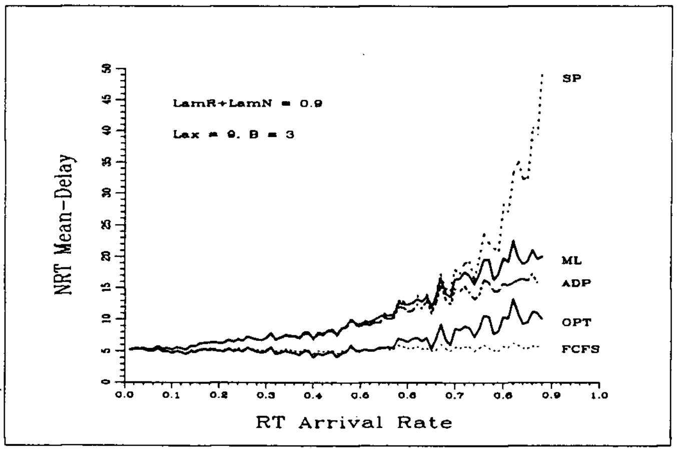

Figure 4.3: Average Delay: with constant total work load

The next set of simulation experiments compare the performance of the scheduling policies during congested and overloaded periods. For transient states of the system, we are interested

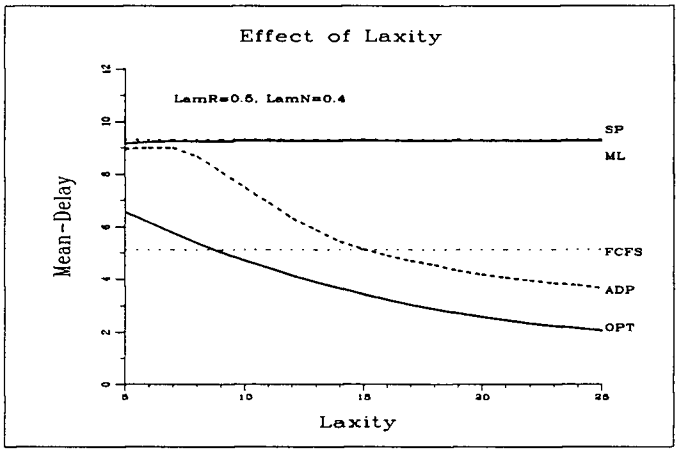

Figure 4.4: Average Delay vs Laxity

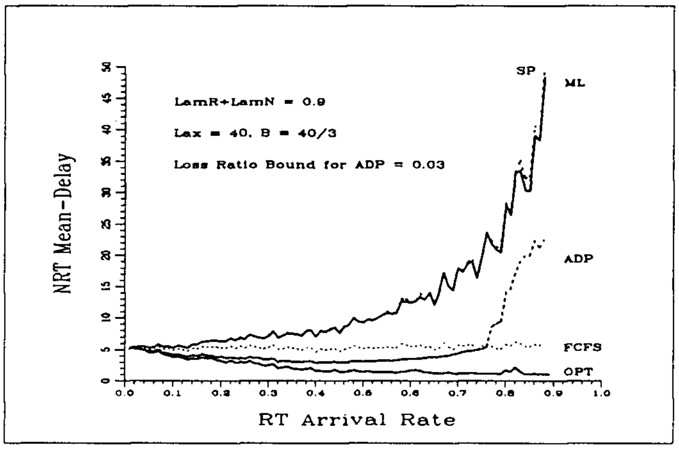

Figure 4.5: Average Delay: with constant total work load (large laxity)

in the values as well as the rate of increase of the performance metrics in a fixed period of congestion. In our simulation experiments, this period is set to be 10,000 time units.

