A Comparison of the Processor Sharing and First Come First Serve Policies for Scheduling Fork-Join Jobs in Multiprocessors*

COINS Technical Report 87-83

August 26, 1987

D. Towsley
Dept. Computer and Information Sciences
Univ. of Massachusetts
Amherst, MA 01003

C. G. Rommel
Department of Electrical Engineering
Univ. of Massachusetts
Amherst, MA 01003

J. A. Stankovic
Dept. Computer and Information Sciences
Univ. of Massachusetts
Amherst, MA 01003

Abstract

In this paper a model of a shared memory multiprocessor that executes *fork-join* parallel programs as a bulk arrival $M^X/M/c$ queueing system is developed. Here a fork-join job is one that consists of a set of X tasks. All of the tasks arrive simultaneously to the system and the job is assumed to complete when the last task completes. We develop tight upper and lower bounds for the mean response time of such programs when the scheduling discipline is processor sharing under the assumptions of exponential task service times and a Poisson job arrival process. We study two processor sharing policies, one called *task scheduling* processor sharing and the other called *job scheduling* processor sharing. The first policy schedules tasks independently of each other and allows parallel execution, whereas the second policy schedules entire jobs as a unit and

*This work was supported in part by the National Science Foundation under grant MCS-8104203 and by RADC under contract RI-44896X and WM23216-4-19.

thereby does not allow parallel execution of an individual program. We find that the job scheduling policy exhibits better performance than task scheduling only on systems with a small number of processors, where the system is operating at high loads and is executing programs that can sustain a large degree of parallelism. Consequently, in general, task scheduling outperforms job scheduling. We also compare the performance of the processor sharing policy with first come first serve. We find that first come first serve exhibits better performance over a wide range of systems. The paper also studies the performance of processor sharing and first come first serve with two classes of jobs, and when a specific number of processors is statically assigned to each of these classes.

1 Introduction

With the advent of multiprocessors[Ost86] and programming languages that support parallel programming, (e.g., Concurrent Pascal [Han75], CSP[Hoa85], and Ada [Pyl81]) there is increasing interest in modeling the performance of parallel programs. In this paper, we evaluate the performance of a particular type of parallel program, a *fork-join* job, on a multiprocessor consisting of identical processors when the service discipline is processor sharing. In our model a fork-join job is composed of a set of tasks each of which can be scheduled independently of the others at any processor. All tasks in a given job arrive simultaneously to the system. The job completes when the last task completes.

The performance of parallel programs such as fork-join jobs is significantly affected by the choice of policy that is used to schedule tasks. We analyze the performance of a processor sharing (PS) policy that schedules *tasks* of a job independently of each other. We refer to this policy as *task scheduling PS*, TS-PS. We compare the performance of this TS-PS policy to that of a second PS policy that schedules entire *jobs* (as a single unit) independently of each other. We refer to this policy as *job scheduling PS*, JS-PS. The TS-PS policy is unaware that jobs exist whereas the JS-PS policy is unaware that tasks exist. We also compare the performance of TS-PS and JS-PS to the first come first serve (FCFS) policy. In these comparisons we consider different numbers of processors, sizes of fork-join jobs, multiple classes, and dedicated assignments of the processors of the multiprocessor to the different classes.

In the course of our study, we develop upper and lower bounds on the mean fork-join job response times under TS-PS. These bounds are generally very tight and we approximate the mean job response time by taking the average of the two bounds. Analyses of the other two policies, JS-PS and FCFS have already appeared in the literature ([RTS87,NTT87]).

We make the following observations from our study.

- FCFS provides better performance than TS-PS or JS-PS for a wide range of workloads and number of processors. It appears that the advantages that FCFS has over PS in

single processor systems carries over to multiprocessors executing parallel programs. This carries the implication that one should choose large quantum sizes for round robin policies operating on multiprocessors.

- TS-PS performs better than JS-PS most of the time. However, if the number of processors is small, the degree of parallelism high, and the processor utilization is high, JS-PS can perform better. This same phenomenon was observed on single processors in an earlier study, [RTS87].

- It may be useful to partition the processors in a multiprocessor into separate pools to handle different classes of jobs rather than having the jobs share the processors. We observe that jobs requiring the least amount of computation can benefit from such a partition.

In the remainder of this section we briefly review earlier work and outline the remainder of this paper. Processor-sharing has been addressed in the literature in several ways since its introduction [Kle64]. A survey of processor-sharing results may be found in [Kle76]. An exact analysis of the TS-PS policy operating on a *single* processor was performed by Rommel, et al. [RTS87]. Unfortunately, the approach used in that paper does not extend to multiple processors. This study first demonstrated that job scheduling can give better performance than task scheduling. In addition, there is a growing literature on fork-join queueing systems [BM85,BMT87,NT85]. Although these referenced papers analyze fork-join jobs, their analysis differs from that studied in this paper in that processors are allocated to specific tasks prior to execution. We are interested in systems where processors can be ***dynamically*** allocated to different tasks.

The format of this paper is as follows. We describe the queueing system under consideration in Section 2. Section 3 contains expressions for the upper and lower bounds on the mean response time for the TS-PS scheduling policy along with an approximate analysis of that policy. This is followed by our numerical results in Section 4. Finally, in Section 5 we summarize the results of the paper.

2 Model Description

We consider a system of c identical processors that serve a single queue. Fork-join jobs enter the system according to a Poisson process with parameter λ. A fork-join job consists of X tasks that can be processed independently of each other where X is a random variable (r.v.) with probability distribution $\alpha_i = P[X = i]$, $i = 1, 2, \cdots$. The service time required by a task is assumed to be an exponential r.v. with parameter μ and is independent of the service requirements of all other tasks.

We are interested in the steady state behavior of this system when operating under the task scheduling processor sharing (TS-PS) and the job scheduling processor sharing (JS-PS) policies. As described in section 1, TS-PS is a policy that performs processor sharing at the task level and JS-PS is a policy that performs processor sharing at the job level. Thus, if the system contains two jobs, one with one task, the other with three tasks, then TS-PS provides an equal amount of service to each task and is capable of utilizing four processors. In this same example JS-PS sees two jobs, one whose service time is that of a single task, the other whose service time is the sum of the service times of the three tasks. JS-PS provides equal service to the two jobs and is only able to utilize two processors.

In both cases, we focus on the response time of a random job, i.e., the interval of time measured from the arrival of a job until the service completion of the *last task* associated with that job. The system can be visualized as a queue for tasks, c servers, and a waiting area for tasks that have completed service but are awaiting the completion of the last task associated with the job (Figure 1). This last queue is sometimes referred to as the *synchronization queue*. We denote this response time as T.

3 Analysis

In this section we concern ourselves with obtaining the mean response time $E[T]$ under both TS-PS and JS-PS. We consider JS-PS first as it is the simplest to analyze.

