The Performance of Multiprogrammed Multiprocessor Scheduling Policies

Scott Leutenegger
Mary Vernon

Technical Report #913

February 1990

TR 913

The Performance of Multiprogrammed Multiprocessor Scheduling Policies

Scott T. Leutenegger
Mary K. Vernon

Computer Sciences Department
University of Wisconsin - Madison
Madison, WI 53706

Abstract

Scheduling policies for general purpose multiprogrammed multiprocessors are not well understood. This paper examines various policies to determine which properties of a scheduling policy are the most significant determinants of performance. We compare a more comprehensive set of policies than previous work, including one important scheduling policy that has not previously been examined. We also compare the policies under workloads that we feel are more realistic than previous studies have used. Using these new workloads, we arrive at different conclusions than reported in earlier work. In particular, we find that the "smallest number of processes first" (SNPF) scheduling discipline performs poorly, even when the number of processes in a job is positively correlated with the total service demand of the job. We also find that policies that allocate an equal fraction of the processing power to each job in the system perform better, on the whole, than policies that allocate processing power unequally. Finally, we find that for lock access synchronization, dividing processing power equally among all jobs in the system is a more effective property of a scheduling policy than the property of minimizing synchronization spin-waiting, unless demand for synchronization is extremely high. (The latter property is implemented by *coscheduling* processes within a job, or by using a thread management package that avoids preemption of processes that hold spinlocks.) Our studies are done by simulating abstract models of the system and the workloads.

1. Introduction

Although multiprocessor architectures, parallel algorithms, and algorithms for scheduling a single parallel job on a multiprocessor, are active areas of current research, only a few studies have been done to determine how best to schedule concurrent jobs running on a multiprocessor [MaEB88, ZaLE89, TuGu89]. The purposes of this paper are to further our understanding of the issues involved, identify which characteristics of a scheduling policy are most significant determinants of the policy's performance, and to identify which policies are good choices for future study or implementation.

The environment we consider is a tightly-coupled shared-memory multiprocessor. We are interested in scheduling the machine for general use, allowing both serial and parallel jobs to run at the same time. We feel this is an important domain now and will remain important in the future.

Two types of scheduling policies were compared in a previous study [MaEB88]: 1) preemptive and non-preemptive versions of policies that are aimed at processing the "shortest" job first, and 2) a round-robin policy that allo-

This research was supported by the National Science Foundation under grant number DCR-8451405, and by grants from Cray Research, INC. and the AT&T Foundation.

cates an equal fraction of the processing power to each *process* in the system (RRprocess). We consider the preemptive versions of the "shortest" job first policies, as well as the RRprocess policy in this paper. Furthermore, we feel that two additional classes of policies are important to consider: 3) policies that explicitly *coschedule* processes that belong to the same job, and 4) policies that allocate an (approximately) equal fraction of the machine's processing power to each *job* in the system. For coscheduling policies, we consider two slightly modified versions of the best policy proposed by Osterhout [Oust82]. In the fourth category, we consider a round-robin policy that allocates equal quanta to each job in the system (RRjob), and a dynamic partitioning policy recently proposed by Tucker and Gupta [TuGu89]. The RRjob policy has not previously been studied.

We study the scheduling policies under a variety of workloads that we feel are more realistic than workloads used in previous studies. In particular, we vary the coefficient of variation of total job service demand, emphasizing the range of 3-5, but bound the number of processes per job by the number of processors in the system. The rationale for these assumptions is provided in section 3. We consider workloads with no correlation between total service demand and number of processes in a job, and workloads with a positive, linear correlation. We also consider the workloads with spinlock synchronization. We explore the effects of various scheduling policies on mean response time under these workload assumptions. When comparing the relative performance of scheduling policies, we find that, in most cases, the effect of lock synchronization spin-waiting is small compared to the effect of variance in job demand.

We study three policies from the shortest job first class: smallest number of processes first (SNPF), preemptive smallest number of processes first (PSNPF), and preemptive shortest cumulative demand first (PSCDF). Based on mean job response time, we find the following general performance ordering of the policies, from best to worst: dynamic partitioning and RRjob, RRprocess, coscheduling policies, SNPF, first-come first-serve (FCFS). Under extreme workloads some of the policies do not remain in this hierarchy, but on the whole this ranking is correct when the coefficient of variation of the total job service demand is higher than 1. In contrast to earlier work, we find that SNPF performs very poorly, even for workloads where the jobs demand is positively correlated with the number of processes. We also find that RRprocess performs poorly when there is a large variation in the number of processes, or a linear correlation between the number of processes and job demand. We reconcile these conclusions with previous work in section 4. PSCDF has excellent performance, but is not practical to implement. Thus, we are lead to conclude that of the policies we consider, scheduling policies that allocate processing power equally per job are the best candidates for future study and/or implementation.

Two types of overheads for preemptive scheduling policies are important to consider in a performance comparison. First, at the operating system level, preemption of a process running on a processor results in overhead for saving registers and starting a new process. Second, when a process is preempted, it must rebuild its cache entries when it starts up again. Even if rescheduled on the same processor, some or most of its entries may have been removed by the intervening processes. These scheduling overheads are not explicitly modeled in our study. Both factors are discussed qualitatively when we examine the performance results. The cache reload effects require more detailed analysis, as pointed out in the conclusions of our study.

The following terminology is used throughout the paper. A job is a serial or parallel program, composed of one (serial) or more (parallel) processes. We assume a job has finished execution once all of its processes have completed execution. We compare the policies according to their mean job response time for the different workloads studied. We define mean job response time to be the mean time from submission to completion of a job.

The rest of the paper is organized as follows. Section 2 describes the scheduling policies studied and discusses related work. Section 3 discusses our job model in more detail, the multiprocessor model, and the workloads used in our studies. Section 4 contains the results from our studies. Section 5 summarizes and proposes future work.

2. Scheduling Policies and Related Work

We describe the FCFS policy, the shortest job first policies, the coscheduling policies, RRprocess, and finally the policies that allocate equal processing power per job. We then offer some qualitative comments on the advantages and disadvantages of the policies.

2.1. First Come First Served Policy

- First Come First Served (**FCFS**) : When a job arrives, each of its processes is placed consecutively at the end of the shared process queue. When a processor becomes idle it simply removes the first process from the queue, and runs it to completion.

2.2. Shortest Job First Policies

- Smallest Number of Processes First (**SNPF**) and Preemptive Smallest Number of Processes First (**PSNPF**): For SNPF the shared process queue is organized as a priority queue, with highest priority given to processes from jobs with the smallest number of unscheduled processes. Jobs of equal priority, are ordered according to which job arrived first. As in FCFS, when a processor becomes idle it simply removes the first process from the queue, and runs it to completion. For PSNPF highest priority is given to jobs with the smallest number of incomplete processes. An arriving job with a smaller number of processes than an executing job will preempt processes belonging to the scheduled job.