Figures 4.6 and 4.7 show the loss ratio and average delay when $\lambda_n = 0.2$. Again, T_q is set to be a function of λ_r and T_p is kept constant (at 7). We observe that ADP has near optimal loss ratio and significantly smaller average delay than both ML and OPT. The rate of increase of the non-real-time average delay over λ_r is linear for all policies. But for real-time task loss ratio, FCFS and SP perform very poorly with a dramatic increase beyond certain points. Figures 4.8 and 4.9 show the same thing for larger laxity ($S = 40$). In this case, ADP offers much smaller average delay in the overloaded region with a moderately small loss ratio. Though OPT and ML can achieve near zero loss ratio, their performance with respect to non-real-time tasks are significantly worse than ADP. For applications that can tolerate certain task loss, ADP is preferred. Note that ADP can even achieve lower average delay than FCFS over some load conditions because ADP not only drops a few real-time tasks but also gives priority to the non-real-time tasks whenever possible. When the real-time load is relatively low, this may make the performance of non-real-time tasks in ADP better than in FCFS.

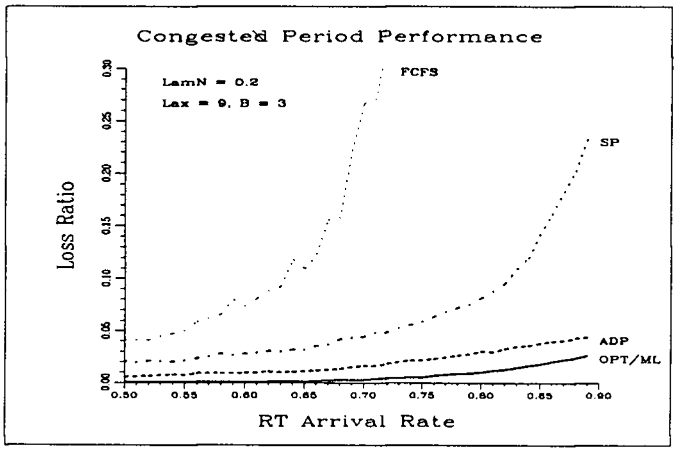

Figure 4.6: Loss Ratio in Overloaded System

We now look at a small case of performance achieved by the different scheduling algorithms.

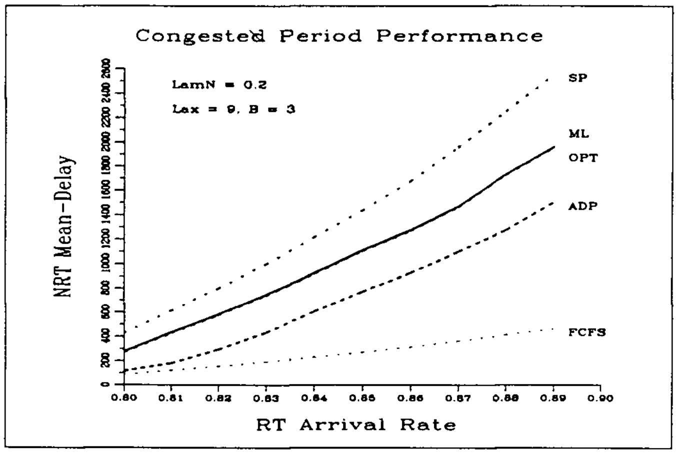

Figure 4.7: Average Delay in Overloaded System

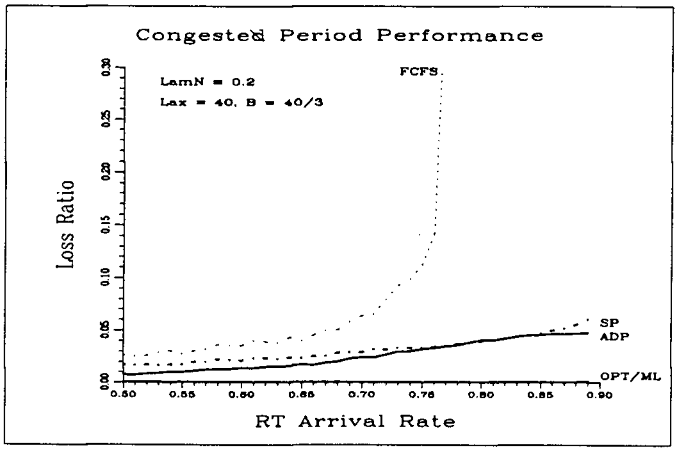

Figure 4.8: Loss Ratio in Overloaded System (large laxity)

