APPROXIMATE MINIMUM LAXITY SCHEDULING ALGORITHMS FOR REAL-TIME SYSTEMS

P. Goli, J. Kurose, D. Towsley
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 90-88

Approximate Minimum Laxity Scheduling Algorithms for Real-Time Systems [1]

Praveen Goli[2], James Kurose[3], Don Towsley[3]
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Minimum laxity (ML) scheduling is a well-known policy for determining the order in which jobs are permitted to access a resource in a soft real-time system. In such systems, a job must either begin or complete service (depending on the type of time constraint) within some specified amount of time after its arrival to the system or be lost. Previously-developed queueing theoretic models have shown that ML scheduling is optimal for a large class of queueing systems in the sense of minimizing the fraction of jobs that are lost (and hence not served) due to exceeding their time constraints. One possible drawback of ML scheduling is that the identity of the job with the closest deadline (minimum laxity) must be determined at each scheduling point – a potentially expensive run-time cost, especially when the number of queued jobs is large. In this paper we present four scheduling disciplines that approximate the behavior of ML scheduling and enjoy the advantage of having a run-time cost which is independent of the number of queued customers. Simulation results show that the best of these scheduling algorithms performs to within 5% of ML over a wide range of traffic loads and deadline distributions.

[1]This research was supported in part by the Office of Naval Research under contract N00014-87-K-0796 and an equipment grant from the National Science Foundation CER-DCR-8500332

[2]Dept. of Electrical and Computer Engineering

[3]Dept. of Computer and Information Science

1 Introduction

Over the last several years there has been an increased interest in the design and analysis of real-time computer systems [7,8]. The workload in these systems consists of jobs having real-time constraints. Time constraints come in several forms. Two of the most common types of time constraints are: *laxity*, the maximum time a job can wait before it begins service, and *deadline*, the latest time by which a job must complete service. In a *soft* real-time system, a job must either be served within some specified amount of time after its arrival to the system or be lost. An important research issue in the design of soft real-time systems is the development of scheduling algorithms which minimize the fraction of customers lost due to their exceeding their time constraint.

In this paper we present and examine the performance of several *approximate* implementations of the *minimum laxity (ML) scheduling discipline.* Under the exact ML scheduling discipline [4,5,6,1] that job whose time constraint is closest to expiring is selected for service; a job whose constraint expires is lost and leaves the system without receiving service. We are particularly interested in the ML policy since it has recently [4,5] been shown to be optimal (in the sense of minimizing the long-term, steady-state fraction of jobs that are lost) over all work-conserving non-preemptive policies for the continuous-time $G/M/c+G$ and $M/G/1+G$ queues and their discrete-time counterparts. Here the last G indicates that laxities can have a general distribution; we also note that while an individual job's initial laxity is assumed to be known by the system, a job's service time is assumed to be unknown.

In many real-time systems, the scheduling algorithm operates on-line and hence must be computationally efficient. One potential drawback of ML scheduling is that the identity of the job with the closest deadline (minimum laxity) must be determined at each scheduling point – a potentially expensive run-time cost, especially when the number of queued jobs is large. For example, if jobs are maintained in a list structure, finding the minimum laxity job in a non-laxity-ordered list or maintaining a sorted list according to laxity are both $O(m)$, when there are m jobs queued; if a dictionary-like structure is used to queue jobs, the time to maintain the data structure is $O(\log(m))$. In this paper we present, and examine the performance of, four scheduling disciplines that approximate the behavior of ML scheduling and enjoy the advantage of having a run-time cost which is *independent* of the number of queued customers. As we will see, our simulation results show that the best of these scheduling algorithms performs to within 5% of ML over a wide range of traffic loads and deadline distributions.

Two recent papers [2,9] have also addressed the issue of approximate implementation of ML scheduling. In [2], a policy known as ML(n) is proposed which divides the overall queue into two queues. The first queue, which directly feeds the server, can hold a maximum of n jobs, and jobs enter the server from this queue according to ML scheduling. If the number of jobs in the first queue equals n, then arriving jobs are stored in a second queue of unbounded size and

are admitted from the second queue into the first queue in a FCFS manner as space becomes available. For the cases studied in that paper, the performance of ML(n) approaches that of exact ML scheduling for values of $n = 2, 3$.

In [9], a variant of FCFS scheduling is proposed in which the laxity of an arriving customer is compared with the queued customer with the largest laxity (which is always maintained in the last position in the queue). If the arriving customer has a larger laxity, it joins the end of the queue; otherwise it joins the queue as the next-to-last customer. Customers which are not the last on the list maintain their relative order and move towards the front of the queue as other customers are served (from the front of the queue). The loss characteristics of this policy are studied analytically for the $M/M/1 + M$ system and are generally shown to be much better than pure FCFS service and (for the cases studied) sometimes approach the performance of exact ML scheduling.

Interestingly enough, the variant of FCFS studied in [9] is shown to exhibit identical behavior to ML(2) studied in [2]. By identical behavior, it is meant that departures and deadline misses occur at the same times under both policies. This equivalence result is given in [3] in more generality than is described here.

In this paper, we propose four variants of the ML(n) scheduling policy and compare their performance with that of ML(n). Our simulation results show that the best of the four policies provides 20–25% improvement over ML(n) and performs within 5% of the exact ML policy over a wide range of traffic loads and laxity distributions. The run-time cost of all proposed policies is still $O(1)$.

The remainder of the paper is organized as follows. Section 2 contains a description of the system and the four scheduling policies to be studied. Section 3 discusses the simulation results and compares the performance of various scheduling policies. Finally, Section 4 summarizes the contributions of this paper.

2 Variants of ML(n)

We consider a uniprocessor system (the scheduling policies can be applied equally well in a multiprocessor system) in which jobs arriving to this system have a service time requirement which is assumed to be unknown by the scheduler. In addition, each arriving job has an associated initial laxity indicating the maximum time it can wait before beginning service. A job is considered to be lost and is discarded (without receiving service) if it cannot begin execution before its time constraint expires.

Any number of different policies could potentially be used to schedule the jobs. Among these

policies, it has been shown that the minimum laxity scheduling policy (ML) maximizes the fraction of jobs that begin service within their time constraints from the class of policies that are non-idling [4,5]. Here ML is the non-preemptive policy that always schedules the job with the smallest laxity, i.e., the job whose time constraint is closest to expiring.