- Preemptive Shortest Cumulative Demand First (**PSCDF**) : Like PSNPF, except processes from jobs with the smallest remaining cumulative service demand are given highest priority.

2.3. Coscheduling Policies

The goal of these scheduling policies is to achieve a high degree of coscheduling (i.e., a high degree of simultaneous execution of all processes from a single job). Ousterhout proposed and evaluated three policies: matrix, continuous and undivided [Oust82]. We have slightly modified the best policy (undivided), to eliminate some of the problems associated with filling in holes in the proposed array structure. We then modify this new policy to improve expected performance for correlated workloads.

- Cosheduled (**Cosched**) : There exists a linked list of processes. When a job arrives its processes are appended to the end of the list. When processes complete they are removed from the list. Scheduling is done by moving a window of length equal to the number of processors over the linked list. Each process in the window gets one quantum of service on a processor. At the end of the quantum, the window is moved down the linked list until the first slot of the window is over the first process of a job that was not completely coscheduled in the previous quantum. When a process within the window is not runnable, the window is extended by one process and the non-runnable process is not scheduled. All processors that switch processes at the end of a quantum do so at the same time.
- Coscheduled, version 2 (**Cosched2**) : Same as the Cosched policy except at the end of each quanta, the window is only moved to the first process of the next job, even if this job was coscheduled in the previous time slice.

2.4. Round Robin Policies

- Round Robin Process (**RRprocess**) : When a job arrives each of the processes are placed at the end of the shared process queue. A round robin scheduling policy is invoked on the process queue.
- Round Robin Job (**RRjob**) : Instead of a shared process queue there is a shared job queue. Each entry in the job queue has a queue holding its own processes. Scheduling is done round robin on the jobs. Each time a job comes to the front of a queue it gets P quanta of size q, where P = the number of processors in the system. If a job has fewer than P processes, each process gets a quantum of size $\frac{P}{N} \times q$, where N is the number of processes in the job's process queue. If a job has greater than P processes there are two choices. The first is to run P processes for one quantum each. Processes are chosen round robin from the job's process queue. The second choice is to give a quantum of size $\frac{P}{N} \times q$ to each process. This second choice has higher scheduling overhead. All of our studies assume such a job runs P

processes for one quanta each.

2.5. Dynamic Partitioning Policy

- Dynamic Partitioning (**dynamic-partitioning**) : This policy was proposed and implemented by Tucker and Gupta [TuGu89]. Their goal was to minimize context switching so that less time is spent rebuilding processor caches. Each job is dynamically allocated an equal fraction of the processors, except that no job is allocated more processors than it has runnable processes. Thus, if a machine has 20 processors and three jobs with 4, 10 and 20 runnable processes each, the first job would be allocated 4 processors and the other two would be allocated 8 processors each. The dynamic acquiring and releasing of processors requires coordination between the system scheduler and the application processes, as described in [TuGu89]. In their paper, Tucker and Gupta state that if the processes frequently reach states where they can safely suspend, then the actual number of processors a job is using will be very close to the allocated number. In our simulations, we assume the ideal case where the number of running processes in a job changes instantly whenever the allocations change. We also assume that when there are more jobs in the system than processors the extra jobs are held back in a "load queue". When a job leaves the system another job from the load queue is allowed into the system. We made this assumption in keeping with the idea that each job in the system is allocated at least one processor. If, instead, we assume all jobs in the system get one runnable process, the policy would be forced to time slice between the jobs. In this case many of the cache benefits of dynamic-partitioning would be lost.

2.6. Summary of Policies

FCFS is a simple policy and is included in the study for comparison purposes. Based on uniprocessor scheduling results, we expect that FCFS will perform poorly for coefficient of variation of job service demand greater than 1.

SNPF and PSNPF give preference to jobs with a small number of processes. This policy should approximate shortest job first for workloads with a positive correlation between the job service demand and the number of processes in the job. This policy was concluded to have good performance in [MaEB88]. However, again based on uniprocessor scheduling results, we expect SNPF to perform less well than other disciplines when the coefficient of variation in job service demand is above 1 and the number of processes is not positively correlated with the demand.

PSCDF should perform well under high demand variance, since scheduling is based explicitly on the demand parameter.

The coscheduling, RRprocess, RRjob, and dynamic partitioning policies all give a share of the processing power to each job. Thus, we expect these policies to perform reasonably well as the variance of job demand increases. However, RRprocess has an unfairness peculiar to the parallel processing environment, as pointed out in [MaEB88]. That is, jobs do not get an equal share of the processing power. Instead, jobs that have a large number of processes receive more processor quanta than jobs with a small number of processes. The coscheduling policies suffer the same problem. The problem is accentuated by the correlated workloads. Thus, we expect that the RRjob policy (not previously studied) and the dynamic-partition policy have the most promise for dealing with job demand variance.

The advantage of Cosched2 over Cosched is that processes from jobs with a small number of processes will receive more quanta than processes from jobs with a large number of processes. As a result, we expect Cosched2 to perform better than Cosched for linearly correlated workloads.

Jobs that access shared locks are affected by scheduling policies in two ways. First, a policy that results in a high degree of coscheduling can result in a high degree of competition for the lock since most processes from the same job will be scheduled at the same time. Second, a policy that can deschedule processes that hold a lock can cause long idle times if processes spin while waiting for locks [ZaLE88]. RRprocess and RRjob are prone to this problem. Non-preemptive policies such as FCFS and SNPF are not subject to this problem. While Cosched and Coshed2 may deschedule processes that are holding a lock, they also will most likely deschedule the other processes from that job at the same time, thus this problem will likely not hinder the co-scheduling policies as much. Dynamic-partitioning assumes a thread management package that ensures that processes holding a lock are not descheduled. Zahjoran, Lazowska and Eager [ZaLE89] have shown that such thread management significantly reduces processor spin-waiting for scheduling disciplines that statically partition the available processors. RRjob could also be implemented along with such a thread package, in which case, like dynamic-partitioning, it would not deschedule processes holding a lock. We reconsider this issue in sections 4.3 and 4.4.

We have not included static partitioning policies in our study. These policies have lower overhead than dynamic partitioning policies, but divide the processing power unequally among jobs, and have poorer load balancing properties. Thus, we feel that the dynamic partitioning policy has more merit.

3. Model Description

In this section we describe our models of the job structures, workloads and the multiprocessor.

3.1. Multiprocessor model

We assume an "ideal" multiprocessor. That is, we assume no overhead for the scheduling policy and no overhead for context switches. We further assume that a single shared run queue is not a bottleneck in the system [LiCh89]. We assume processes arrive and queue for service in this single work queue shared by all the processors. How the work queue is organized depends on the scheduling policy. For all the studies presented in this paper we set the number of processors equal to 20. This model is the same as in [MaEB 88].