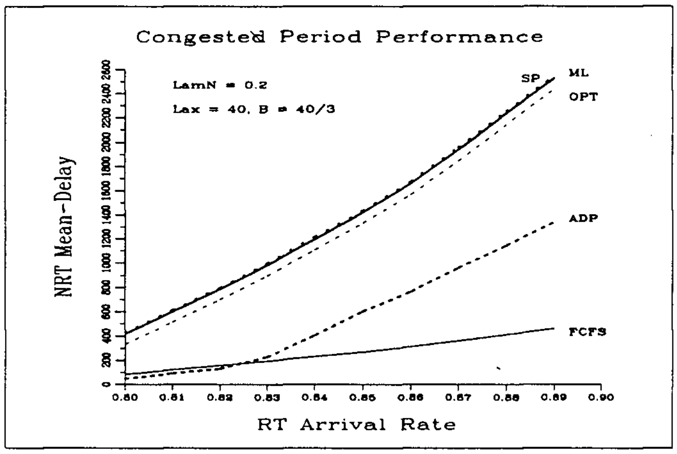

Figure 4.9: Average Delay in Overloaded System (large laxity)

The simple threshold function used in ADP is shown in Table 4.1. The non-real-time task arrival rate λ_n is set to be constant at 0.4. The average delay of the non-real-time tasks and the real-time task loss ratio are listed in Tables 4.2 and 4.3. We can see that a very simple (therefore not expensive to implement) threshold function can achieve fairly good performance. (Note that the loss ratio bound ϵ for ADP is set to be constant at 0.03.) Note that we are using very simple threshold functions throughout all experiments. This is because, with the timing granularity in our experiments, a simple threshold function has already obtained most benefits. We expect this to be true for most applications. Generally, more fine-grain timing in an application requires more complicated threshold functions.

λ_r	0.1 - 0.4	0.5	0.6
T_p	1	2	8
T_q	0	0	13

Table 4.1: Threshold Functions (LamN = 0.4)

Finally, we examine the robustness of ADP by comparing the performance of ADP under different arrival distributions and laxities.

λ_r	0.1	0.2	0.3	0.4	0.5	0.6
FCFS	1.52	1.79	2.28	2.86	5.13	69.25
SP	1.57	1.98	2.88	4.09	9.30	169.87
ML	1.57	1.98	2.88	4.09	9.26	165.12
OPT	1.33	1.34	1.36	1.37	1.42	128.93
ADP	1.33	1.34	1.35	1.37	1.39	28.01

Table 4.2: NRT Average Delay (LamN = 0.4)

λ_r	0.1	0.2	0.3	0.4	0.5	0.6
FCFS	0.01	0.01	0.02	0.03	0.06	0.83
SP	0.002	0.003	0.008	0.010	0.017	0.021
ML	0.0	0.0	0.0004	0.0005	0.0006	0.0007
OPT	0.0	0.0	0.0004	0.0005	0.0006	0.0007
ADP	0.001	0.002	0.01	0.016	0.030	0.030

Table 4.3: RT Loss Ratio (LamN = 0.4)

We first compare the performance of ADP for four different arrival distributions: exponential, geometric, uniform and train model. In the train model, tasks come in continuous bunches like a train. At any time slot, a train may appear with a constant probability. All trains are of fixed length L. We adjust the parameters of the above distributions to make their mean arrival rate identical.

We observe little variation of loss ratio over most regions until the system is saturated. (figure 4.10). On the other hand, figure 4.11 shows that the average delay is quite different for different distributions. Our conclusion is that ADP performs better if arrivals come with higher randomness. For the train model with length $L = 5$, ADP has worst average delay and loss ratio compared to the other arrival distributions. In the overloaded region, however, the average delay has little variation (figure 4.12). To see how the other scheduling policies perform, we also let them take on train-like arrivals. Figure 4.13 shows that ADP is much better than the others.