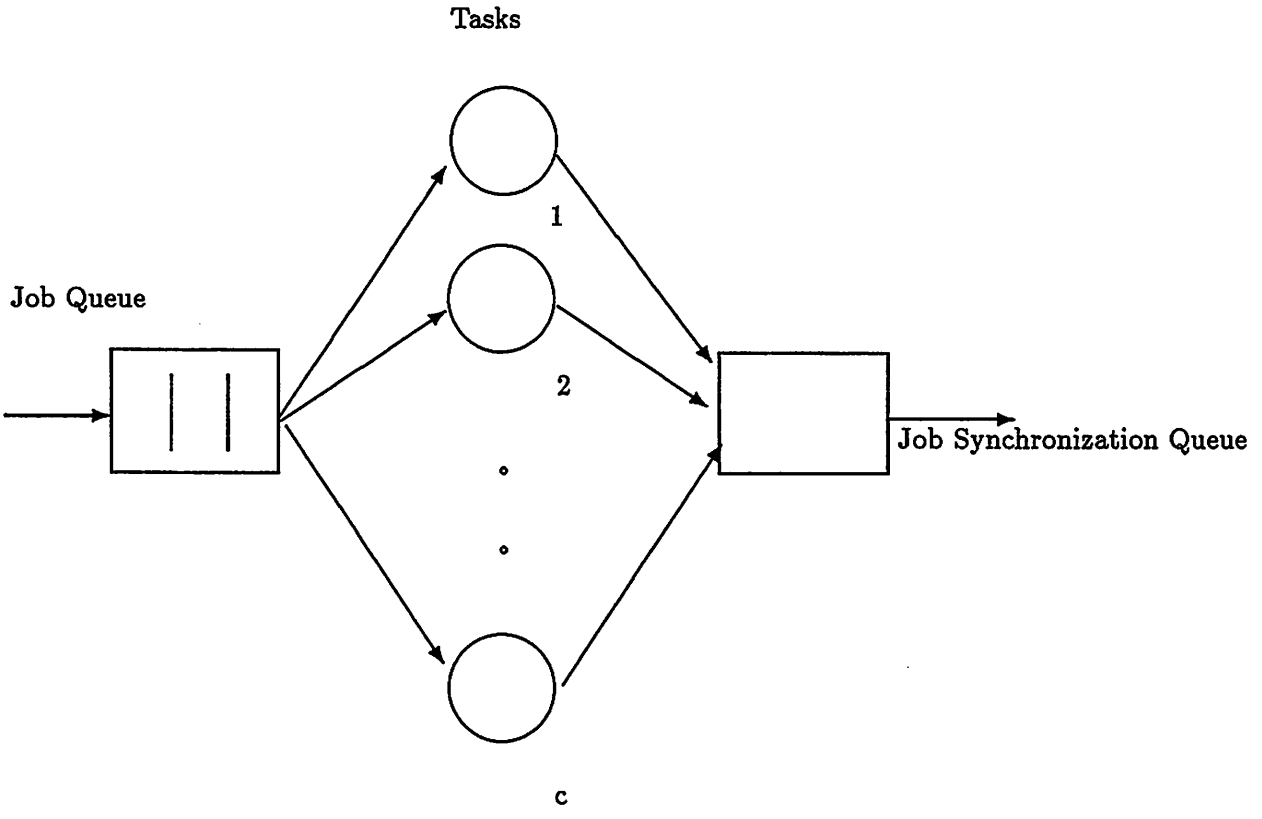


Figure 1: System Model

3.1 The JS-PS policy

Let L denote the number of jobs in the system under JS-PS. The distribution of L is identical to the queue length distribution of an $M/M/c$ system with arrival rate λ and average service time $E[X]/\mu$. Consequently, the average response time, $E[T]$, is ([All78])

$$E[T] = \frac{u^c(E[X]/\mu)}{cc![u^c/c! + (1-u/c)\sum_{n=0}^{c-1} u^n/n!](1-u/c)} + E[X]/\mu. \tag{1}$$

where $u = \lambda E[X]/\mu$. $E[T] = E[L]/\lambda$.

3.2 The TS-PS policy

To analyze the TS-PS policy, consider the delay that a randomly selected job incurs. Let J denote this job. Let N be a r.v. that denotes the number of tasks in the system at the time that J arrives. Let $\pi_n = P[N = n]$, $n = 0, 1, \cdots$ denote the stationary distribution of N. Let $t_{i,n}$ denote the mean response time of J conditioned on the event that J consists of i tasks and that the system contains $N = n$ tasks at the time of its arrival, i.e. $t_{i,n} = E[T|X = i, N = n]$. We can write the following expression for the mean job response time,

$$E[T|X = i] = \sum_{n=0}^{\infty} \pi_n t_{i,n}, \quad i = 1, \cdots \tag{2}$$

Removal of conditioning on the number of tasks in J yields

$$E[T] = \sum_{i=1}^{\infty} \alpha_i E[T|X = i]. \tag{3}$$

As described above, the number of tasks in the system is described by a Markov process. Fortunately, the behavior of this Markov process is independent of the policy used to schedule tasks so long as the policy does not schedule jobs based on service time information. Consequently, the distribution of N is identical to that for a bulk arrival $M^X/M/c$ system that schedules tasks in a FCFS manner. Expressions for the queue length distribution for this system can be found in earlier papers [CT83,Yao85,NTT87] and are omitted here.

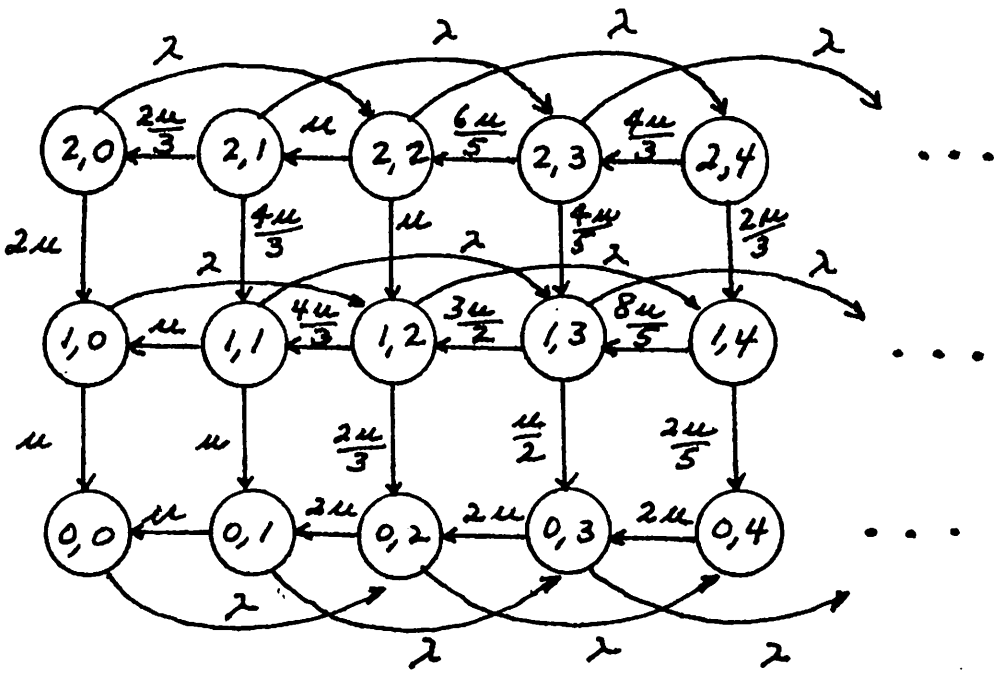


Figure 2: State diagram for the exact system when jobs consist of 2 tasks.

We focus on the conditional expectations $t_{i,n}$. We define a Markov Chain with state (I_t, M_t) with infinitesimal generator $\mathbf{Q}$ where I_t is the number of tasks remaining in J at time t after J is introduced at time 0, M_t is the number of tasks in the system at time t that are not part of J, and $\mathbf{Q} = [q_{(i,n),(l,m)}]$ where

$$q_{(i,n),(l,m)} = \begin{cases} \frac{i}{i+n}\mu_{i+n}, & l = i-1,\ n = m, \\ \frac{n}{i+n}\mu_{i+n}, & l = i,\ m = n-1, \\ \lambda\alpha_{m-n}, & i = l,\ m > n, \\ -(\lambda + \mu_{i+n}), & i = l,\ m = n, \\ 0 & ,otherwise \end{cases} \tag{4}$$

where

$$\mu_k = \begin{cases} k\mu, & k = 1, \cdots, c \\ c\mu, & k = c+1, \cdots. \end{cases}$$

The resulting chain is transient. Figure 2 illustrates the associated state diagram when all jobs consist of exactly 2 tasks.

It follows from the definition of $\mathbf{Q}$ that $t_{i,n}$ satisfies

$$
\begin{aligned}
t_{1,0} &= \frac{1}{\lambda+\mu_1} + \frac{\lambda}{\lambda+\mu_1}\sum_{k=1}^{\infty}\alpha_k t_{1,k}, \\
t_{1,n} &= \frac{1}{\lambda+\mu_{n+1}} + \frac{\lambda}{\lambda+\mu_{n+1}}\sum_{k=1}^{\infty}\alpha_k t_{1,n+k} + \frac{n\mu_{n+1}/(n+1)}{\lambda+\mu_{n+1}}t_{1,n-1}, \quad n=1,\cdots, \\
t_{i,0} &= \frac{1}{\lambda+\mu_i} + \frac{\lambda}{\lambda+\mu_i}\sum_{k=1}^{\infty}\alpha_k t_{i,k} + \frac{\mu_i}{\lambda+\mu_i}t_{i-1,0}, \quad i=2,\cdots, \\
t_{i,n} &= \frac{1}{\lambda+\mu_{i+n}} + \frac{\lambda}{\lambda+\mu_{i+n}}\sum_{k=1}^{\infty}\alpha_k t_{i,n+k} \\
&\quad \frac{n\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}}t_{i,n-1} + \frac{i\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}}t_{i-1,n}, 1=2,\cdots; n=1,\cdots. \qquad (5)
\end{aligned}
$$

Consider the last expression, $t_{i,n}$. The first term is the average time that the system spends in state (i,n). The second term is the contribution to $t_{i,n}$ due to an arrival. The third and fourth terms are the contributions due to a departure of a task belonging to J and a task not belonging to J, respectively.

We are unable to obtain a closed form solution to equation (5). As there are a countably infinite number of unknown variables $t_{i,n}$, $i=1,\cdots; n=0,\cdots$, it is impossible to obtain exact numerical values for these quantities. Consequently, the remainder of this section is concerned with developing upper and lower bounds on the conditional expectations $t_{i,n}$. These can be used to obtain upper and lower bounds for $E[T|X=i]$, $i=1,\cdots$. We treat each in turn.