Before describing the four scheduling algorithms to be considered in this paper, let us first examine ML(n) [2] in more detail. As discussed above and as shown in Figure 1, this policy divides the overall queue into two queues, $Q1$ and $Q2$ where $Q1$ can hold at most n jobs. If the total number of jobs waiting for service is less than or equal to n, they are all held in $Q1$. Jobs in $Q1$ are scheduled according to ML. However, if the number of jobs exceeds n, then an arriving job is placed at the *end* of $Q2$. At a service completion instant, the job with minimum laxity among all jobs queued in $Q1$ is scheduled for service. Thus, if the jobs in $Q1$ are sorted according to laxities, at the time of a departure, the job at the head of $Q1$ is next scheduled for service. When the scheduler moves a job from $Q1$ to the server, it also moves a job from the *front* of $Q2$ to $Q1$. In summary then, $Q1$ is an ML queue of size n and $Q2$ is an FCFS queue of unbounded size and $Q2$ feeds $Q1$. Note that ML(1) is same as FCFS and ML(∞) is equivalent to exact ML.

We consider the four variants of the ML(n) policy shown in Figure 1. In each of these policies, as in the ML(n) policy described above, the overall queue is divided into two queues. At the time of a service completion, a job with minimum laxity among the jobs in the ML queue (i.e., in $Q1$) enters service. At the time of an arrival, if there are less than n jobs in $Q1$, the arriving job immediately enters the ML queue. If there are more than n jobs, the new arrival is handled differently by each of the four policies considered below.

Policy 1: If the number of jobs in $Q1$ (the ML queue) is equal to n, the laxity of the new arrival is compared with the laxity of the n-th job in $Q1$. If the laxity of the new arrival is greater, it is placed at the *end* of $Q2$; otherwise, it is placed at the *end* of $Q1$ and the n-th job is placed at the *front* of $Q2$. The positional *order* of jobs in $Q1$ is maintained in the order of entry to $Q1$ and thus the n-th job is always that job which entered $Q1$ last. However, when the server selects a job for service, it chooses the job with minimum laxity among the jobs in $Q1$ (ignoring those jobs in $Q2$ as well as the positional order of jobs in $Q1$).

Policy 2: This policy is similar to Policy 1. If the laxity of a new arrival is greater than that of the n-th job in $Q1$, the new arrival is placed at the *end* of $Q2$; otherwise the new arrival is placed at the *end* of the ML queue and, unlike Policy 1, the n-th job is placed at the *end* of $Q2$.

Policy 3: In this policy, the laxity of the new arrival is compared with the laxity of the job with the *maximum laxity* among the n jobs in $Q1$. If the laxity of the new arrival is

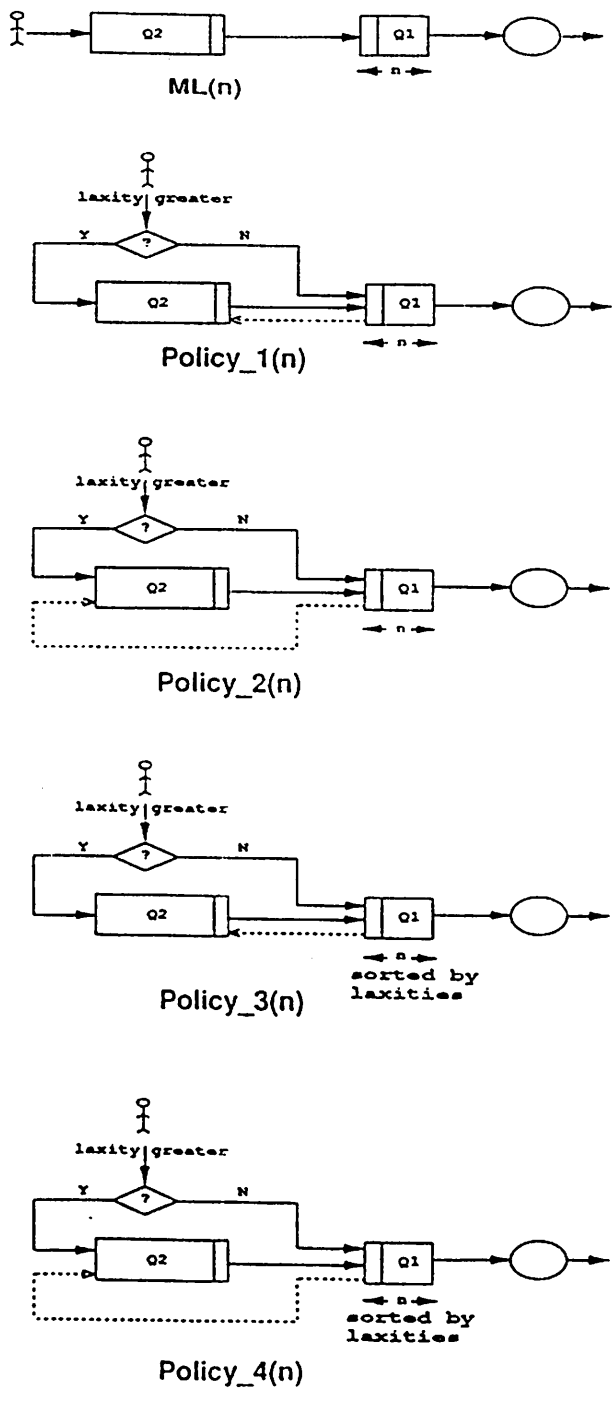

Figure 1: Approximate Implementations of ML(n) scheduling

greater, the new arrival is placed at the *end* of $Q2$; otherwise, it is placed in the position of the job with maximum laxity among the jobs in $Q1$ and the job with maximum laxity in $Q1$ is placed at the *front* of $Q2$.

Policy 4: This policy is similar to Policy 3. If the laxity of the new arrival is smaller than that of the maximum laxity customer in $Q1$, the new arrival replaces that job in $Q1$. However, unlike Policy 3, the job with maximum laxity among those jobs in $Q1$ is then placed at the *end* of $Q2$.

3 Simulation Results

In this section, we compare and discuss the performance of the various scheduling policies described in the previous section. The performance results are obtained through simulation. For each simulation point, we simulated a uniprocessor system for 100,000 service time units. Independent random number generators are used for generating various random numbers during the simulation. However, the *same set* of random numbers is used for comparing the results of different policies (and thus the same set of job arrival times, service times and laxities are used for evaluating the performance of each policy). The regenerative method is used to compute the 95% confidence intervals. The ratio of the width of the confidence interval to its corresponding point estimate was typically less than 0.1. In our simulations, the mean interarrival time $1/\lambda$ and the mean laxity $1/l$ are normalized with respect to the mean service time $1/\mu$. Unless stated otherwise, the interarrival times, service times and laxities are exponentially distributed. We study the performance of these policies over a wide range of mean laxities and arrival rates.