3.2. Job Model

We have studied the scheduling policies based on two job models. The first model assumes that the processes are independent. When a job arrives to the system, it splits into n processes. Each process is independent of the others. The job is not finished until all of its processes have completed. Similar structures have been studied in [NeTT 87][MaEB 88].

The second job model assumes processes access shared locks. Our model is essentially the same as in [ZaLE89]. Once again the job splits into n processes and the job is not completed until all processes have finished. We assume there is one shared lock for each job, and all processes within a job contend for the lock. We assume a process requesting a lock that is currently held by another process spins (busy waits) until the lock becomes available. When a lock is released, the next process to acquire the lock is chosen randomly from all running processes waiting for the lock. We assume that the lock holding time is deterministic and is set equal to $0.01 \times q$ where q is the size of the quantum used in the RRprocess scheduling policy. The time between requests to the lock is exponentially distributed. The mean inter-request time is set according to the number of processes in the job, so that the lock demand per job remains fixed. Each time a process releases the lock it gets a new interrequest time from this distribution. The process does not request the lock again if the time to the next request plus the lock holding time is less than the remaining service time of the process.

3.3. Workload Model

The workload model defines the ways in which the number of processes per job and total service demand per job are calculated. We assume that the total service demand is divided equally among the processes. In [Maju88] both equal division of the demand, and a uniform distribution of demand amongst the processes were considered. Majumdar found no qualitative and very little quantitative difference between the results obtained from both assumptions. We have confirmed these claims for both job models. We thus assume equal distribution of demand in all our experiments.

We have used the two workload models defined in [MaEB88], and added two new workloads. The new workloads use the same models of job demand as the earlier workloads, but have a different model of the number of processes per job. For reasons that will be explained below, we call the previous models the "high_C_n" workloads, and our new models the "low_C_n" workloads. We first describe the high_C_n and low_C_n models of the number of processes per job, and then we review the "uncorrelated" and "correlated" models of job demand.

- **high_C_n workloads:** The input parameters are mean number of processes ($\bar{n}$) and coefficient of variation of number of processes (C_n). The number of processes is determined from a two stage hyperexponential distribution. For example, a 95% probability of choosing the small mean, with small mean equal to 0.82 processes and large mean equal to 60 processes, is used to implement a distribution of number of processes with mean 4.0 and coefficient of variation of 5.0.

The above model can be used to generate workloads with values for C_n greater than 1.0, but may not be realistic. That is, in the example where C_n is 5.0, 67% of the jobs are sequential, whereas 5% of the jobs have an exponentially distributed number of processes with a *mean* three times the number of processors in the system. We find in section 4 that these assumptions significantly affect the relative performance of the scheduling policies. We hypothesize that a real workload might have more jobs with small amounts of parallelism, as well as a reasonable fraction of jobs with parallelism equal to the number of processors. We thus define the following new model of the number of processes per job.

- **low_C_n workloads:** The maximum number of processes per job is equal to the number of processors. Input parameters are the probability that a job has a number of processes equal to the number of processors (P_p), and the mean number of processes for all other jobs ($\bar{n}$). With probability $1 - P_p$ the job has a number of processes chosen from an exponential distribution with mean $\bar{n}$. The number of processes is set equal to the number of processors if the random number chosen from the exponential distribution exceeds the number of processors. (For an exponential distribution with a mean of 4.0 only 0.7% of the samples exceed 20.) We designate this workload as "low_C_n" since the coefficient of variation of the number of processes will always be less than 1.0.

We combine each of the above models with the following two methods (defined in [MaEB88]) for computing the total service demand for the job, yielding a total of four distinct workloads.

- **uncorrelated job demands:** There is no correlation between the number of processes and the total demand of the job. The input parameters are mean job demand ($\bar{d}$) and the coefficient of variation for the demand (C_d). The job demand is determined from a two stage hyperexponential distribution.

- **correlated job demands:** A job's mean service demand is linearly dependent on its number of processes. Thus jobs with a large number of processes are likely to have a large demand, whereas jobs with a small number of processes are likely to have a small demand. (Such a linear correlation is partially justified by the arguments presented in [Gust88].) The input parameters are a demand variation parameter (C_v), and a scalar (t). Let the number of processes computed for a job be represented by n. The demand for the job is obtained from a hyperexponential distribution with the mean set equal to $n \times t$ and coefficient of variation equal to C_v. Note that the input C_v is not equal to the coefficient of variation of the demand due to the linear dependence.

In all the workloads, the computed number of processes is generally a non-integer value. As in [MaEB88], if the computed value is not an integer, the next largest integer is used. As a result, the actual measures of $\bar{n}$ and C_n are not exactly equal to the input parameters.

4. Results

In this section we present the results of our studies. Section 4.1 presents the results for the uncorrelated_high_C_n and uncorrelated_low_C_n workloads, assuming independent processes (i.e., no lock synchronization). Section 4.2 presents the results for the workloads where number of processes per job is positively correlated with total demand for the job (i.e., correlated_high_C_n and correlated_low_C_n), again assuming independent processes. Section 4.3 presents the results assuming processes access shared locks. In section 4.4 we further compare the two most promising policies from Sections 4.1-4.3: RRjob and dynamic-partitioning. Due to the converting non-integer numbers of processes to integers, and also due to the linear correlation in the correlated workloads, input parameters do not equal the measured output parameters. Tables 4.1 and 4.2 contain the input and the measured output parameters for each of the figures. A "-" in an entry in the table signifies that the parameter was varied. For the round robin policies the quantum size, q, is set equal to 0.1 units. Hence, the mean job demand for most experiments is 200 quanta. We quote only the most relevant input parameters with the figures. All models have been simulated using the DeNet [Livn88] simulation language. All results have confidence intervals of 10% or less (usually less than 5%) at a 90 percent confidence level.

Table 4.1: Input Parameter Values

Figure	C_n	$\bar{n}$	P_p	C_d	$\bar{d}$	C_v	t
4.1a	5.0	4.0		5.0	20.0		
4.1b		4.0	0.4	5.0	20.0		
4.1c		4.0	0.4	-	20.0		
4.2a	5.0	4.0				2.0	5.0
4.2b		4.0	0.05			2.0	5.0
4.2c		4.0	0.4			5.0	2.0
4.2d		4.0	-			5.0	2.0
4.2e		4.0	0.4			-	2.0
4.3a		4.0	0.4	5.0	20.0		
4.3b		4.0	0.4	-	20.0		
4.4		4.0	0.4			3.0	2.0