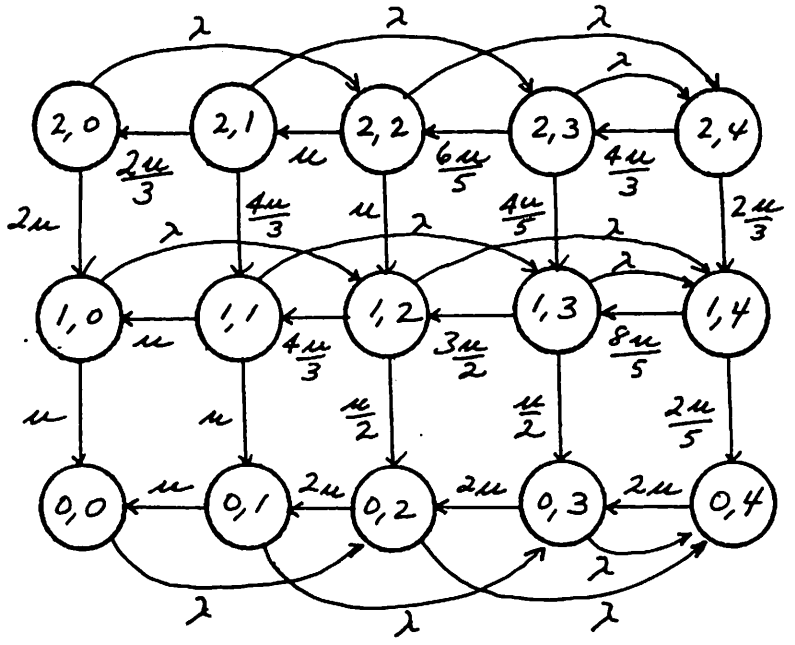


Figure 3: The state diagram associated with the lower bound, 2 tasks per job.

3.2.1 A Lower bound on $E[T|X=i]$

We study a Markov chain with state $(I_t^{(lb)}, M_t^{(lb)})$ that yields lower bounds $t_{i,n}^{(lb)}$ on $t_{i,n}$, $i = 1, \cdots, n = 0, \cdots$. This chain has infinitesimal generator $\mathbf{Q}^{(lb)} = [q_{(i,n),(l,m)}^{(lb)}]$ where

$$q_{(i,n),(l,m)}^{(lb)} = \begin{cases} \frac{i}{i+n}\mu_{i+n}, & l = i-1,\ n = m;\ 0 \le m \le B, \\ \frac{n}{i+n}\mu_{i+n}, & l = i,\ m = n-1;\ 1 \le m < B, \\ \lambda\alpha_{m-n}, & i = l,\ 0 \le n < m < B, \\ \lambda\sum_{k=B-n}^{\infty}\alpha_k, & l = i,\ m = B, 0 \le n < B, \\ -(\lambda+\mu_{i+n}), & i = l,\ m = n,\ 0 \le m < B, \\ -\mu_{i+B}, & i = l,\ m = n = B, \\ 0, & \textit{otherwise.} \end{cases} \tag{6}$$

This Markov chain corresponds to a system in which no more than B tasks not belonging to J are allowed in. Consequently, this modified system has fewer tasks that do not belong to J than the original system. The response time of J will be less in this system. Figure 3 illustrates the state diagram for this Markov chain when each job consists of exactly 2 tasks.

The conditional expectations, $t_{i,n}^{(lb)}$ satisfy

$$
\begin{aligned}
t_{1,0}^{(lb)} &= \frac{1}{\lambda+\mu_1} + \frac{\lambda}{\lambda+\mu_1}\left(\sum_{k=1}^{B}\alpha_k t_{1,k}^{(lb)} + \sum_{k=B+1}^{\infty}\alpha_k t_{1,B}^{(lb)}\right), \\
t_{1,n}^{(lb)} &= \frac{1}{\lambda+\mu_{n+1}} + \frac{n\mu_{n+1}/(n+1)}{\lambda+\mu_{n+1}} t_{1,n-1}^{(lb)} + \\
&\quad \frac{\lambda}{\lambda+\mu_{n+1}}\left(\sum_{k=1}^{B-n}\alpha_k t_{1,n+k}^{(lb)} + \sum_{k=B-n+1}^{\infty}\alpha_k t_{1,B}^{(lb)}\right), \quad n=1,\cdots,B, \\
t_{i,0}^{(lb)} &= \frac{1}{\lambda+\mu_i} + \frac{\mu_i}{\lambda+\mu_i} t_{i-1,0}^{(lb)} + \\
&\quad \frac{\lambda}{\lambda+\mu_i}\left(\sum_{k=1}^{B}\alpha_k t_{i,k}^{(lb)} + \sum_{k=B+1}^{\infty}\alpha_k t_{i,B}^{(lb)}\right), \quad i=2,\cdots, \\
t_{i,n}^{(lb)} &= \frac{1}{\lambda+\mu_{i+n}} + \frac{\lambda}{\lambda+\mu_{i+n}}\left(\sum_{k=1}^{B-n}\alpha_k t_{i,n+k}^{(lb)} + \sum_{k=B-n+1}^{\infty}\alpha_k t_{i,B}^{(lb)}\right) + \\
&\quad \frac{n\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}} t_{i,n-1}^{(lb)} + \frac{i\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}} t_{i-1,n}^{(lb)}, 1=2,\cdots; n=1,\cdots,B. \qquad (7)
\end{aligned}
$$

Last, $t_{i,n}$, $n > B$ is bounded from below by $t_{i,B}^{(lb)}$, i.e.,

$$
t_{i,n}^{(lb)} = t_{i,B}^{(lb)}, \quad i=1,\cdots; n=B+1,\cdots. \qquad (8)
$$

Thus we have the following lower bound on $E[T|X=i]$,

$$
E[T|X=i] \le \sum_{n=0}^{B-1}\pi_n t_{i,n}^{(lb)} + P[N \ge B] t_{i,B}^{(lb)}, \quad i=1,\cdots. \qquad (9)
$$

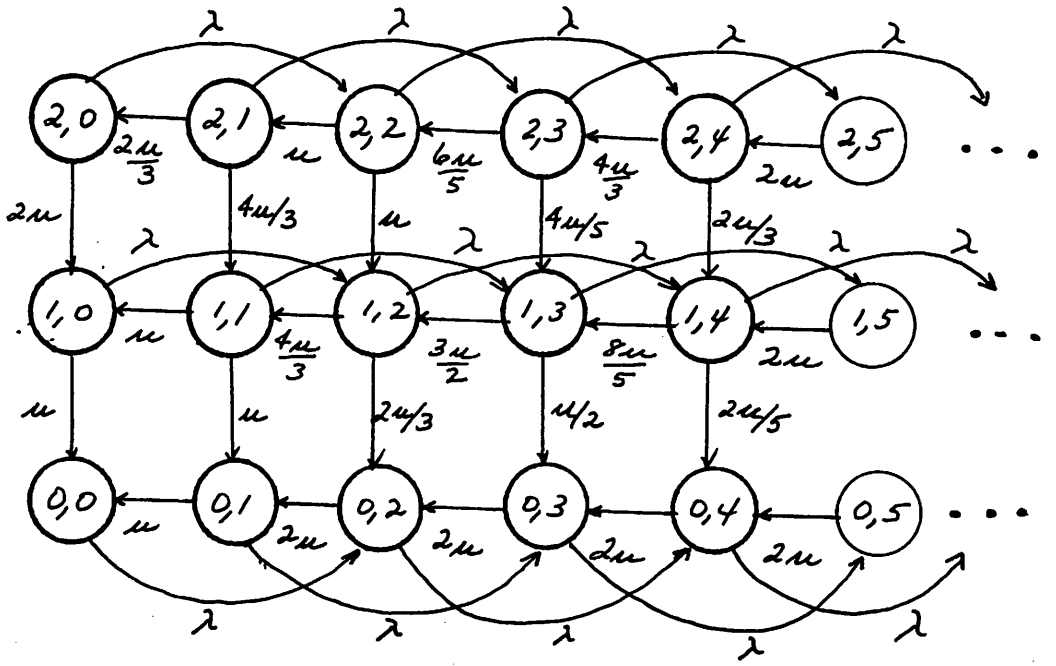


Figure 4: State diagram for upper bound, $B = 4$.