Since laxity has a great deal to do with the advantage that ADP can have, we compare ADP's performance under different arrival distributions and laxities. We see from figure 4.14 that there are similar decreases in mean delay for all arrival distributions. Note also that in the case of train model and geometric distributions, there is a slight increase in mean delay when laxity is small. This is because, like ML, fewer real-time tasks will be dropped as laxity increases.

4.5 Summary

In summary, ADP is a flexible way of scheduling a mixture of real-time and non-real-time tasks. It offers the system designer explicit tradeoff choices depending on the needs of the applications. We found that ADP provides satisfactory performance trade-off under widely varied conditions. For most cases, it offers bounded loss ratio for real-time tasks and significantly lower average delay for non-real-time tasks compared to the other common policies.

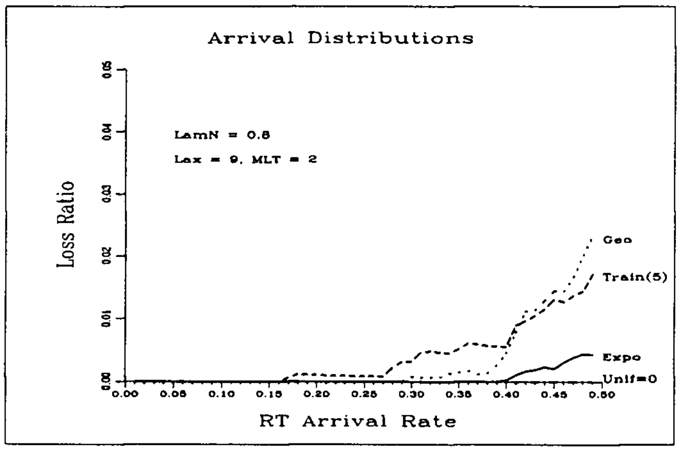

Figure 4.10: Loss Ratio: For Different Arrival Distributions ($\epsilon < 0.05$)

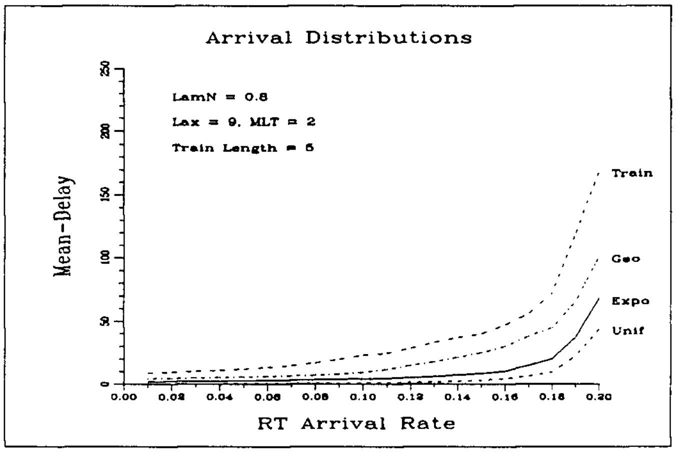

Figure 4.11: Average Delay: For Different Arrival Distributions

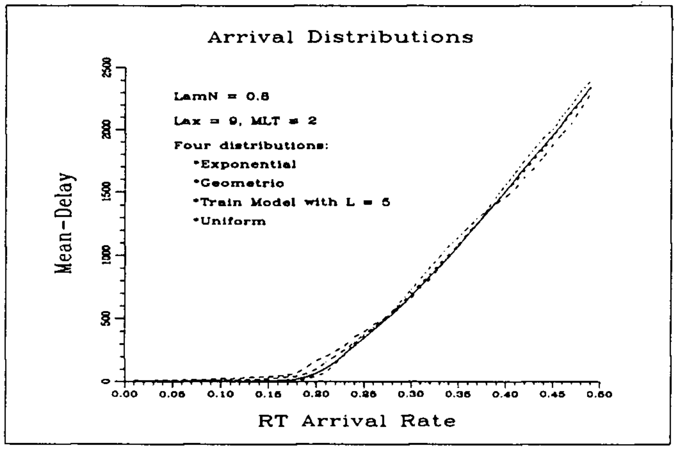

Figure 4.12: Average Delay: For Different Arrival Distributions (overloaded region)