Table 4.2: Output Parameters

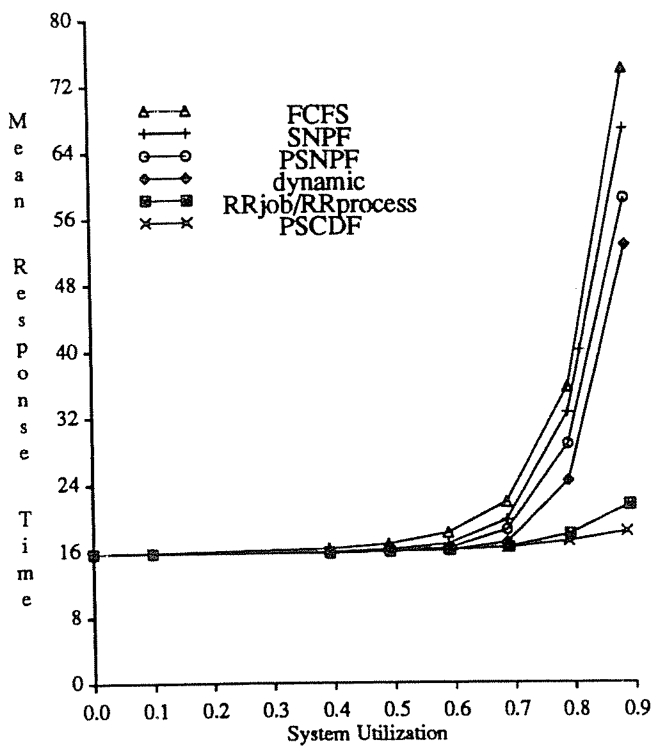
Figure	C_n	$\bar{n}$	C_d	$\bar{d}$
4.1a	4.36	4.56		
4.1b	0.76	10.72		
4.1c	0.76	10.68		
4.2a	4.29	4.59	10.50	23.50
4.2b	0.95	5.31	2.92	26.52
4.2c	0.76	10.72	6.34	21.60
4.2e	0.76	10.70	-	21.58
4.3a	0.76	10.69		
4.3b	0.76	10.72		
4.4	0.75	10.75	3.80	21.42

4.1. Uncorrelated Workloads, Independent Processes

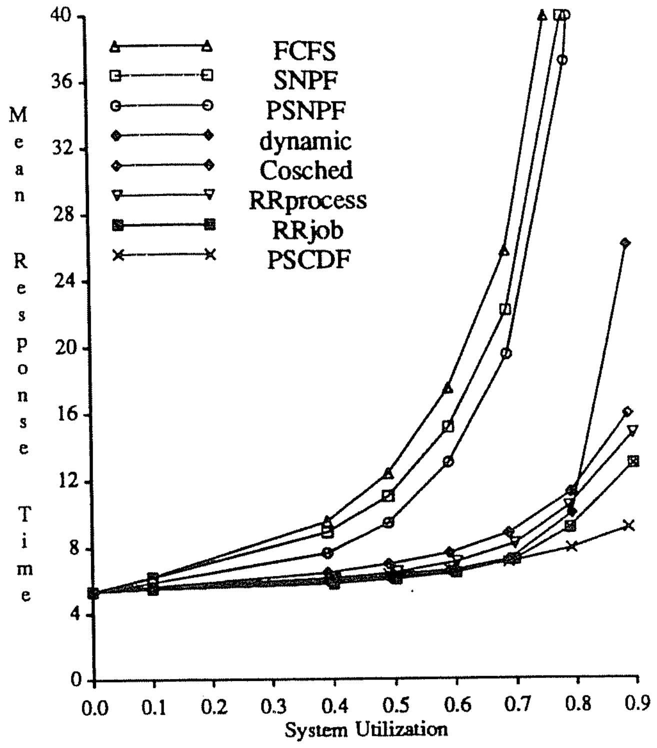
In this section we compare the scheduling policies assuming the jobs do not require any inter-process synchronization and that the job service demand is not correlated with the number of processes. Figure 4.1a plots response time versus utilization for the uncorrelated_high_C_n workload, assuming C_d=5.0, and C_n=5.0. The curves are listed in the legend in order of decreasing mean response time. The response time curves are nearly identical for RRjob, RRprocess, Cosched, and Cosched2, and are represented by the RRjob/RRprocess curve. Note that RRjob and RRprocess would be expected to perform approximately the same for this workload, since a large fraction of the jobs are sequential.

The input parameters and assumptions of figure 4.1a are similar to figure 3.a in Majumdar et. al. [MaEB88]. The discrepancy between the two figures is due to a subtle programming error in Majumdar's workload A generator, which resulted in a positive correlation between number of processes and total job demand [MaEa89]. As a result, their experiment showed SNPF and PSNPF perform well and RRprocess performed poorly (see section 4.2). However, for the uncorrelated case, just the opposite is true, as can be seen in figure 4.1a. SNPF and PSNPF perform slightly better than FCFS, since more jobs run simultaneously, increasing the likelihood of running a job with a small demand. Dynamic-partitioning also performs poorly for this workload, due to the workload parameters and to our assumption that this discipline only allows as many jobs into the system as there are processors (i.e., 20 in our experiments). The excess jobs are held in a load queue, not receiving service (see section 2.5). Since a large percentage of the jobs are sequential, at moderate to high load the number of jobs in the system often exceeds the number of processors. As a result dynamic-partitioning behaves similar to FCFS. PSCDF performs better than the round robin policies at high loads.

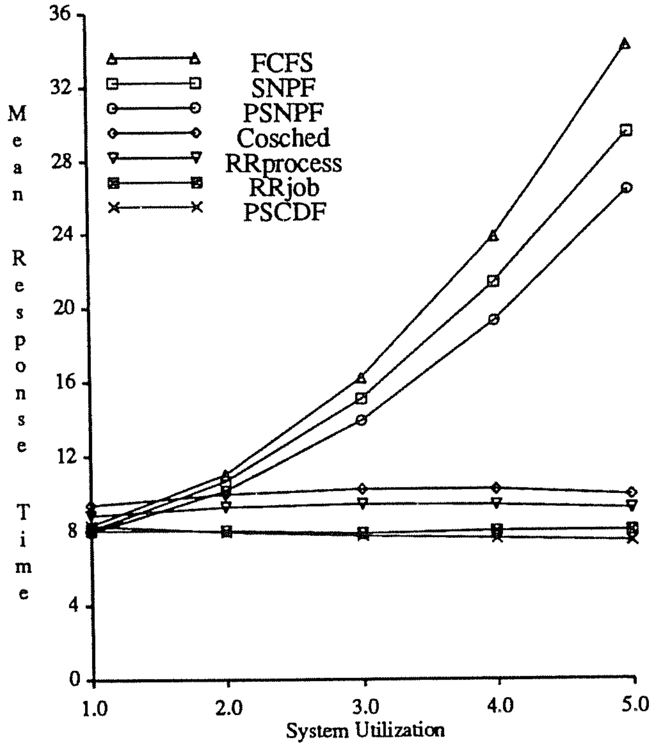
As discussed in section 3.3, we feel the uncorrelated_low_C_n workload is more realistic than the workload in figure 4.1a. Figure 4.1b plots mean response time versus utilization for the uncorrelated_low_C_n workload. (Note that the difference in scale between figures 4.1a and 4.1b.) Once again, SNPF and PSNPF are only slightly better than FCFS.