3.2.2 An upper bound on $E[T|X = i]$

We study a Markov chain with state $(I_t^{(ub)}, M_t^{(ub)})$ that yields upper bounds $t_{i,n}^{(ub)}$ on $t_{i,n}$, $i = 1, \cdots, n = 0, \cdots$. This chain has infinitesimal generator $\mathbf{Q}^{(ub)} = [q_{(i,n),(l,m)}^{(ub)}]$ where

$$q_{(i,n),(l,m)}^{(ub)} = \begin{cases} \frac{i}{i+n}\mu_{i+n}, & l = i-1,\ n = m;\ 0 \le m \le B, \\ \frac{n}{i+n}\mu_{i+n}, & l = i,\ m = n-1;\ 1 \le m < B, \\ \mu_{i+n}, & l = i,\ m = n-1,\ B \le m, \\ \lambda\alpha_{m-n}, & i = l,\ 0 \le n < m, \\ -(\lambda + \mu_{i+n}), & i = l,\ m = n,\ 0 \le m, \\ 0, & \textit{otherwise.} \end{cases} \tag{10}$$

This system behaves like the original system except when the number of tasks n not belonging to J exceeds B. In this case, the system *is not allowed to serve* J, but instead only serves the other tasks. This continues until the number of additional tasks falls to B at which point the system behaves like the original system. Figure 4 illustrates the behavior of this system when jobs contain exactly two tasks and $B = 4$.

Assume that $B \ge c$. Consider the situation where J contains i tasks and there are an

additional $j < B$ tasks in the system. Now assume that k tasks arrive and that $n+k > B$. In this case, the time during which there are $B+1$ or more additional tasks in the modified system is equal to the length of the busy period associated with a bulk arrival $M^X/M/1$ queue with rate μc that is initiated by the arrival of $n+k-B$ tasks. Consequently, we can write the following set of equations describing the expected response time of a job conditioned on the number of tasks at the time of arrival and the number of tasks in the arriving job, $t_{i,n}^{(ub)}$,

$$
\begin{aligned}
t_{1,0}^{(ub)} &= \frac{1}{\lambda+\mu_1} + \frac{\lambda}{\lambda+\mu_1}\left(\sum_{k=1}^{B}\alpha_k t_{1,k}^{(ub)} + \sum_{k=B+1}^{\infty}\alpha_k(t_{1,B}^{(ub)} + b_{k-B})\right), \\
t_{1,n}^{(ub)} &= \frac{1}{\lambda+\mu_{n+1}} + \frac{\lambda}{\lambda+\mu_{n+1}}\left(\sum_{k=1}^{B-n}\alpha_k t_{1,n+k}^{(ub)} + \sum_{k=B-n+1}^{\infty}\alpha_k(t_{1,B}^{(ub)} + b_{n+k-B})\right) + \\
&\quad \frac{n\mu_{n+1}/(n+1)}{\lambda+\mu_{n+1}} t_{1,n-1}^{(ub)}, \quad n = 1,\cdots,B, \\
t_{i,0}^{(ub)} &= \frac{1}{\lambda+\mu_i} + \frac{\lambda}{\lambda+\mu_i}\left(\sum_{k=1}^{\infty}\alpha_k t_{i,k}^{(ub)} + \sum_{k=B+1}^{\infty}\alpha_k(t_{i,B}^{(ub)} + b_{k-B})\right) + \\
&\quad \frac{\mu_i}{\lambda+\mu_i} t_{i-1,0}^{(ub)}, \quad i = 2,\cdots, \\
t_{i,n}^{(ub)} &= \frac{1}{\lambda+\mu_{i+n}} + \frac{\lambda}{\lambda+\mu_{i+n}}\left(\sum_{k=1}^{B-n}\alpha_k t_{i,n+k}^{(ub)} + \sum_{k=B-n+1}^{\infty}\alpha_k(t_{i,B}^{(ub)} + b_{n+k-B})\right) + \\
&\quad \frac{n\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}} t_{i,n-1}^{(ub)} + \frac{i\mu_{i+n}/(i+n)}{\lambda+\mu_{i+n}} t_{i-1,n}^{(ub)}, 1 = 2,\cdots; n = 1,\cdots,B. \qquad (11)
\end{aligned}
$$

where b_l is the average length of a busy period of an $M^X/M/1$ queue with arrival rate λ and service rate μc that is started by the arrival of i tasks. The value of b_l, $l = 1,\cdots$ is ([GH76])

$$b_l = \frac{l}{\mu c}\left(1 + \frac{\lambda E[X]}{(\mu c - \lambda E[X])}\right).$$

Last, $t_{i,n}$, $n > B$ can be bounded by $t_{i,n}^{(ub)}$ given by

$$t_{i,n}^{(ub)} = t_{i,B}^{(ub)} + b_{n-B}, \quad n = B+1, \cdots. \tag{12}$$

These expressions can be substituted into the following relation to obtain an upper bound on $E[T|X=i]$,

$$\begin{aligned} E[T|X=i] &\leq \sum_{n=0}^{\infty} \pi_n t_{i,n}^{(ub)}, \quad i = 1, \cdots, \\ &\leq \sum_{n=0}^{B} \pi_n t_{i,n}^{(ub)} + (1 - P[N \leq B]) t_{i,B}^{(ub)} \\ &\quad + b_1 \left(E[N] - B(1 - P[N \leq B]) - \sum_{n=1}^{B} n\pi_n \right), \quad i = 1, \cdots. \end{aligned} \tag{13}$$

3.2.3 Approximate analysis of TS-PS

Let $T^{(lb)}$ and $T^{(ub)}$ denote the r.v.'s defined in the preceding sections that bound T from below and above. We use the following approximation for $E[T|X=i]$,

$$E[T^{(approx)}|X=i] = (E[T^{(lb)}|X=i] + E[T^{(ub)}|X=i])/2. \tag{14}$$

The accuracy of this approximation is high when the system load is low and/or when the parameter B takes a large value. We explore both of these effects in Table 1. Here we evaluate the upper and lower bounds on $E[T]$ for a system of 16 processors that process fork-join jobs containing exactly 16 tasks. The bounds are tabulated for different values of the processor utilization, $\rho = \lambda/\mu$ and for different values of B. We observe that sufficient accuracy is possible for processor utilizations up to .9 provided $B = 350$. In this case, the maximum error incurred by the approximation is 3.6% at $\rho = .9$ and less than .05% for $\rho \leq .8$. We shall use $B = 350$ throughout our studies.

ρ	B=50		B=100		B=200		B=350	
	lower	upper	lower	upper	lower	upper	lower	upper
.1	55.03	55.03	55.03	55.03	55.03	55.03	55.03	55.03
.2	56.68	56.41	56.68	56.68	56.68	56.68	56.68	56.68
.3	59.21	59.41	59.32	59.32	59.32	59.32	59.32	59.32
.4	62.95	63.91	63.47	63.47	63.47	63.47	63.47	63.47
.5	68.18	71.78	70.02	70.16	70.10	70.10	70.10	70.10
.6	75.17	87.01	80.60	81.70	81.17	81.17	81.17	81.17
.7	84.14	121.64	97.97	105.37	101.50	101.28	101.38	101.38
.8	95.20	225.95	126.68	175.40	142.94	148.56	144.88	145.04
.9	108.14	792.05	173.09	601.33	242.70	389.11	271.46	291.96

Table 1: Approximation Analysis

4 Comparison of Scheduling Policies

In this section we compare the performance of TS-PS, JS-PS, and FCFS. Specifically, we compare the mean job response time for different processor utilizations as we vary the number of processors and the job size. We also compare the performance of TS-PS and FCFS on a system that serves two classes of jobs: edit jobs and batch jobs. Edit jobs are assumed to consist of a single task whereas batch jobs consist of many tasks. Last, we consider the effects of partitioning the processors into two sets; one to serve edit jobs exclusively and the other to serve batch jobs exclusively. For this last study, we compare the performance of the partitioned system under TS-PS to one where the processors are available to all jobs under TS-PS.

4.1 Comparison of TS-PS, JS-PS, and FCFS

In this section we compare the TS-PS, JS-PS, and FCFS policies as a function of the processor utilization. In Figure 5 we plot the ratio of response times of TS-PS to JS-PS, and TS-PS to FCFS for two workloads as a function of the processor utilization, ρ. The workloads consist of jobs with a constant number of tasks that is equal to the number of processors, i.e., $X = 8, c = 8$ and $X = 16, c = 16$. The average task service time is taken to be $1/c$. From this figure we observe that FCFS provides uniformly better response over

the two PS policies for all processor utilizations. Furthermore, TS-PS gives lower response times than JS-PS for all processor utilizations less than 0.9. This is due to the fact that TS-PS takes advantage of the parallelism inherent in the fork-join job. We shall observe, however, TSPS is not always better than JSPS for very high utilizations in Section 4.2. The better performance exhibited by FCFS is due to the fact that TS-PS penalizes larger jobs, while no such penalty exists for FCFS (a more detailed discussion of this penalty phenomenon is given in the next section).