Figure 2 presents the simulation results comparing FCFS, ML(n) for $n = 3$ and $n = 5$, and ML scheduling policies. We plot the fraction of jobs lost versus the normalized mean laxity. Curves are shown for $\lambda = 0.75$ and $\lambda = 1.00$. It is clear that when the mean laxity is low, all policies provide approximately the same performance. However, as the mean laxity is increased, the difference between the optimal policy ML and ML(n) policy increases. It is also worth noting that the difference between FCFS and ML(n) policies also increases. The difference is quite significant when the system is highly loaded. When laxities are tight, a large fraction of the jobs miss their deadlines and many scheduling policies exhibit similar performance since the order of jobs in the queue becomes unimportant. We conjecture that there are two related reasons for the difference in performance as the mean laxity increases. First, with relaxed laxities, more jobs with unexpired laxities will typically be present in the queue waiting for service. Hence the queue length will typically be long. Since a new arrival is placed at the end of the queue in both the FCFS and ML(n) policies, if the new arrival has a very small laxity, it will typically miss its deadline whenever the queue length is large. Second, since the laxities are relaxed, more customers in the queue receive service before their laxity expires (indeed, the main effect

of relaxing the laxities is a reduced loss fraction). This implies that a new arrival has to wait longer before it receives its service. A new arrival with a small laxity which would typically miss the deadline with FCFS and ML(n) policies would get served using ML.

In Figure 3, we plot the loss fraction versus the arrival rate for two different mean laxities. We see here that the difference between ML, ML(n) and FCFS increase with increasing λ up to the point at which the system becomes overloaded (i.e., the arrival rate of jobs exceeds the systems capacity to process jobs). As the system becomes heavily loaded, once again all policies perform approximately the same. Arguments similar to the ones in the previous paragraph explain this phenomenon. Note that in the low load case, there is little difference in performance among the four policies shown since there is seldom more than a small number of jobs present in the system, and hence the order of jobs in the queue is relatively unimportant. As the load increases beyond capacity, the performance of the policies begins to converge and indeed we expect that the loss fraction of all policies will approach 1 as the arrival rate approaches infinity.

As noted above, we conjecture that a new arrival with a "small" laxity is one of the main causes for the increase in the difference in performance. This was the motivation for us to consider Policy 1. Policy 1 provides an opportunity to insert the new arrival into $Q1$, thus providing it an opportunity to receive service before its deadline expires. Recall that we provide this chance for the new arrival, by inserting it as the n^{th} job in $Q1$ and pushing the n^{th} job to the front position of $Q2$ if the laxity of the new arrival is smaller. From Figure 4, it is seen that this provides close to 15% improvement over ML(n), when $n = 3$ and $\lambda = 1.0$. Further (although not shown in Figure 4), our results indicate that Policy 1 with $n = 1$ performs better than ML(n) with $n = 3$.

Even though Policy 1 performs better than ML(n), the performance is still noticeably poorer than ML. A closer look at this policy suggests that pushing the n^{th} job to the $(n+1)^{st}$ position of the overall queue (i.e., the front of the FCFS queue) is part of the reason for the degraded performance with respect to ML. Consider the situation in which the n^{th} job has a *large* laxity. Upon the arrival of a new job (with a relatively smaller laxity), it is placed in front of the n^{th} job. Hence the job that was previously at the end of $Q1$ is now at the front of $Q2$. In the absence of departures, any future arrival will compare its laxity to that of the new job. If the laxity of this future arrival is larger, it is placed at the *end* of $Q2$, which feeds $Q1$ in FCFS manner. When the job with *large* laxity that was displaced by the first arrival moves from its position in the front of $Q2$ back to the end of $Q1$, it may again be pushed back by a future arrival. As a consequence, the other jobs which always remain behind this job with large-laxity will remain in $Q2$ for a long period of time and as a result may miss their deadlines.

To avoid such a scenario, in Policy 2, the laxity of the new arrival is compared with the laxity of the n^{th} job, and the job with larger laxity is placed at the end of $Q2$. Informally, this manner of pushing jobs with larger laxity to the end of the FCFS queue makes $Q2$ (and thus the overall

queue as well) approximate the behavior of a minimum-laxity-ordered queue. From Figure 4, it is seen that this provides up to a 5% improvement over Policy 1 and a 19% improvement over ML(n).

Policy 3 is a variant of Policy 1; we compare the laxity of the new arrival with the laxities of all n customers in the ML queue rather than just the n^{th} customer in the ML queue. This provides approximately 6% improvement over Policy 2 and 23% improvement over ML(n). The cost is the additional overhead of comparing the new job with all n jobs. However, since n is fixed and, in the cases we have studied, can be quite small, the overhead is minimal, and will always be of constant order, regardless of the total number of jobs in the combined ML/FCFS queueing system.

Policy 4 combines the ideas of policies 2 and 3. This provides a significant improvement in performance. As shown in Figure 4, using this policy, the fraction of jobs lost is within 13% of the optimal policy when $n = 3$ and is a 28% improvement over ML(n). Figure 5 compares the performance of the above policies when $n = 5$. Policy 4 now performs within 5% of the ML policy.

In Figure 6, we plot the performance of various scheduling policies as n is varied. Plots are shown for $\lambda = 0.75$ and $\lambda = 1.0$. It is interesting to note that all polices proposed in this paper perform better than FCFS even when $n = 1$. Note also that ML(1) performs identically as FCFS, as expected. Further, it is of interest to note that Policy 4 with $n = 1$ performs better than ML(n) with $n = 4$.

In Figure 7, we plot the performance of various scheduling policies versus the normalized mean laxity for $\lambda = 0.75$ and $\lambda = 1.0$. All four policies described in this paper are found to perform better than FCFS and ML(n). Our results indicate that these observations are true for different traffic loads as well. We also note that Policy 4 performs very close to the ML policy over a wide range of mean laxity values.