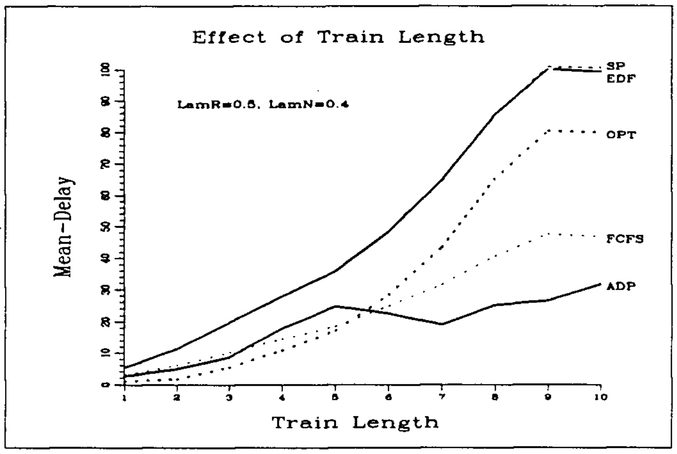

Figure 4.13: Effect of Train Length

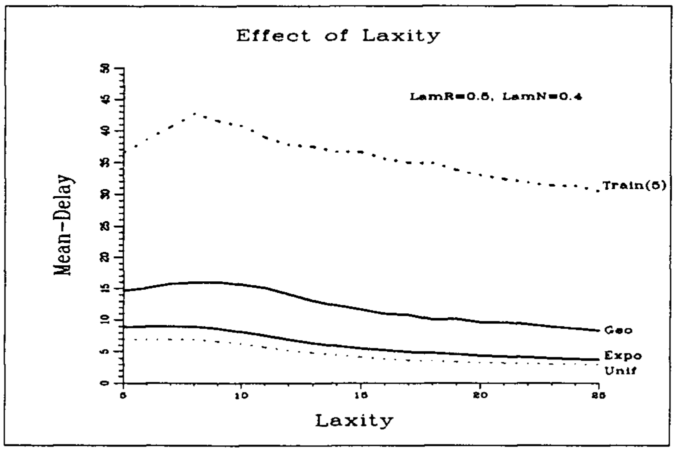

Figure 4.14: Effect of Laxity: For Different Arrival Distributions

Chapter 5

Conclusion and Future Work

This thesis has explored scheduling mixed real-time and non-real-time tasks adaptively in a dynamic system environment. We studied two threshold-based mechanisms, Queue Length Threshold (QLT) and Minimum Laxity Threshold (MLT), in multiplexing system resources among the two classes of tasks. Our simulation and analytical results showed that neither static QLT nor MLT could provide satisfactory performance in a dynamic environment where the arrival rates are not constant, and especially for bursty arrivals. When the system load varies, QLT biases toward non-real-time tasks, while MLT favors real-time tasks. The approximate analysis in chapter 2 illustrated and proved essential characteristics of these two threshold-based policies.

Based on these observation and analysis, we proposed an adaptive scheme employing threshold functions. Instead of trying to set one trade-off point for all situations, we capture the best trade-off points for diverse system conditions by a series of threshold values. We described two such adaptive schemes, AQLT and AMLT, and analyzed the performance they can be expected to achieve. We also discussed the idea of integrating the two threshold mechanisms to provide performance trade-offs, and addressed some important implementation related issues.