We also tested a workload consisting of two classes of jobs: edit jobs and batch jobs. Edit jobs consist of a single task and batch jobs consist of 16 tasks. Let f denote the fraction of jobs that are edit jobs. We considered three mixes, $f = .5, .95, .99$ operating on a system containing $c = 16$ processors. Figure 6 illustrates ratios of the mean job response time of TS-PS to FCFS as a function of the processor utilization ρ. We observe that the FCFS policy exhibits the best performance everywhere except when the utilization is high and the fraction of edit jobs is high ($f = .95, .99$). In this region TS-PS provides slightly lower response times.

This workload, ($f = .95, .99$), was chosen so as to increase the variability in the service job service times in an attempt to illustrate a setting in which TS-PS outperforms FCFS. It is surprising that the difference is so small. This is an indication that FCFS is a more robust policy on multiprocessors that execute parallel programs than it is in a system where jobs are executed serially.

From this figure we can also observe that TS-PS provides only slightly better service to edit jobs than FCFS, but significantly worse service to batch jobs.

4.2 Dependence on Number of Servers

In the last section we observed that TS-PS provides better performance than JS-PS for all of the examples. This appears to be at odds with observations that we noted in an earlier study [RTS87] on the performance of TS-PS and JS-PS in a *single processor* system. In a single processor system, JS-PS was shown to be uniformly better than TS-PS. This is due

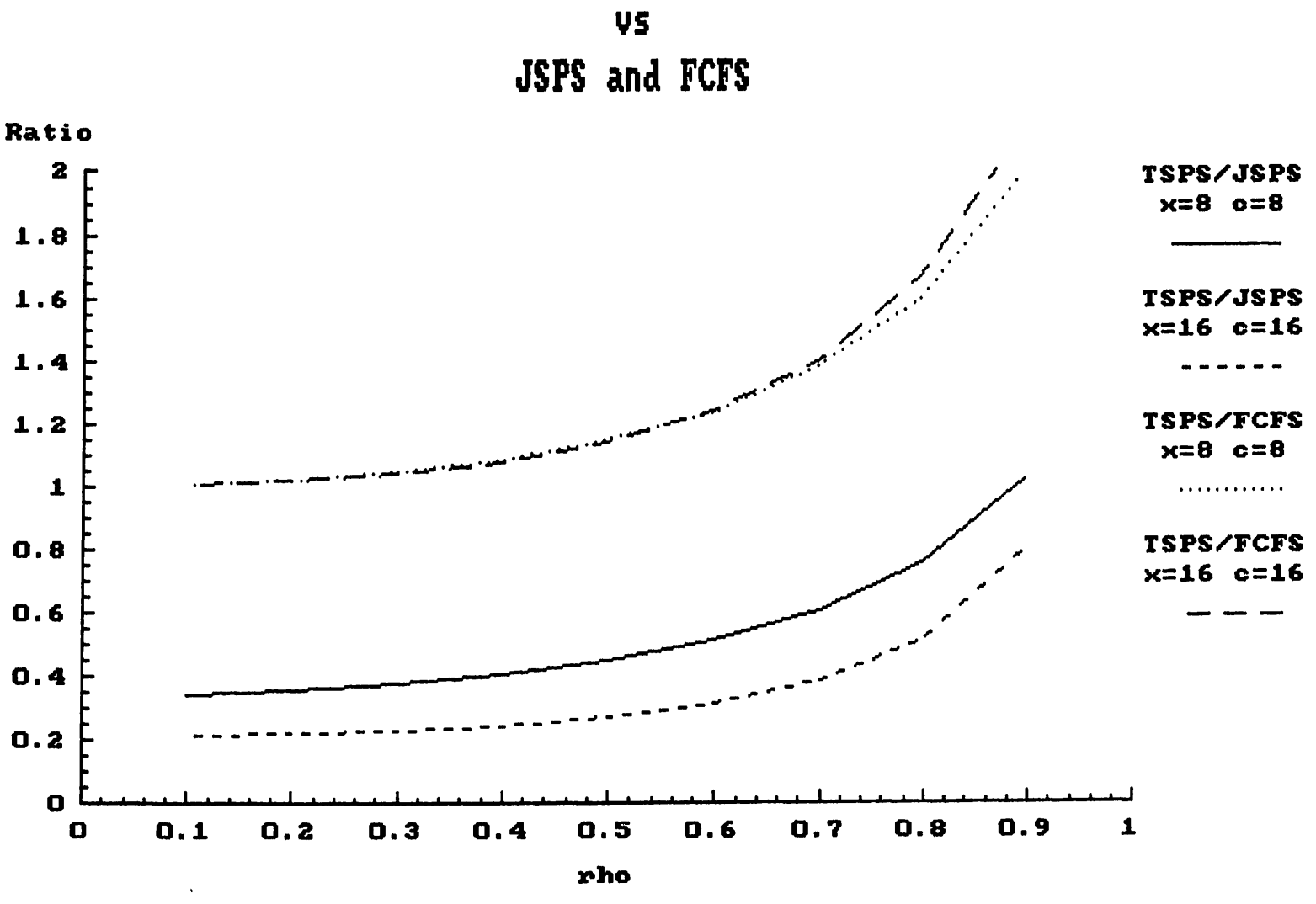
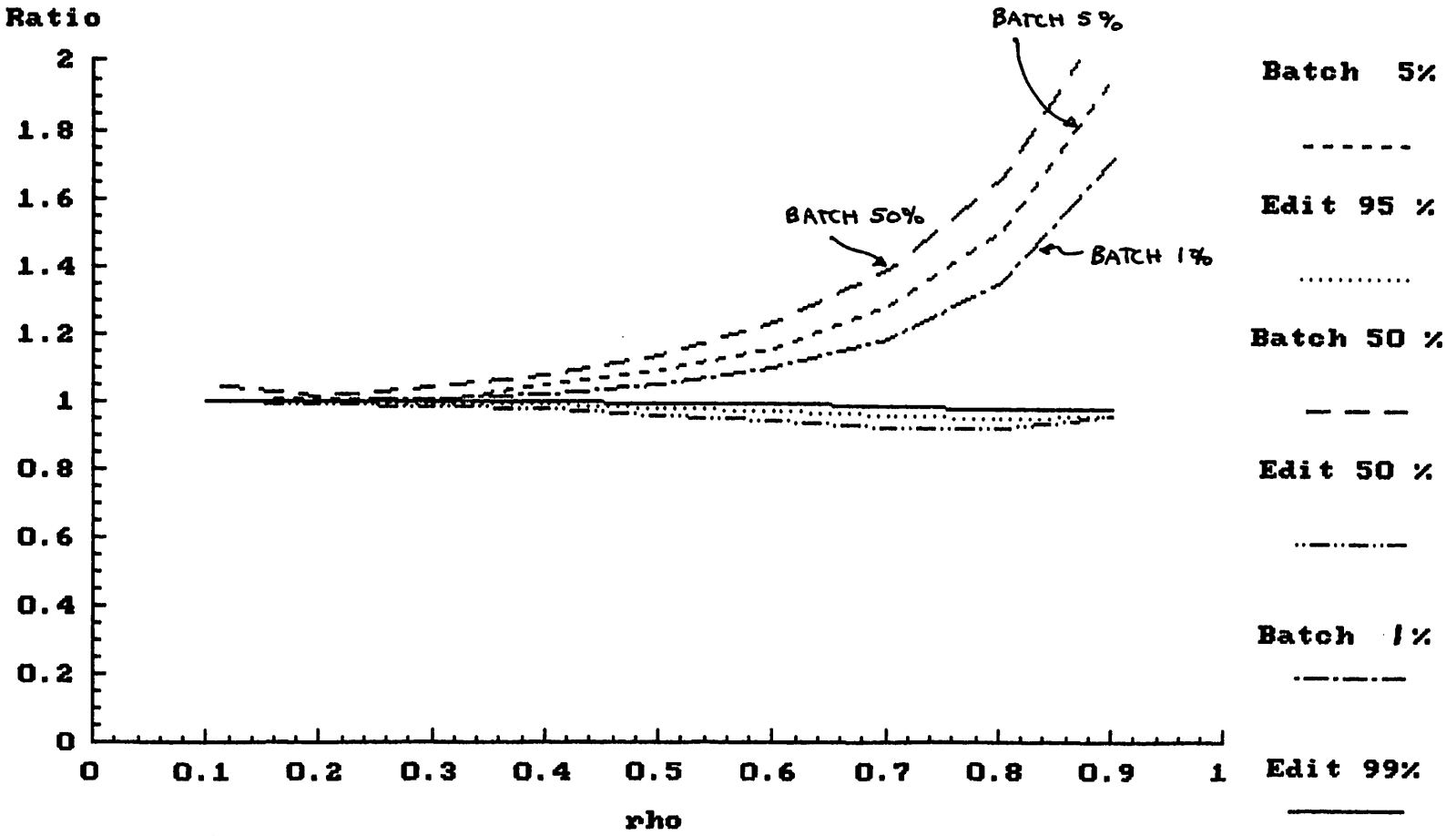


FIGURE 5

B=350

Batch and Edit Jobs with TSPS vs FCFS



c=16
x=16
B=350

FIGURE 6

to the fact that in such a system there is no possibility for parallelism and the following occurs. Assume that there are 2 jobs, one with 1 task and one with 9 tasks. Then TS-PS gives the job with 9 tasks, 9/10 of the processor, and the job with a single task only 1/10 of the processor. However, JS-PS would give each job 1/2 of the processor. So on a single processor, TS-PS penalizes jobs with a small number of tasks. In a multiprocessor, there exists sufficient possibilities for parallelism so that this anomaly found in a single processor for TS-PS does not exist.