For Figure 8, we have an hyper-exponential distribution for the laxities with coefficient of variation equal to 5. The performance of the various scheduling policies is very similar to what was observed with exponentially distributed laxities with the same mean. Figure 9 graphs the variation in the performance of the scheduling policies as the coefficient of variation of the laxity distribution is varied. Curves plotted are for $\lambda = 0.85$ and mean laxity equal to 20. The fraction of jobs lost increases as the coefficient of variation is increased. This is because as the coefficient of variation is increased, the fraction of jobs with smaller laxity increases. However, the relative performance of the scheduling policies remains the same.

4 Conclusions

In this paper, we described four variants of ML policy for scheduling jobs with real-time constraints. The best policy was found to be Policy 4, in which a new arrival compares its laxity with the laxities of n jobs in the ML queue and the job with larger laxity is placed at the end of the FCFS queue; this policy was shown to provide significant improvement over FCFS and ML(n) policies. In many circumstances, Policy 4 was also shown to be identical or very close in performance to that of the ML policy, which is known to be optimal. The overhead of these scheduling policies is independent of the number of jobs in the system. This desirable feature, combined with the near optimal performance of this policy, makes it suitable for on-line use in real-time systems.

References

[1] P. Bhattacharya and A. Ephremides, "Optimal Scheduling with Strict Deadlines," *IEEE Trans. on Auto. Control,* Vol. 34, No. 7 (July 1989), pp 721-728.

[2] J. Hong, X. Tan, D. Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System," *IEEE Transactions on Computers*, Vol. 38, No. 12, (December 1989), 1736–1744.

[3] P. Nain, D. Towsley, "Properties of the ML(n) Policy for Scheduling Jobs with Real-Time Constraints," to appear in *Proc. 29-th IEEE Control and Decision Conf.*, Dec. 1990.

[4] S. Panwar, D. Towsley, J.K. Wolf, "Optimal Scheduling Policies for a Class of Queues with Customer Deadlines Until the Beginning of Service," *J. of the ACM,* Vol. 35, No. 4 (Oct. 1988), pp. 832-844.

[5] S.S. Panwar and D. Towsley, "On the Optimality of the STE Rue for Multiple Server Queues that Serve Customers with Deadlines," *COINS Technical Report 88-81*, Dept. of Computer & Information Science, Univ. of Massachusetts, July 1988.

[6] H. Saito, "Optimal Queueing Discipline for Real-Time Traffic at ATM Switching Nodes," *Proc. of the IEICE of Japan,* Sept. 1988, pp. 47-54.

[7] J. Stankovic, "Misconceptions About Real-Time Computing," *IEEE Computer,* Oct. 1988, 10-19.

[8] J. Stankovic and K. Ramamritham, *Hard Real Time Systems,* IEEE Press, Piscataway, NJ, 1988.

[9] W. Zhao and J.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, 156–165.

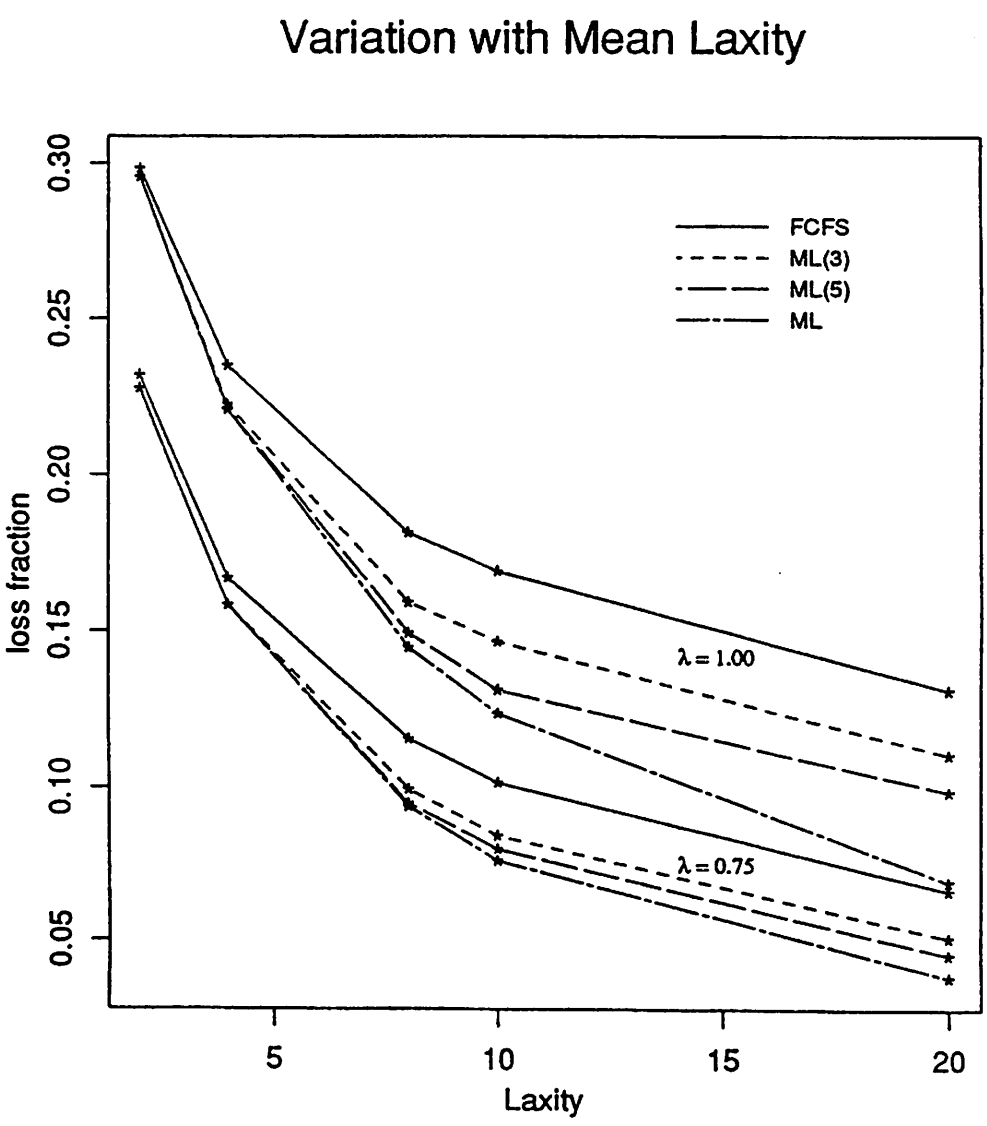

Figure 2: Variation with Laxity of ML(n)