(a): Mean response time, high_C_n workload, $C_d = 5$



(b): Mean response time, low_C_n workload, $C_d = 5, P_p = 0.4$



(c): Effect of C_d, low_C_n workload, system utilization = 0.75

Figure 4.1: Performance when Job service demand and number of processes are not correlated

The performance of dynamic-partitioning has improved and now performs similar to RRjob at system utilizations up to 0.8. At 90% utilization, dynamic-partition performs worse than Cosched. Once again, the reason for this is due to jobs being withheld from the system. Only at very high loads does the number of jobs exceed the number of processors a large fraction of the time. We also find that, for this workload, RRjob performs better than RRprocess and Cosched. In particular, at 70% utilization, RRjob has a 11.9% lower mean response time than RRprocess; whereas Cosched has a 7.6% higher mean response time than RRprocess. Cosched2, not shown, has just slightly higher mean response time than Cosched. PSCDF remains the best.

Figure 4.1c shows the effect of C_d on the uncorrelated_low_C_n workload. C_d is varied from 1.0 to 5.0. Utilization is fixed at 75%. The response time curves for RRjob and dynamic-partition are nearly identical and are represented by the RRjob curve. Consistent with uniprocessor scheduling results, FCFS, SNPF, and PSNPF are very sensitive to C_d. At $C_d = 1.0$ FCFS, SNPF, and PSNPF perform somewhat better than RRprocess and almost as well as RRjob. However, as C_d increases the performance of FCFS, SNPF, and PSNPF degrade dramatically. All other polices studied are rather insensitive to C_d. It appears that SNPF and PSNPF are not good policies for uncorrelated workloads, assuming C_d will be greater than one for real parallel system workloads, as is the case in typical uniprocessor workloads [SaCh81].

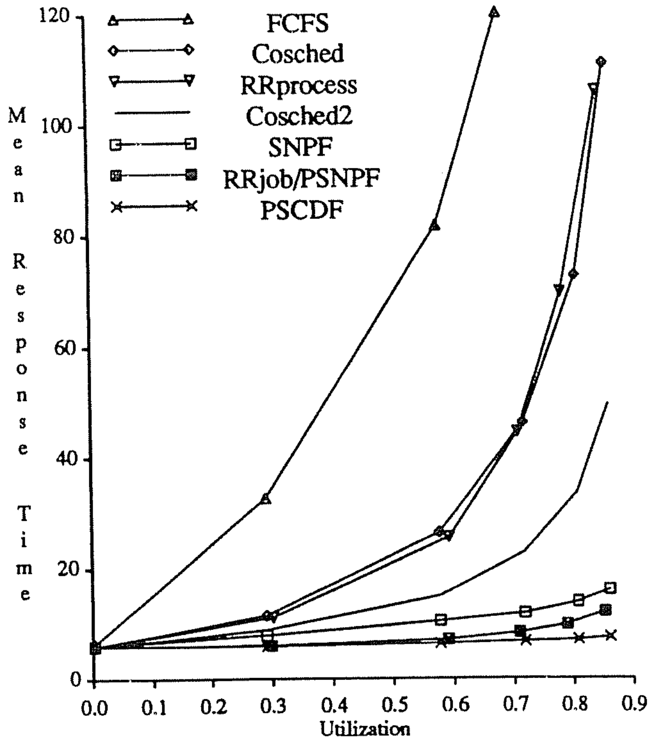
Our conclusions from the independent process workload are:

- SNPF and PSNPF perform poorly, except when coefficient of variation of job service demand is below 1.
- RRprocess performs better than the coscheduling policies.
- RRjob performs better than all non-equal processing power allocation policies, as does dynamic-partition for the uncorrelated_low_C_n workload at up to 80% system utilization.

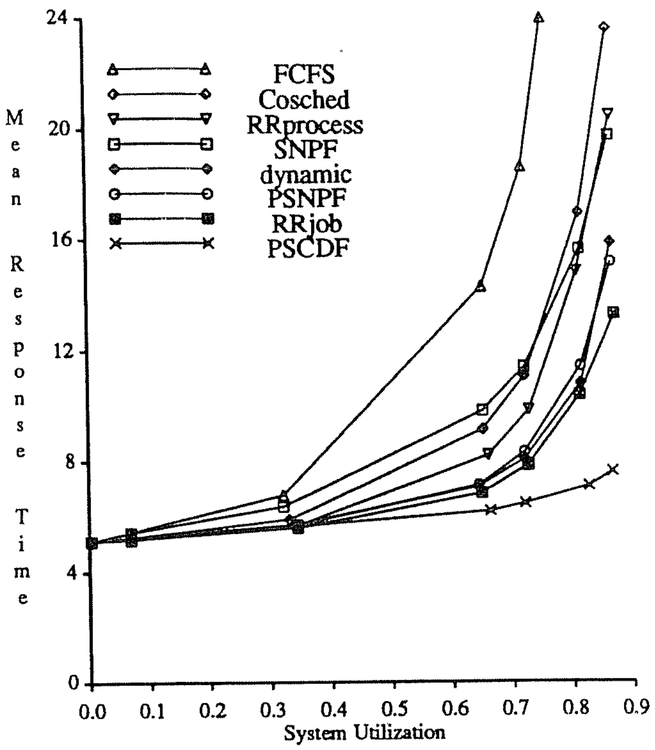
4.2. Correlated Workloads, Independent Processes

In this section we compare the scheduling policies assuming the jobs do not require any inter-process synchronization and that the job service demand is correlated with the number of processes.