To study the effect of parallelism, we consider a workload of jobs consisting of 16 tasks and study the performance of TS-PS and JS-PS on systems with $c = 1, 2, 4, 8, 16, 32$ processors as a function of processor utilization. Figure 7 illustrates the results of this study plotting the response time ratios of TS-PS to JS-PS. We observe that TS-PS is always better than JS-PS in multiple processor systems when processor utilization is low. However, when the number of processors is small (< 8), there exists a utilization value, say ρ_0 such that system performance is better under JS-PS when $\rho > \rho_0$. This threshold is an increasing function of c the number of processors. This results because as the number of processors increases, the capability of sustaining parallel program execution under TS-PS increases.

4.3 Processor Partitioning

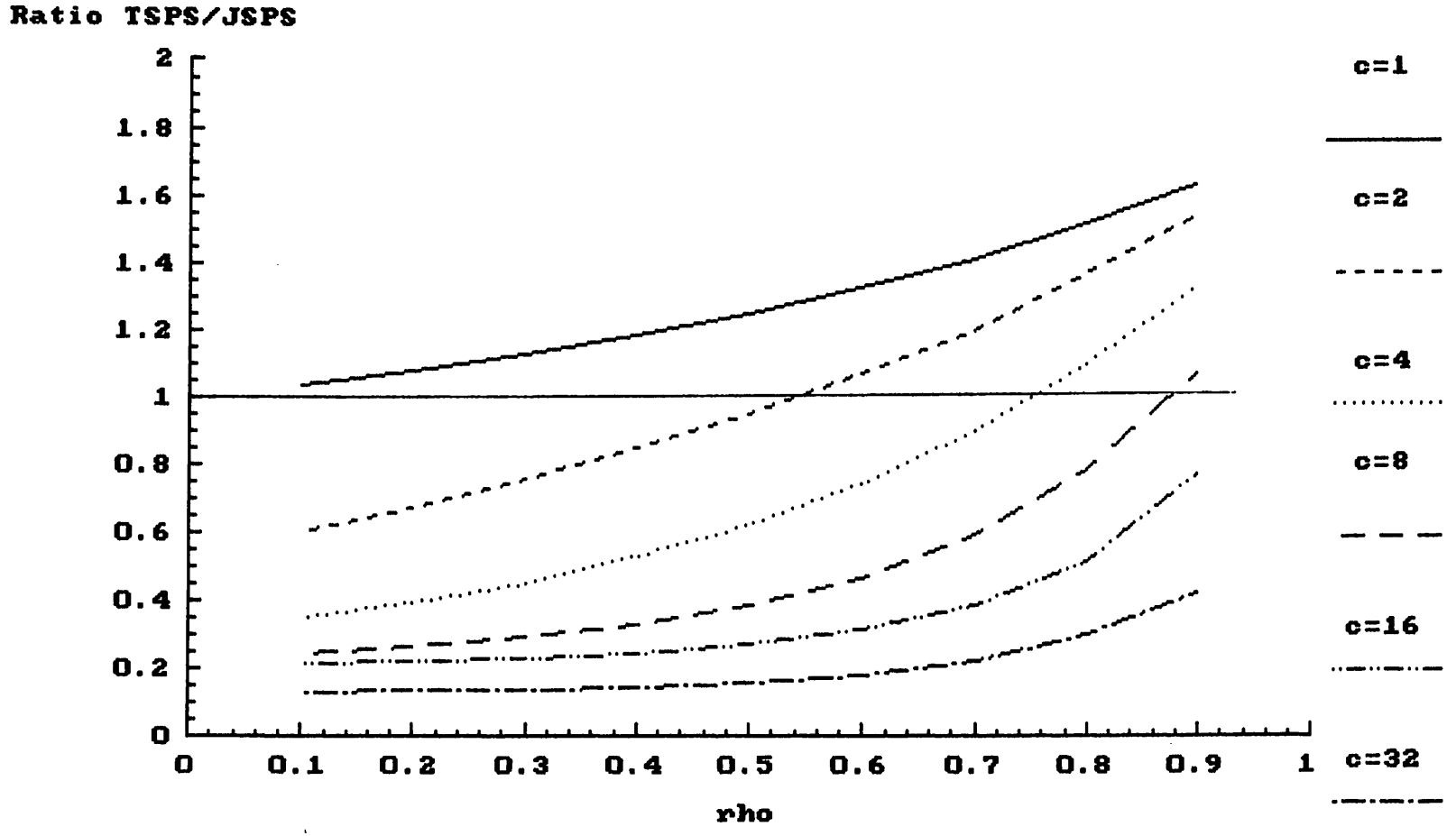
We now study the effect of dedicating a potion of the multiprocessor to each of the batch and edit classes. For edit jobs we assume that the computation time is small and equivalent to one task unit. **Batch jobs** are assumed to be large, consisting of fork-join tasks. The individual tasks from either class are assumed to have the same service requirements.

In order to examine the effect of statically dedicating a portion of the multiprocessor to each class, we compare the performance of a system composed of 16 servers where each server can run either class of job, to a partitioned system where some fraction of the processors are dedicated to each class. The combined system is composed of $c = 16$ servers. The partitioned system is composed of $c = 16$ servers such that K servers are dedicated to edit jobs and $c - K$ servers are dedicated to batch jobs. Our performance metric is the ratio of the response time of the partitioned system to that of the combined system.

TSPS/JSPS

VS

Number of Processors



x=16
B=350

FIGURE 7

gives the job with 9 tasks, 9/10 of the processor, and the job with a single task only 1/10 of the processor. However, JS-PS would give each job 1/2 of the processor. So on a single processor, TS-PS penalizes jobs with a small number of tasks. In a multiprocessor, there exists sufficient possibilities for parallelism so that this anomaly found in a single processor for TS-PS does not exist.

To study the effect of parallelism, we consider a workload of jobs consisting of 16 tasks and study the performance of TS-PS and JS-PS on systems with $c = 1, 2, 4, 8, 16, 32$ processors as a function of processor utilization. Figure 7 illustrates the results of this study plotting the response time ratios of TS-PS to JS-PS. We observe that TS-PS is always better than JS-PS in multiple processor systems when processor utilization is low. However, when the number of processors is small (< 8), there exists a utilization value, say ρ_0 such that system performance is better under JS-PS when $\rho > \rho_0$. This threshold is an increasing function of c the number of processors. This results because as the number of processors increases, the capability of sustaining parallel program execution under TS-PS increases.

4.3 Processor Partitioning

We now study the effect of dedicating a potion of the multiprocessor to each of the batch and edit classes. For edit jobs we assume that the computation time is small and equivalent to one task unit. **Batch jobs** are assumed to be large, consisting of fork-join tasks. The individual tasks from either class are assumed to have the same service requirements.

In order to examine the effect of statically dedicating a portion of the multiprocessor to each class, we compare the performance of a system composed of 16 servers where each server can run either class of job, to a partitioned system where some fraction of the processors are dedicated to each class. The combined system is composed of $c = 16$ servers. The partitioned system is composed of $c = 16$ servers such that K servers are dedicated to edit jobs and $c - K$ servers are dedicated to batch jobs. Our performance metric is the ratio of the response time of the partitioned system to that of the combined system.

In this experiment the independent parameter is the combined system utilization. Our first

In this experiment the independent parameter is the combined system utilization. Our first experiment consists of an arrival of 50 percent edit jobs and 50 percent batch jobs. Note that this arrival pattern results in the total computation time of edit jobs to be 1/16 of batch jobs. The partitioned system is defined by K and the equivalent flow of jobs. We plot our results in (Figure 8).

We can observe several interesting phenomena from Figure 8. First, by dedicating only one server to the edit jobs, $K = 1$, both edit and batch jobs degrade. Thus, a poor partitioning choice negatively effects both classes of jobs. Second, improvements can be made in the edit jobs by allocating enough additional servers, $K = 2, 3$, to handle the computational load of edit jobs, but this is done at the expense of the batch jobs. This phenomena is especially striking at high utilizations.