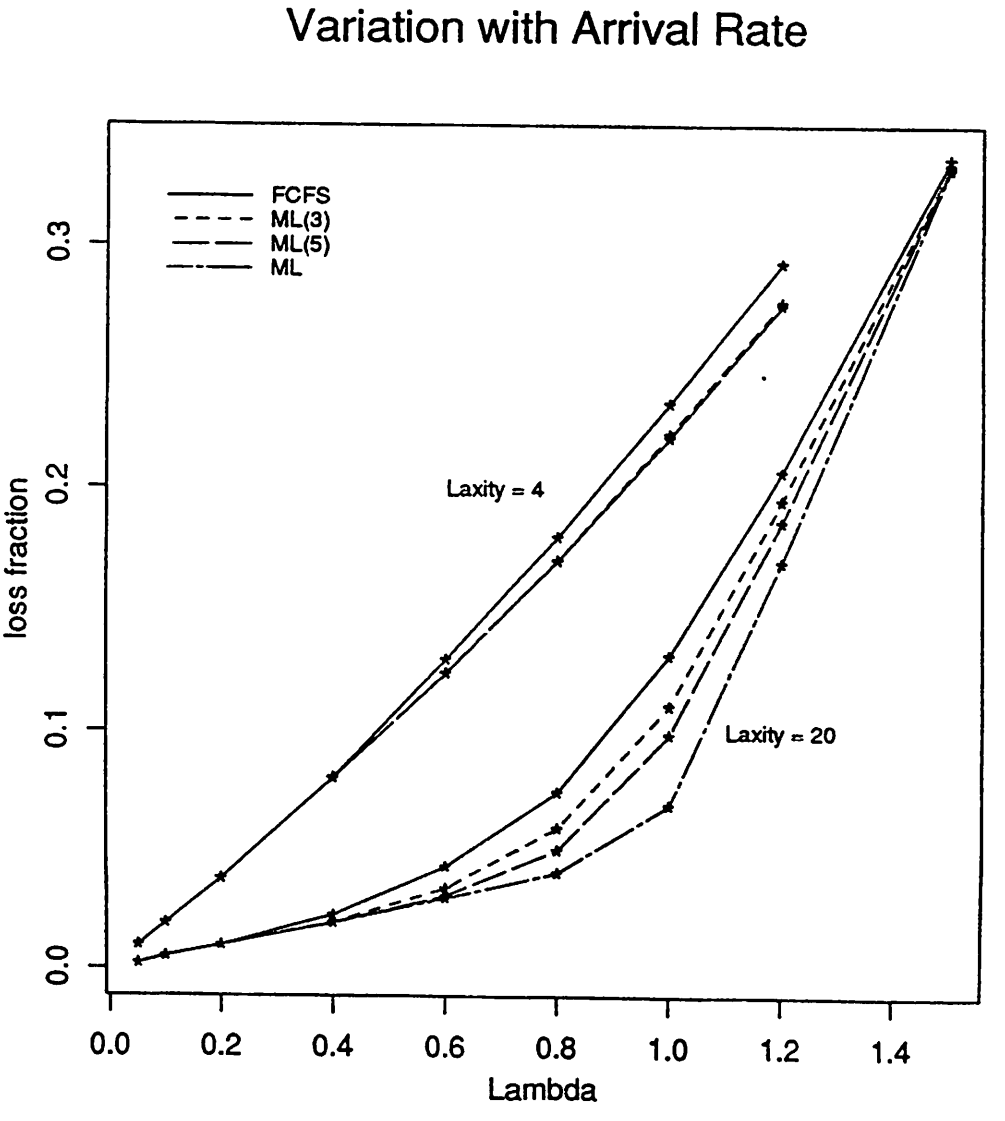

Figure 3: Variation with Lambda of ML(n)