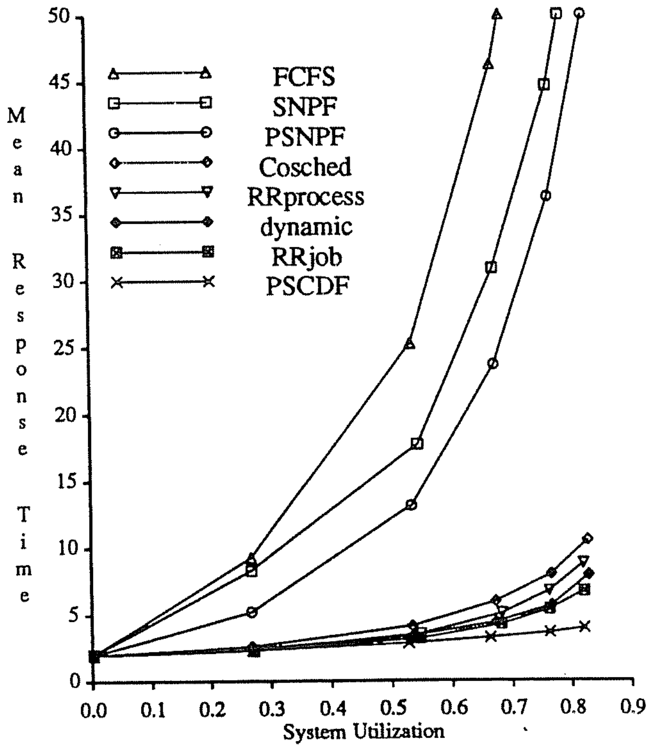
Figure 4.2a plots mean response time versus utilization for Majumdar et. al.'s correlated_high_C_n workload [MaEB88]. The parameters of figure 4.2a are similar to figure 3.b in [MaEB88], and the results in the two figures agree. The curves are listed in the legend in order of decreasing mean response time. The first thing to note is that RRprocess and Cosched do not perform well. This is because jobs with a large number of processes get a larger fraction of processing power than jobs with a small number of processes. These jobs are also likely to have large service demands since demand is correlated with the number of processes. As a result these jobs tie up the system for long periods of time while the other jobs in the system get little access to the processors. Cosched2 perform substantially better than RRprocess and



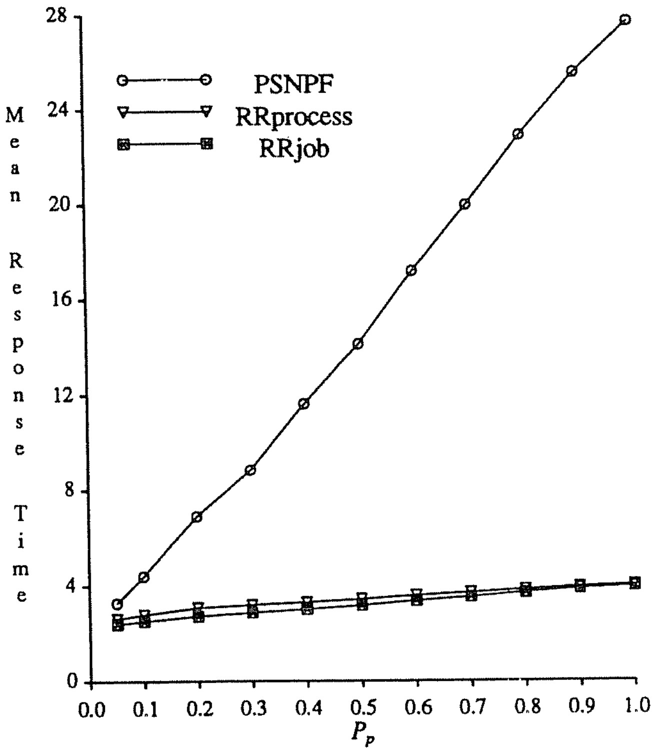
(a): Mean response time, high_C_n workload, C_n = 5



(b): Mean response time, low_C_n workload, P_p = 0.05



(c): Mean response time, low_C_n workload, P_p = 0.4



(d): Effect of P_p, system utilization = 83%

Figure 4.2: Performance when job service demand is linearly dependent on the number of processes