As the relative arrival rate between edit and batch jobs increases, as show in (Figure 9) where the proportion of edit jobs is 95 percent, we see that more servers must be dedicated to edit jobs before the mean response time is decreased. Note that in this case the total computation time required by edit jobs is greater than needed by batch jobs. The result is that 9 of the 16 processors are required to reduce the edit job response times (see (Figure 9)). There are regions in the figure in which the performance of both jobs classes decrease, however, we observe no region in which both classes improve performance. This phenomena has also been reported in [NTT87] for FCFS scheduling.

Figure 10 reports the results when batch jobs are composed of 4 tasks and the workload contains 50% batch and 50% edit jobs. The results in this figure are similar to the 95% edit jobs and 5% batch job tests shown in the previous figure. The reason for this is that when batch jobs are fairly small, $x = 4$, and there are 50% edit jobs and 50% batch jobs in the workload, then the total computational requirements of edit jobs is high (as in the 95% test) for a given utilization. Therefore, edit jobs will saturate a small number of processors. Notice that only when the number of processors dedicated to editing reaches 4, does editing perform well.

Partitioning
at
50 %

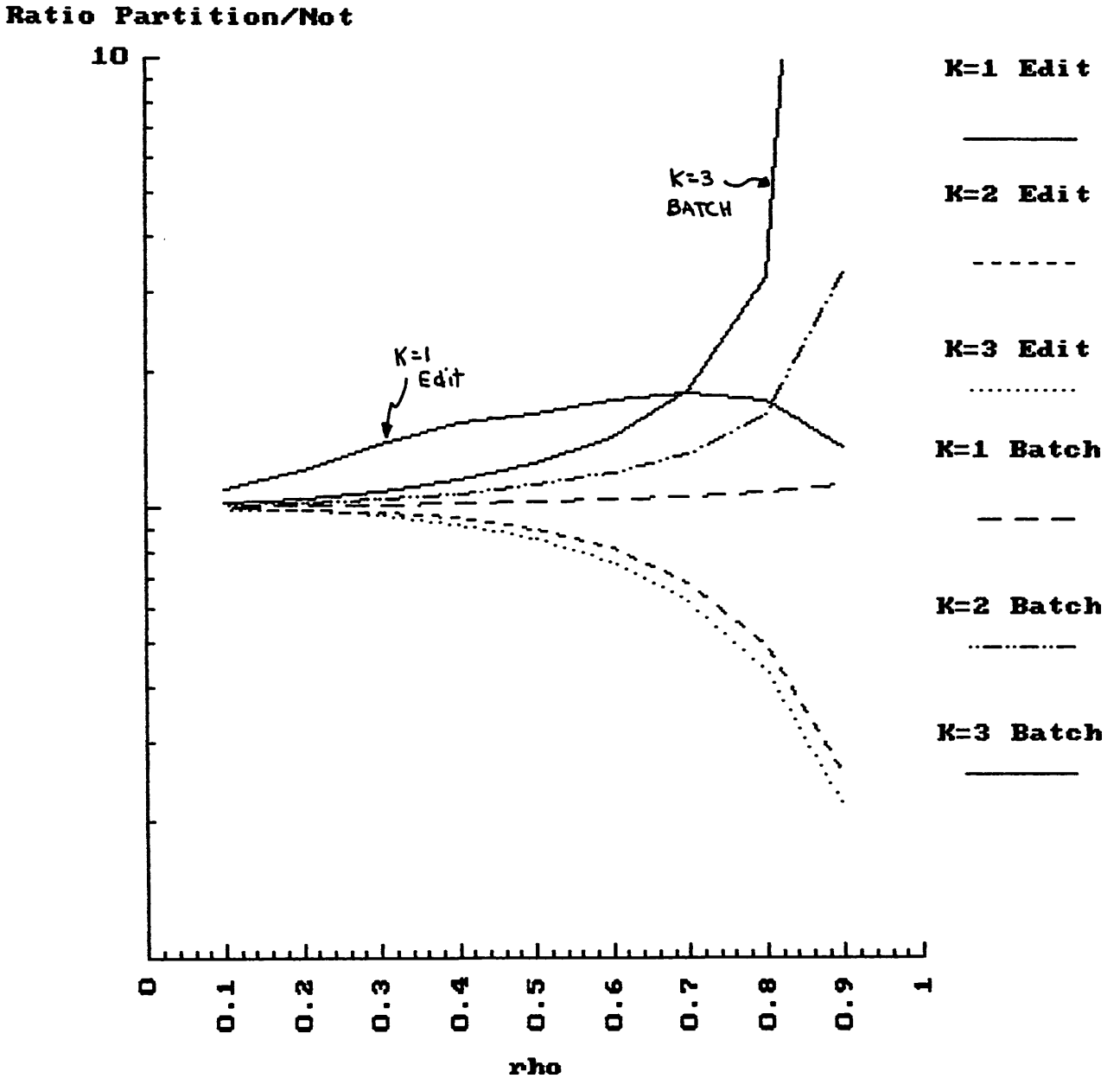


FIGURE 8

x=16
B=350

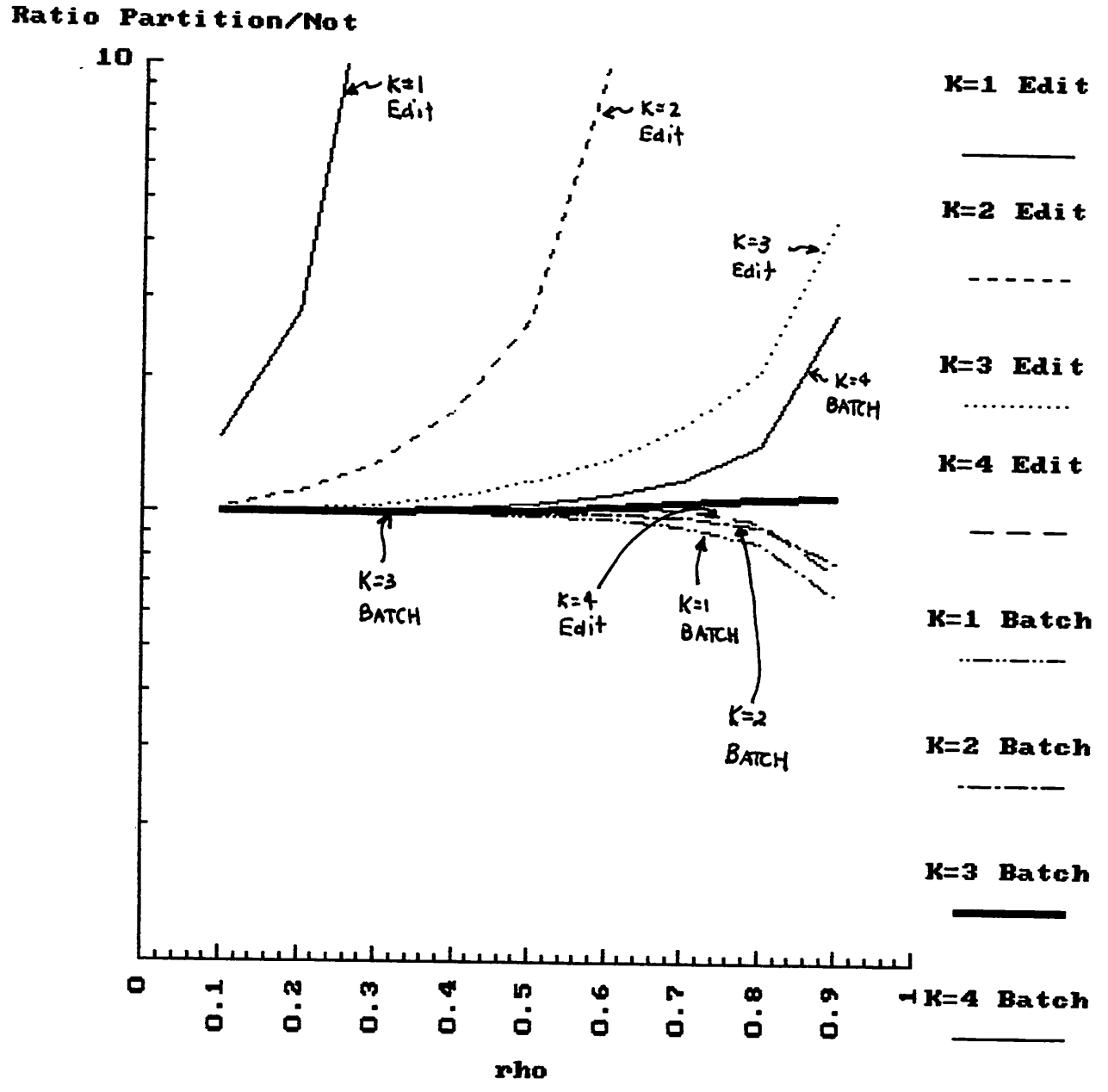
experiment consists of an arrival of 50 percent edit jobs and 50 percent batch jobs. Note that this arrival pattern results in the total computation time of edit jobs to be 1/16 of batch jobs. The partitioned system is defined by K and the equivalent flow of jobs. We plot our results in (Figure 8).