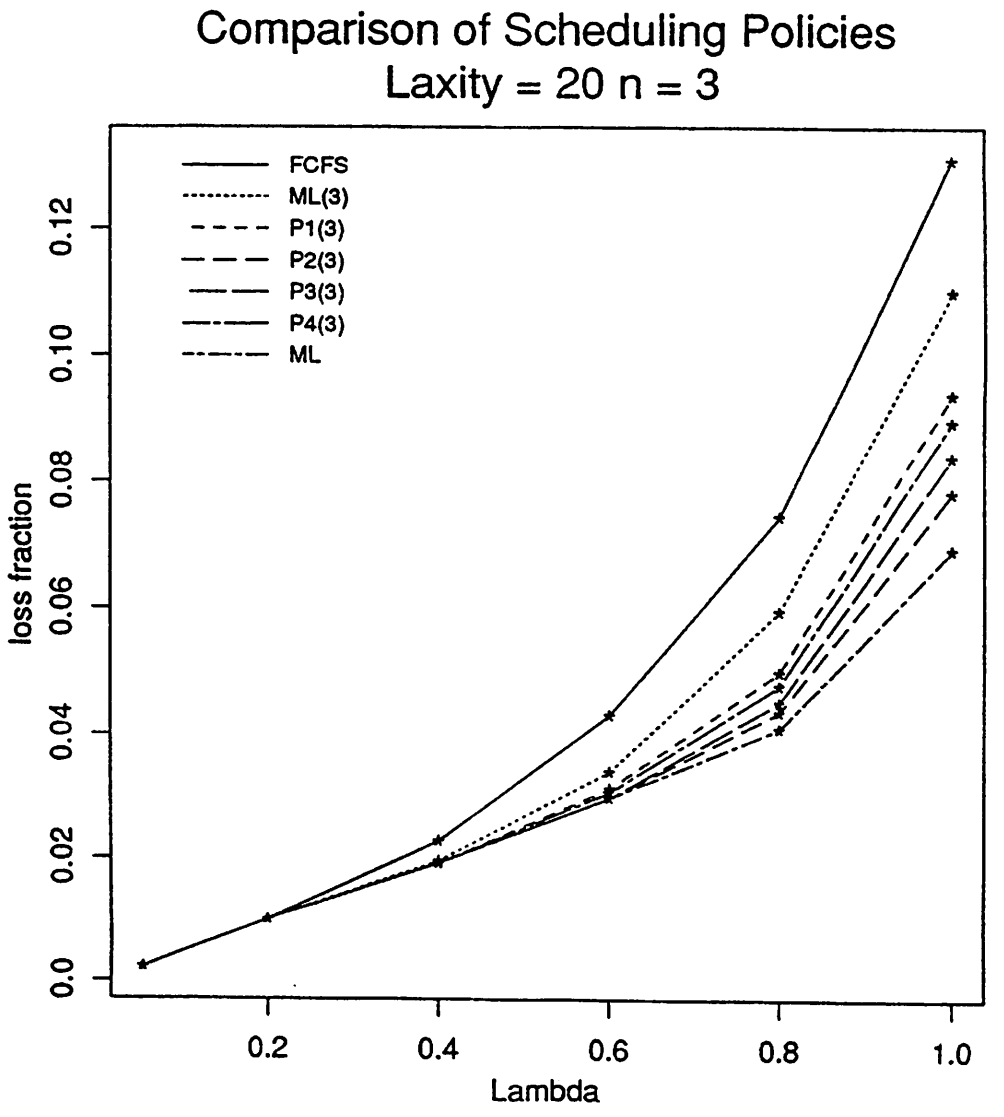

Figure 4: Comparison of Scheduling Policies for $n = 3$

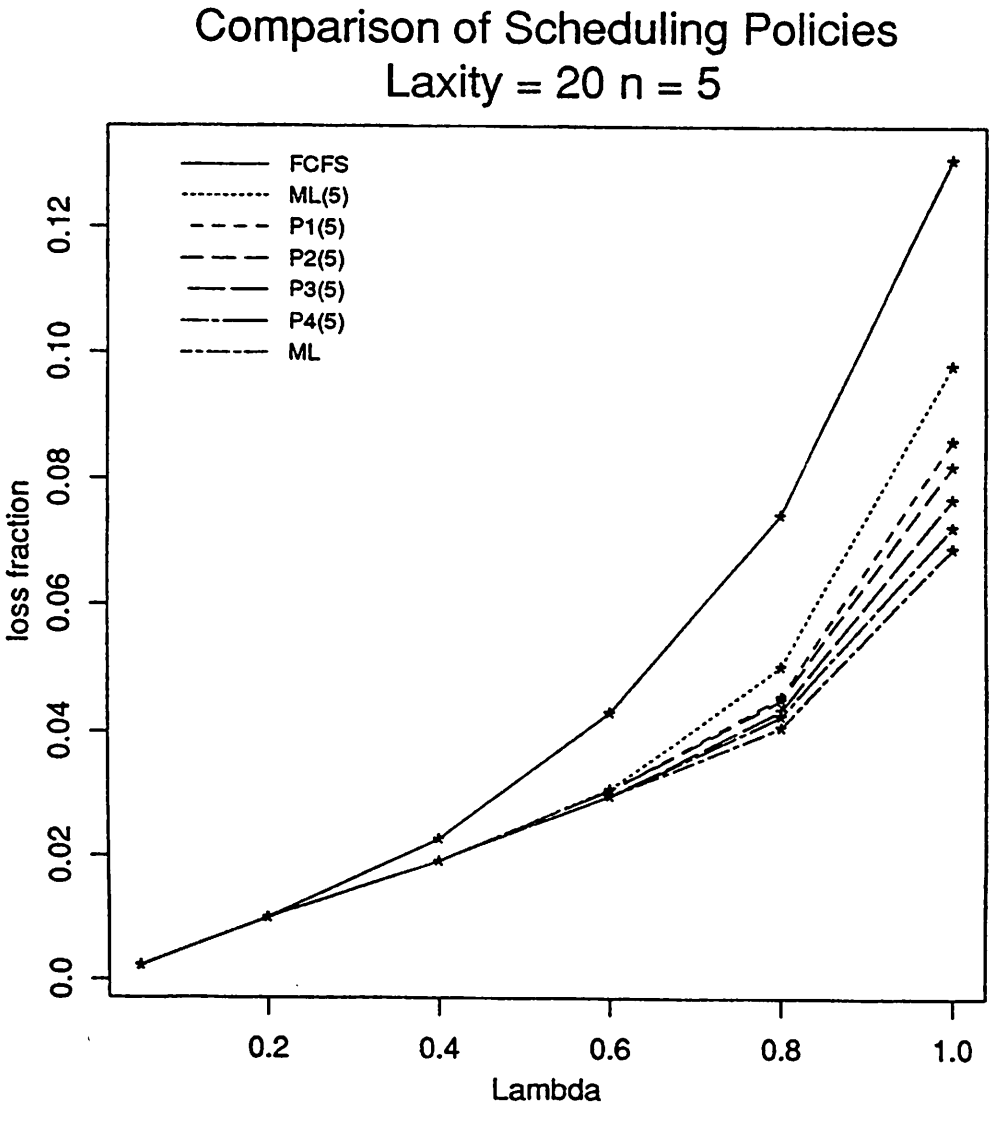

Figure 5: Comparison of Scheduling Policies for $n = 5$