Though the exact performance of our integrated adaptive algorithm (ADP) depends on the actual values of the thresholds, we showed two major performance metrics achieved by ADP, loss ratio and average delay, for typical loss ratio bounds and various load conditions. Simulation experiments were conducted to compare the strengths and weaknesses of ADP with other common scheduling policies and the optimal algorithm. We found that ADP offers the best performance trade-off under a wide range of conditions. For most cases, it offers bounded loss ratio for real-time tasks and significantly lower average delay for non-real-time tasks than the other policies under comparison. ADP also achieves the desired performance trade-off during overloaded periods. All of these were achieved by very simple step-wise threshold functions. In short, ADP meets our goals very well in scheduling real-time and non-real-time tasks with different performance metrics. It is a good scheduling algorithm for applications such as multimedia processing and communications where there is a mix of real-time and non-real-time tasks, and where the occasional drop of a real-time task is non-fatal.

5.1 Future Work

This study can be further pursued in several ways. In order to develop a scheduling algorithm for practical use, we need to test our ideas in real systems with real applications. Many insights and understanding will result from a real implementation of our algorithm. Issues of implementation overhead and the feasibility in a real system are only addressed briefly in this thesis. The work we have done here indicates that it is worthwhile to investigate these issues further in the future.

The work on mathematical analysis in chapter 2 could be developed further to obtain better approximation or even exact explicit results of the most important characteristics of the two threshold strategies.

Bibliography

[1] Arnold O. Allen, "Probability, Statistics, and Queueing Theory with Computer Science Applications", Academic Press, New York, 1978.

[2] Richard Bellman, "Adaptive Control Processes: A Guided Tour", Princeton University Press, Princeton, New Jersey, 1961.

[3] R. Bettati, D. Gillies, C. C. Han, K. J. Lin, J. W. S. Liu and W. K. Shih, "Recent Results in Real-Time Scheduling", Report No. UIUCDCS-R-90-1629, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1990.

[4] Partha P. Bhattacharya and Anthony Ephremides, "Optimal Scheduling with Strict Deadlines", *IEEE Transactions on Automatic Control*, Volumn 34, Number 7, July 1989.

[5] Renu Chipalkatti, James F. Kurose and Don Towsley, " Scheduling Policies for Real-Time and Non-Real-Time Traffic in a Statistical Multiplexer", *Proceedings of IEEE Infocom'89*, Ottawa, Ont., Canada, April 25-27, 1989.

[6] Harvey M. Deitel, "An Introduction to Operating Systems", Addison-Wesley, 1984.

[7] Domenico Ferrari, "Computer Systems Performance Evaluation", Pretice-Hall, Inc., 1978, Englewood Cliffs, New Jersey, 07632.

[8] Michael R. Garey and David S. Johnson, "Computers and Intractability", W.H. Freeman and Company, New York, 1979.

[9] P. Goli, James F. Kurose and Don Towsley, " Approximate Minimum Laxity Scheduling Algorithms for Real-Time Systems", Technical Report, COINS Department 90-88, University of Massachusetts at Amherst.

[10] R. L. Graham, E. F. Lawler, T. K. Lenstra and A. H. G. Rinnoy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey", *Annals of Discrete Mathematics*, Volumn 5, 1979. pp. 287-326.

[11] Ralf Guido Herrtwich, "An Introduction to Real-Time Scheduling", TR-90-035, International Computer Science Institute, Berkeley, CA.

[12] Jiawei Hong, Xiaonan Tan and Don Towsley, "The Binary Simulation of the Minimum Laxity and Earliest Deadline Scheduling Policies for Real-Time Systems", COINS Technical Report 89-70, Department of Comput. Inform. Sci., University of Massachusetts, July 1989.

[13] Jiawei Hong, Xiaonan Tan and Don Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System", *IEEE Transactions on Computers*, Volume 38, Number 12, December 1989.

[14] Jane W. S. Liu et al., "Algorithms for Scheduling Imprecise Computations", *IEEE Computer*, Volumn 24, Number 5, 1991.

[15] Leonard Kleinrock, "Queueing Systems, Volumn I: Theory", John Wiley & Sons, 1975.