We can observe several interesting phenomena from Figure 8. First, by dedicating only one server to the edit jobs, $K = 1$, both edit and batch jobs degrade. Thus, a poor partitioning choice negatively effects both classes of jobs. Second, improvements can be made in the edit jobs by allocating enough additional servers, $K = 2, 3$, to handle the computational load of edit jobs, but this is done at the expense of the batch jobs. This phenomena is especially striking at high utilizations.

As the relative arrival rate between edit and batch jobs increases, as show in (Figure 9) where the proportion of edit jobs is 95 percent, we see that more servers must be dedicated to edit jobs before the mean response time is decreased. Note that in this case the total computation time required by edit jobs is greater than needed by batch jobs. The result is that 9 of the 16 processors are required to reduce the edit job response times (see (Figure 9)). There are regions in the figure in which the performance of both jobs classes decrease, however, we observe no region in which both classes improve performance. This phenomena has also been reported in [NTT87] for FCFS scheduling.

Figure 10 reports the results when batch jobs are composed of 4 tasks and the workload contains 50% batch and 50% edit jobs. The results in this figure are similar to the 95% edit jobs and 5% batch job tests shown in the previous figure. The reason for this is that when batch jobs are fairly small, $x = 4$, and there are 50% edit jobs and 50% batch jobs in the workload, then the total computational requirements of edit jobs is high (as in the 95% test) for a given utilization. Therefore, edit jobs will saturate a small number of processors. Notice that only when the number of processors dedicated to editing reaches 4, does editing perform well.

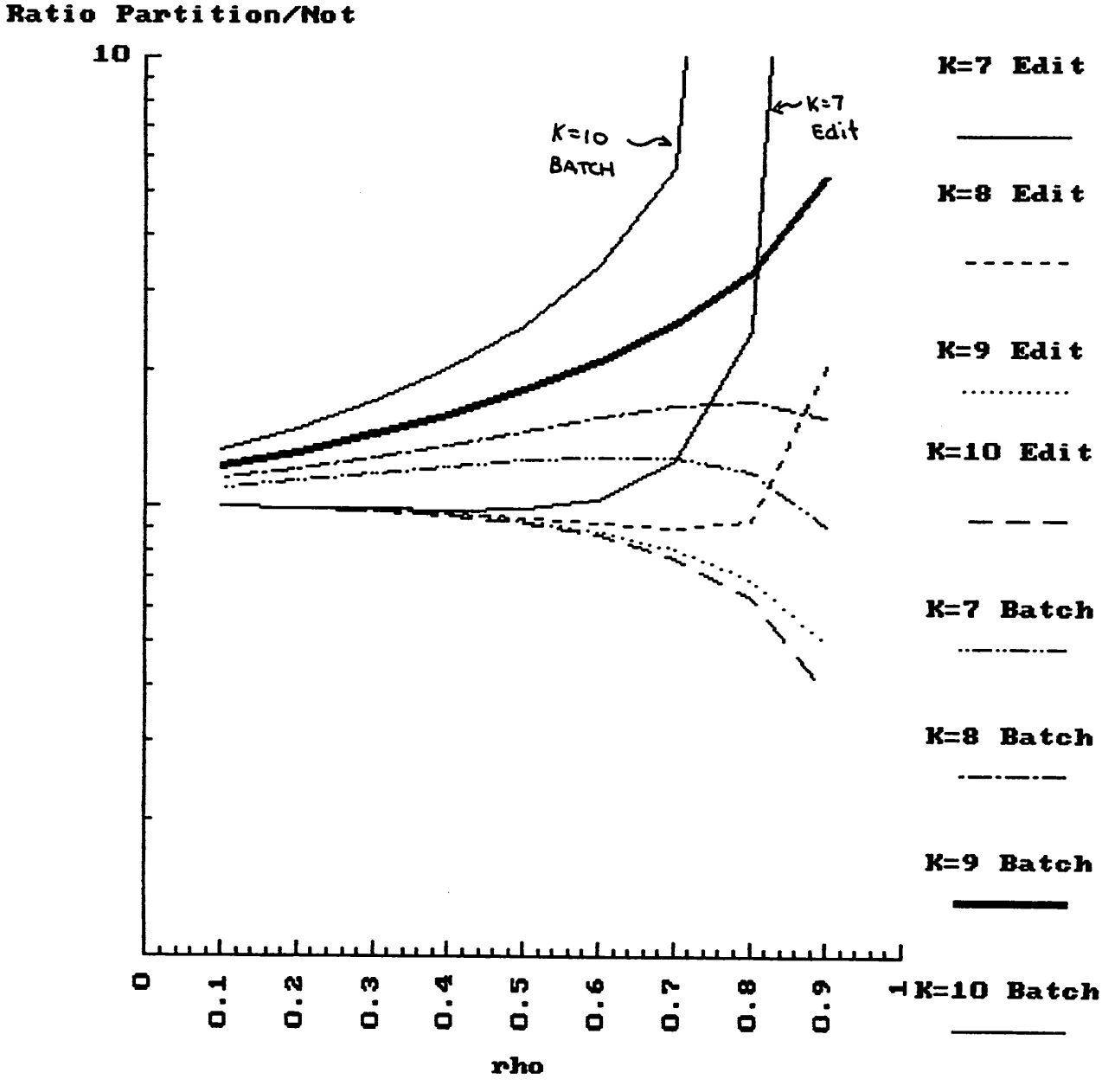
Partitioning
at
50 %



x=4
B=350

FIGURE 10

Partitioning
at
95 %



x=16
B=350

FIGURE 9

5 Summary

We have analyzed **fork-join** programs as a $M^X/M/c$ queueing system. We have obtained am expression for the mean response time of a fork-join task under processor-sharing. Since our expression is not in closed form, but given as a set of recurrent equations, we have obtained expressions for both lower and upper bounds. Our bounds become tight as the number of states increase.

We have compared three scheduling approaches: TS-PS, JS-PS and FCFS. We have observed that in general FCFS out performs both TS-PS and JS-PS. Likewise, we have observed that TS-PS performs better than JS-PS unless that number of servers is small compared to the number of tasks.

We have considered the interesting problem of partitioning the system into two subsystems. Each subsystem is dedicated to one of two job classes: edit jobs and batch jobs. We determined several interesting results. When half the jobs are edit jobs and one server is dedicated for edit jobs exclusively, both classes experience an increase in response time. Improvements in edit jobs always cause a reduction in the performance of batch jobs in the partitioned system. This suggests that a parallel system should have a controllable boundary for processor partitioning.

References

[All78] Arnold Allen. *Probability, Statistics and Queueing Theory.* Academic Press, New York, New York, 1978.

[BM85] F. Baccelli and A. Makowski. Simple computable bounds for the fork-join queue. *Proc. Conf. Inform. Sci. Systems*, 1985.

[BMT87] F. Baccelli, W. Massey, and D. Towsley. Acyclic fork-join queueing networks. *submitted to JACM*, 1987.

[CT83] Chaudhry and Templeton. *First Course in Bulk Queues.* J. Wiley, New York,

New York, 1983.

[GH76] Gross and Harris. *Introduction to Queueing Theory.* J. Wiley, New York, New York, 1776.

[Han75] P. Brinch Hansen. The programming language concurrent pascal. *IEEE Transaction on Software Engineering.*, 1, 1975.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, London, 1985.

[Kle64] Leonard Kleinrock. Analysis of a time-shared processor. *Naval Research Logistics Quarerly*, 11, 1964.

[Kle76] Leonard Kleinrock. *Queueing Systems Volume II: Computer Applications.* J. Wiley, New York, New York, 1976.

[NT85] R. Nelson and A.N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IBM Report RC11481*, 1985. to appear in *IEEE Transactions on Computers.*

[NTT87] R. Nelson, D. Towsley, and A. Tantawi. Performance analysis of parallel processing systems. to appear in *IEEE Trans. on Software Engineering.*

[Ost86] Anita Osterhaug. *Guide to Parallel Programming.* Sequent Computer Systems, Inc, Beaverton, Oregon, 1986.

[Pyl81] I.C. Pyle. *The Ada Programming Language.* Prentice-Hall International, London, 1981.

[RTS87] C. Gary Rommel, D. Towsley, and J. Stankovic. Ananlysis of fork-join jobs using processor-sharing. *Submitted to Operations Research*, 1987.

[Yao85] D. D. Yao. Some results for queues $M^X/M/c$ and $GI^X/G/c$. *Operations Research Letters*, 4, 1985.