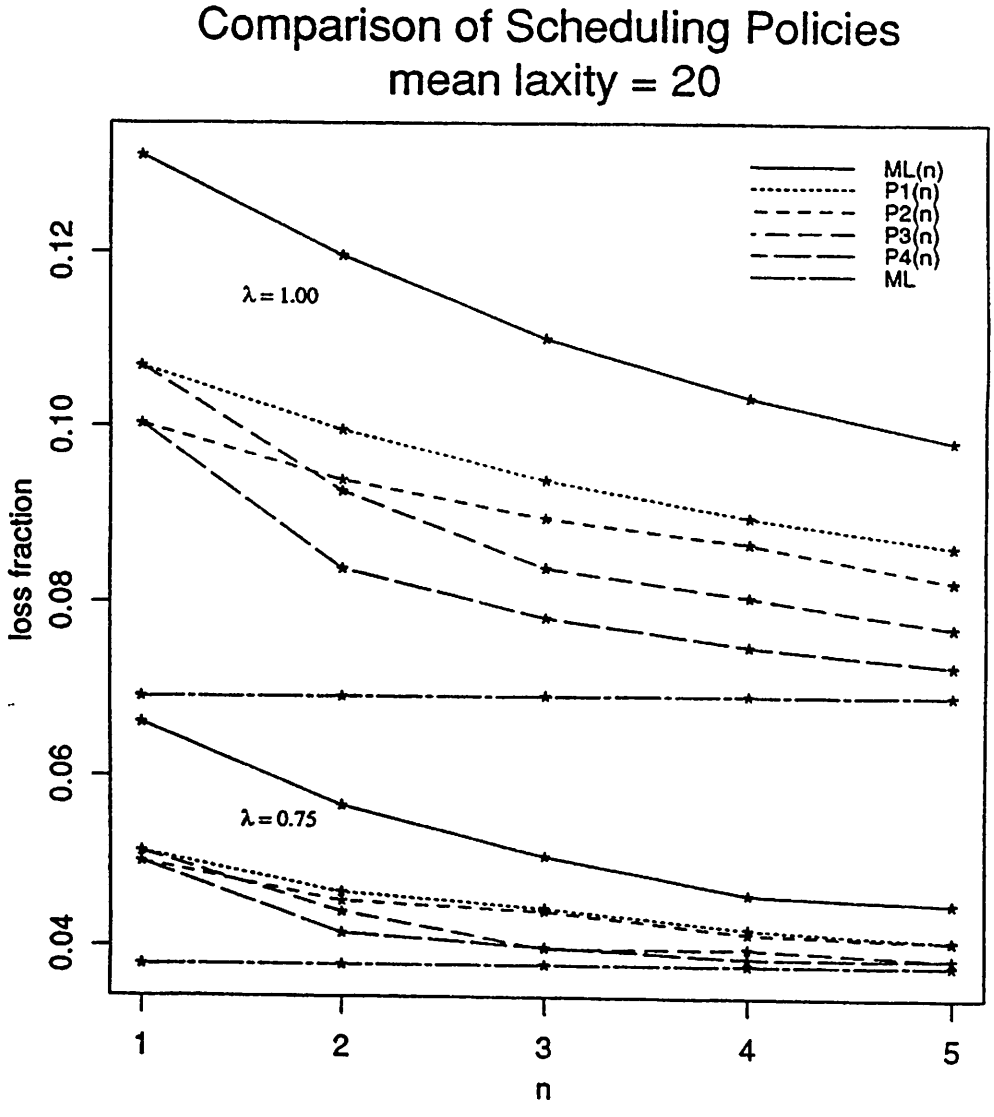

Figure 6: Variation with Parameter n

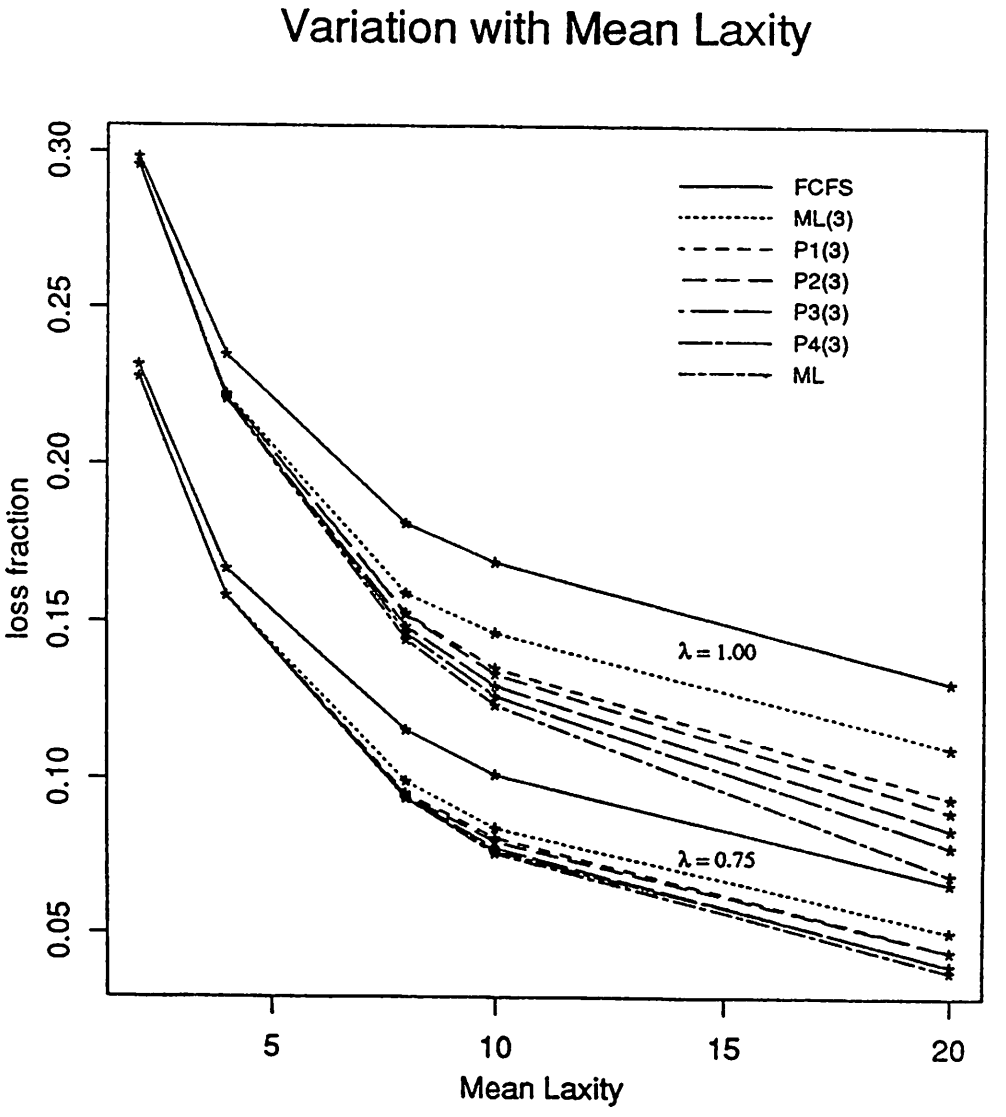

Figure 7: Variation with Mean Laxity