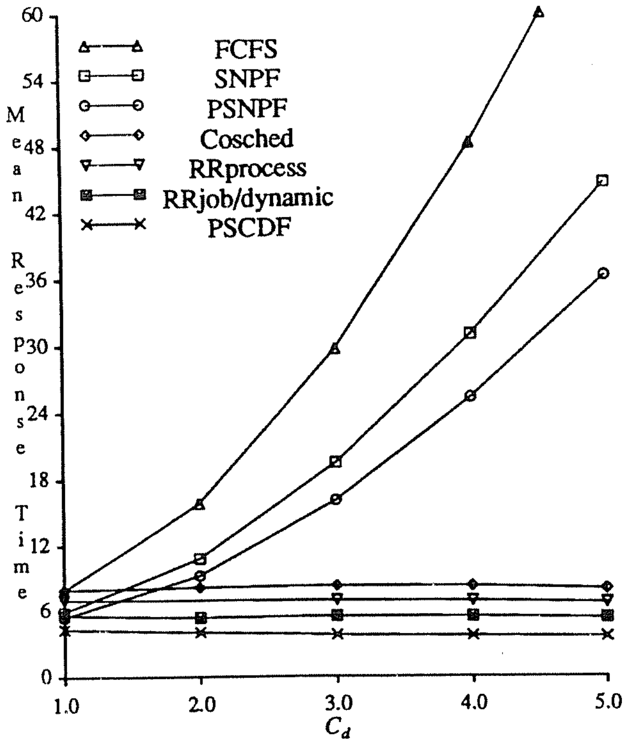


Figure 4.2e: Effect of C_d

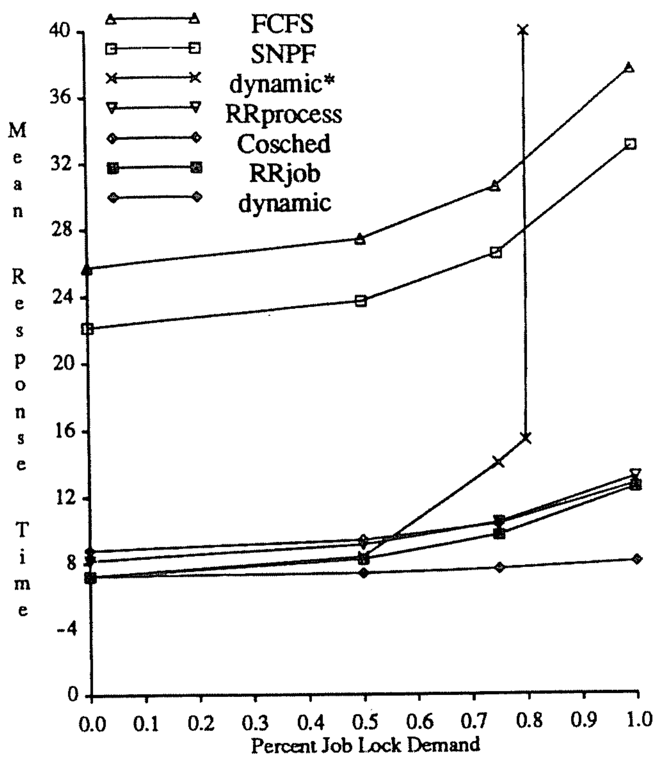


Figure 4.3a: Mean response time, with lock accessing, offered system load = 70%

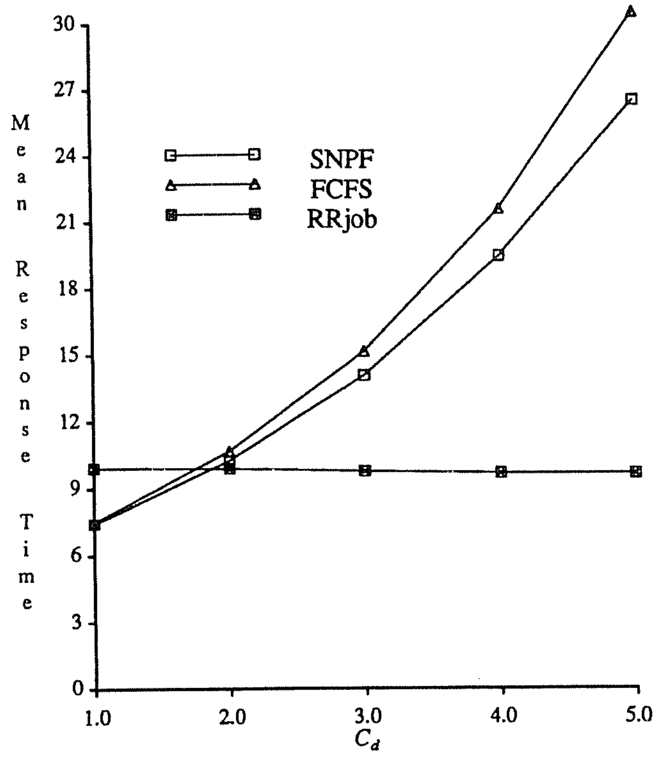


Figure 4.3b: Effect of C_d with lock accessing, offered system load = 70%, lock demand = 75%

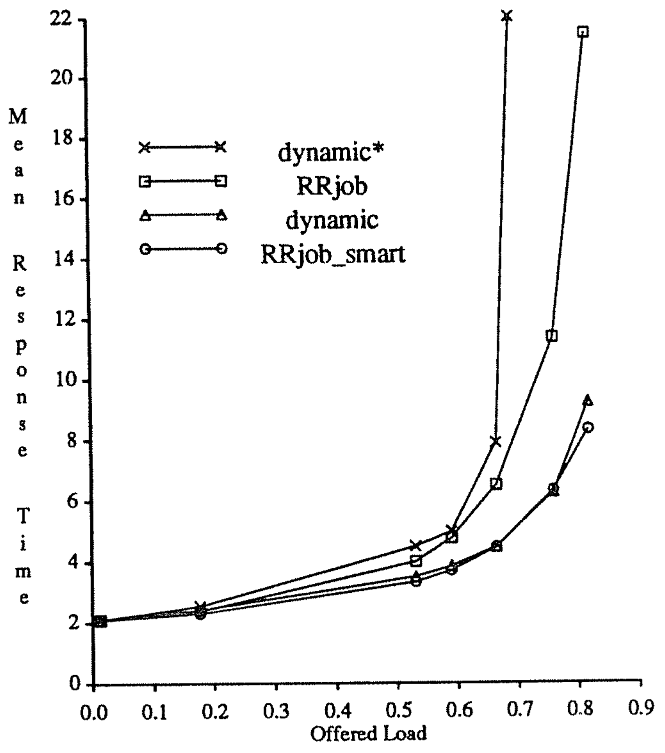


Figure 4.4: RRjob compared to Dynamic-partitioning

Cosched. This is because processes from jobs with a small number of processes receive multiple quanta in Cosched2, thus giving a more equal fraction of the processing power to jobs that are likely to have a small demand.

The curves for dynamic-partitioning, RRjob, and PSNPF are almost identical and are represented by the line RRjob/PSNPF. SNPF and PSNPF perform well. As a result of the correlation, SNPF and PSNPF give preference to what are likely to be the shortest jobs. RRjob performs very well, as does dynamic-partition. This is because, in both policies, jobs with a small number of processes get a fair share of the processing power. Dynamic-partitioning is very close to RRjob even at high utilizations.

The effect of reducing the variance of parallelism for the "highly parallel" jobs in the correlated workload can be seen by comparing figures 4.2a and 4.2b. In figure 4.2a the parameters of the hyperexponential distribution resulted in 95% of the jobs having the number of processes drawn from an exponential distribution with mean 0.82 and 5% of the jobs having the number of processes drawn from an exponential distribution with mean 60. In figure 4.2b 95% of the jobs have the number of processes drawn from an exponential distribution with mean 4.0 and 5% have 20 processes. As a result, the workload of figure 4.2a has a much greater variation in job demand than the workload for figure 4.2b. Note the difference in y-axis scale between figures 4.2a and 4.2b. The curves are listed in the legend in order of decreasing mean response time. Relative to the other policies, SNPF and PSNPF perform considerably worse in figure 4.2b than in 4.2a, due to the lower variation in cumulative job service demand. Conversely, RRprocess and Cosched perform much better for the correlated_low_C_n workload than for correlated_high_C_n workload. This is because there are no jobs with extremely large numbers of processes, and hence also very large demands, receiving a disproportionate share of the system. Dynamic-partitioning and RRjob perform almost identical until high utilization where dynamic-partition performs worse. For this workload RRprocess performs as well as SNPF and RRjob performs better than PSNPF.

Figure 4.2c shows the correlated_low_C_n workload when $P_p = 0.4$ and $t = 2$. The parameter t was decreased to 2 to keep the demand per job near 20, since the mean number of processes increases as a result of increasing P_p. Here we find that SNPF and PSNPF do not perform well at all. The larger value of P_p implies there are fewer jobs that can benefit from PSNPF scheduling.

The effect of P_p on PSNPF, RRprocess and RRjob can be seen explicitly in figure 4.2d. Here all parameters are held constant except P_p which is varied from 0.05 to 1.0. Arrival rate is adjusted to keep the system utilization constant at 50%. There is no entry for figure 4.2d in table 4.2 because varying P_p effects all output parameters. The round-robin policies experience a slight increase in response time, since mean job demand increases as P_p increases. PSNPF degrades because of the increased job demand and more importantly because with fewer small jobs PSNPF has less leverage to approximate shortest job first. At $P_p = 1.0$, PSNPF is identical to FCFS. RRprocess and RRjob are identical at $P_p = 1.0$

since in RRjob each process gets 1 quantum as in RRprocess.

We experimented with varying t while adjusting the arrival rate to keep the system utilization constant. We found that for all policies, mean response times increases by about the same percent as t increases.

Figure 4.2e shows the effect of varying C_d for the correlated_low_C_n workload. C_d is varied from 1.0 to 5.0, with utilization fixed at 75.6%. At $C_d = 1.0$ SNPF and PSNPF perform better than RRprocess, and the same as RRjob. However as C_d increases SNPF and PSNPF quickly degrade while mean response times for the round robin policies remain almost constant. The curves for RRjob and dynamic-partition are almost identical and are drawn as one line.

Our conclusions from the linearly correlated workloads are:

- Policies that allocate processing power equally among the jobs perform better than RRprocess, coscheduling, and, perhaps surprisingly, SNPF or PSNPF.
- RRprocess also performs better than SNPF and PSNPF when C_n is low and C_d is 2.0 or greater.