[16] James F. Kurose and Renu Chipalkatti, "Load Sharing in Soft Real-Time Distributed Computer Systems", *IEEE Transactions on Computers*, Volume C-36, Number 8, August 1987.

[17] Eugene L. Lawler, "A *pseudopolynomial* algorithm for sequencing jobs to minimize total tardiness", *Annals of Discrete Mathematics* , Volumn 1, 1977. pp. 343-362.

[18] T. K. Lenstra, A. H. G. Rinnoy Kan and P. Brucker, "Complexity of Machine Scheduling Problems", *Annals of Discrete Mathematics*, Volumn 1, 1977. pp. 343-362.

[19] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *Journal of ACM*, Vol. 20, No. 1, Jan. 1973.

[20] Henry Massalin and Calton Pu, "Fine-Grain Adaptive Scheduling using Feedback", USENIX Computing Systems, Volume 3, Number 1, Winter, 1990.

[21] A. K. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment", *Proceedings of the 7th Texas Conference on Computing Systems*, November 1978.

[22] Philippe Nain and Don Towsley, "Comparison of Hybrid Minimum Laxity/First-in-First-out Scheduling Policies for Real-Time Multiprocessors", COINS Technical Report 90-33, University of Massachusetts at Amherst, May 1, 1990.

[23] Philippe Nain and Don Towsley, "Properties of the ML(n) Policy for Scheduling Jobs with Real-Time Constraints", *Proceedings of the 29th Conference on Decision and Control*, Honolulu, Hawaii, December 1990. pp 915-920.

[24] E. I. Organick, "The Multics System: An Examination of Its Structure", Cambridge, Mass.: MIT Press, 1972.

[25] Shivendra S. Panwar, Don Towsley and Jack K. Wolf, " Optimal Scheduling Policies for a Class of Queues with Customer Deadlines", *Journal of the ACM*, Volume 35, Number 4, October 1988.

[26] Jon M. Peha, Fouad A. Tobagi, "Evaluation of Scheduling Algorithms for Integrated-Services Packet-Switched Networks", Technical Report: CSL-TR-90-447, Computer Systems Laboratory, Stanford University, September, 1990.

[27] Jon M. Peha, Fouad A. Tobagi, "A Cost-Based Scheduling Algorithm To Support Integrated Services", Technical Report: CSL-TR-90-448, Computer Systems Laboratory, Stanford University, September, 1990.

[28] D. M. Ritchie and K. Thompson, "The UNIX Timesharing System", *Communications of ACM*, Volume 17, Number 7, July 1974. pp 365-375.

[29] M. D. Schroeder, D. D. Clark and J. H. Saltzer, "The Multics Design Project", *Proceedings of the 6th ACM Symposium on Operating System Principles*, November 1978. pp43-56.

[30] W. K. Shih, J. W. S. Liu and J. Y.Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints", to be published in *SIAM Journal of Computing* , July 1991.

[31] Barbara Simons, "A Fast Algorithm for Single Processor Scheduling", *Proceedings of 19th Annual IEEE Symposium on Foundations of Computer Science*, Syracuse, NY, October 1978. pp 246-252.

[32] John A. Stankovic, "Misconceptions About Real-Time Computing", *IEEE Computer*, Volumn 21, Number 10, October 1988.

[33] John A. Stankovic and Krithi Ramamritham, "Tutorial: Hard Real-Time Systems", New York: Computer Society Press of the IEEE, 1988.

[34] Wei Zhao, Krithi Ramamritham and John A. Stankovic, " Preemptive Scheduling Under Time and Resource Constraints", *IEEE Transactions on Computers*, Volume C-36, Number 8, August 1987.

[35] Wei Zhao and John A. Stankovic, "Performance Analysis of FCFS and Improved FCFS Scheduling Algorithms for Dynamic Real-Time Computer Systems", *Proceedings of Real-Time Systems Symposium*, December 1989, pp.156-165.