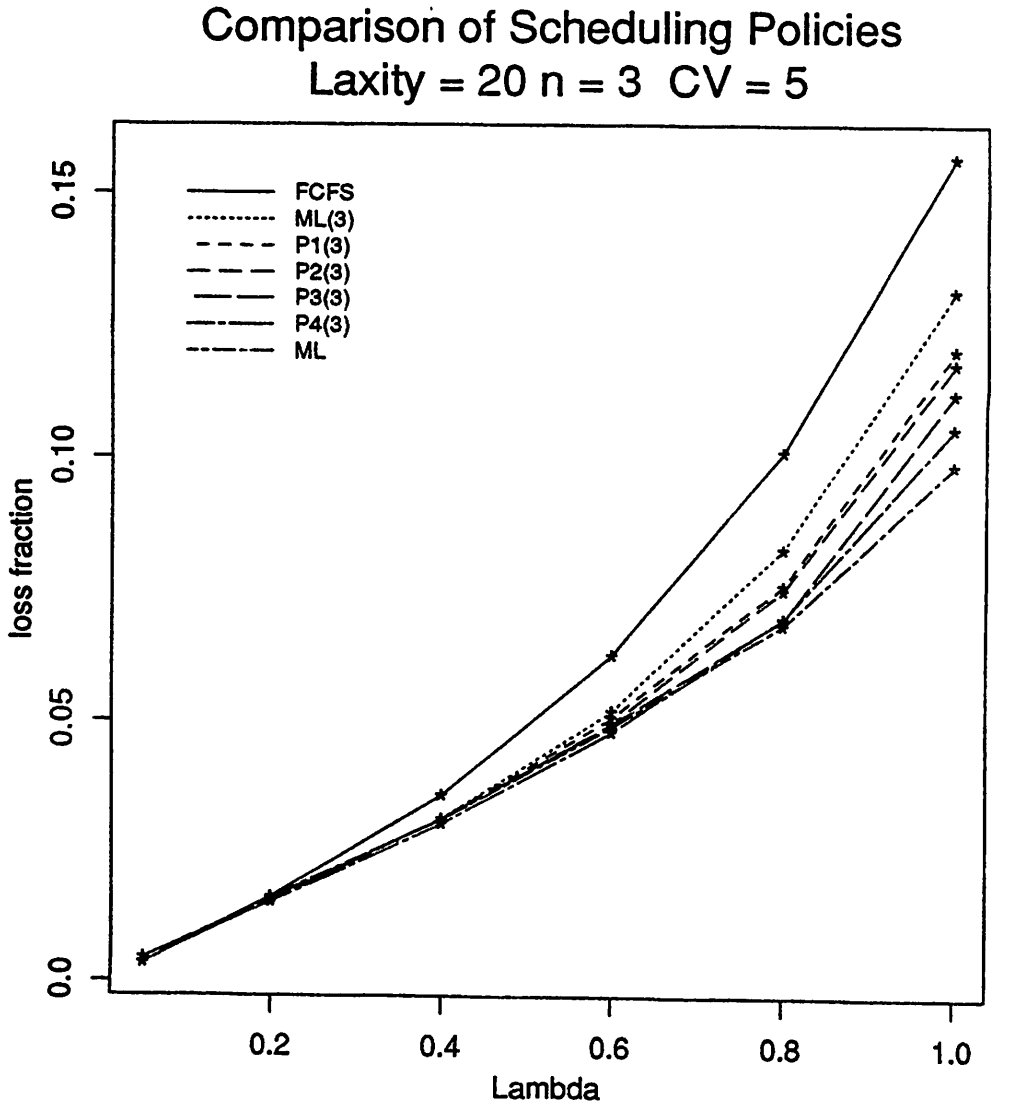

Figure 8: Comparison of Scheduling Policies for CV = 5

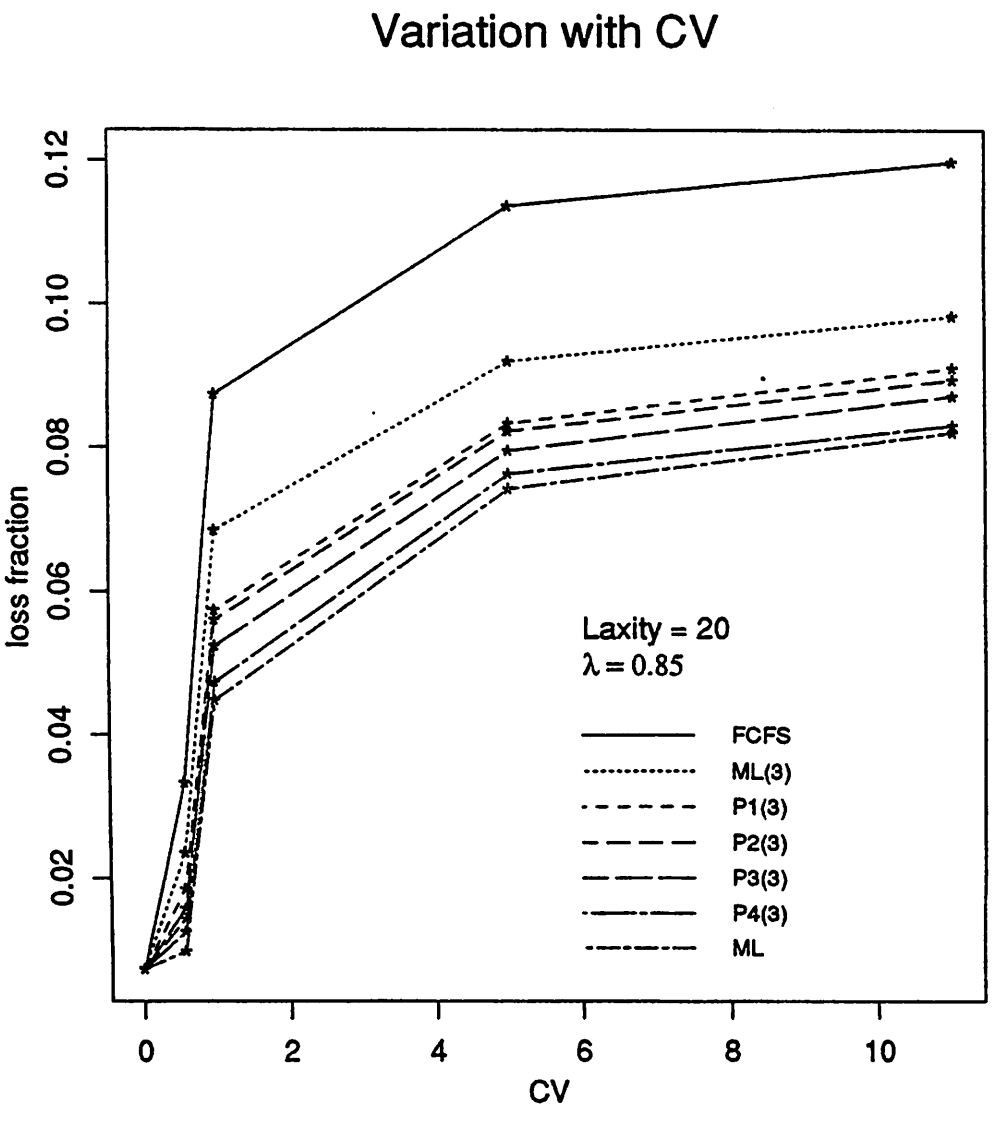

Figure 9: Variation with Coefficient of Variation