4.3. Lock Accessing Synchronization

For the lock accessing job structure we made the assumption that processes spin when they request a lock that is held by another process. There are two types of spinning overhead that occur. The first is spinning while the process holding the lock executes. This type of spinning occurs for all policies. However, a higher degree of co-scheduling of processes from the same job leads to higher competition and thus more spinning. The second type of spinning overhead occurs when a process requests a lock held by a process that is descheduled. FCFS and SNPF are immune from this problem, since these policies are non-preemptive. Dynamic-partition uses thread management (or "application scheduling") [Doep 87] [BeLL 88], to eliminate the second type of spinning. The round robin and coscheduling policies might also use thread management to avoid the second type of spinning, but are assumed not to in the experiments we perform in this section.

Figure 4.3a plots response time versus job lock demand for the uncorrelated_low_C_n workload. We define job lock demand as (process lock demand) times (number of processes), where process lock demand is the percent of time each process would use the lock if there were no competition. For example, if job lock demand is 100% and the number of processes is 10 then the process lock demand is 10%, or the inter-request time is $0.09 \times q$, where q is the size of the quantum used in the RRprocess policy. For a job with 20 processes running by itself on the machine, a job lock demand of 100% results in the lock being utilized 84.6 percent of the time. The offered load to the system is fixed at 70% utilization. We define offered load as the arrival rate that generates this utilization for the independent process workload. The real utilization of the system is higher since there is extra work introduced by the spinning.

The mean response time of FCFS, SNPF, and Cosched increases by 46.5%, 49.1%, and 45.1% respectively when job lock demand is varied from 0 to 100%. For FCFS and SNPF all of this increase is due to competition, since these policies never deschedule processes. RRprocess and RRjob increase by 61.4% and 74.6% respectively. The percent increase in response time seen by RRjob and RRprocess, in comparison to the smaller increase in FCFS and SNPF, is insignificant compared to the difference in response time caused by C_d.

Dynamic-partition assumes a thread package that ensures processes holding locks are not descheduled. As a result it performs much better than RRprocess and RRjob. If a thread package was assumed for RRprocess and RRjob they would perform similar to dynamic-partition. We include a new variation of dynamic-partition in this experiment called dynamic-partition*. Dynamic-partition* is the same as dynamic-partition, except that it does not assume an intelligent thread package. As a result, processes holding locks can be descheduled. At a lock demand of 85% the mean response time for dynamic-partition* is 219.3. This poor performance occurs because once a process holding a lock is descheduled it remains descheduled for very long periods of time resulting in large amounts of spinning. This implies that an intelligent thread package is a requirement if dynamic-partitioning is to be implemented.

As job lock demand is increased, Cosched has a lower mean response time than RRprocess, whereas even at 100% job lock demand RRjob is better than Cosched. Cosched2 performs slightly worse than Cosched, and is not shown.

Figure 4.3b plots mean response time versus C_d for the uncorrelated_low_C_n workload with job a lock demand of 75% and an offered load of 70.0%. Although SNPF is better than RRjob for $C_d = 1.0$ it degrades as C_d increases. RRjob improves as C_d increases. The curves for RRprocess and Cosched are similar to RRjob, but have slightly higher mean response times. Once again we see that SNPF is not an acceptable policy for workloads with a C_d greater than 2.

Our conclusions from the lock accessing workloads are:

- The ability of a policy to handle a high C_d is more important than how well it coschedules jobs.
- In spite of the extra overhead due to spinning incurred by RRprocess and RRjob, they outperform SNPF and PSNPF when C_d is larger than 2.
- For very high lock demands, coscheduling outperforms RRprocess, but not RRjob.
- An intelligent thread package is a requirement if dynamic-partitioning is to be implemented.

4.4. Comparison of dynamic-partitioning and RRjob

All the results so far tend to show that either the RRjob or dynamic-partitioning policies perform best. Figure 4.4 plots mean response time versus offered load for RRjob and dynamic-partitioning assuming a lock synchronization job model with job lock demand equal to 75%. The offered load is the load that would results if there was no extra demand caused by spinning for the locks. The workload is the correlated_low_C_n workload with $C_v = 3$ and $t = 2$. The top two curves assume a process may be descheduled while holding a lock. Dynamic-partitioning deschedules process much less frequently than RRjob, but when it does, it takes much longer for the process holding the lock to be rescheduled. At a high offered loads dynamic-partition* becomes much worse than RRjob. RRjob. At an offered load of 81.7% dynamic-partition* has a mean response time of 2417.5, RRjob has a mean response time of 21.4. The bottom two curves assume that a process holding a lock is never descheduled. For RRjob this is achieved assuming that if a process is holding a lock when it is about to end its time slice the time slice is extended just until the process releases its lock. This is similar to the "non-relinquishing locks" of Presto [BeLL 88]. Similar results might also be achieved by smart application scheduling with a thread management package [ZaLE89]. Dynamic-partitioning simply never deschedules processes holding locks. The only time a process would ever be suspended is when it is safe to do so. The curves for RRjob and dynamic-partition when processes holding locks are not descheduled are similar until very high loads where the assumption of jobs being held back in a load queue causes degradation.

A more detailed study of RRjob and dynamic-partitioning is needed to determine which is the better policy. Both have different strengths and weaknesses, as discussed below.

A major drawback of dynamic-partitioning is how it handles blocking. One example of a blocking is waiting for I/O. The problem occurs when the number of processors allocated to a job is roughly equal to the number of processes. In the current policy, when all of a job's processes are blocked, the processor guaranteed to that job sits idle. This could cause a serious performance degradation if the system is running jobs that frequently block. RRjob on the other hand skips unavailable processes when allocating time slices so blocking is not a problem.

A major strength of dynamic-partitioning is that it context switches less often than RRjob. As a result there is much less time wasted for the context switch and for reloading the cache. How significant this strength is requires further study.

Another issue is how dynamic-partitioning should handle having more jobs than processors. Our studies have shown that excluding jobs from service until jobs complete can hurt performance. An alternative is to allow the system to have more runnable jobs than processors. In this case time slicing must be done and then the benefits of not having to rebuild the cache diminish under heavy loads.

The dynamic-partitioning policy requires that a thread management package is used to run jobs. In addition, Tucker and Gupta propose partitioning the processors into two sets, one for jobs that use the thread package and one set for jobs that are not under the control of a thread management package. This is necessary for operating system processes and jobs that are submitted without using the thread package. Such a static partition would result in a load balancing problem and further degrade performance. RRjob works without a thread management package. Using one might improve performance by preventing descheduling of processes which if descheduled would slow down the rest of the job, but the basic policy does not require such a package.

5. Conclusions and Future Work

We have studied four classes of scheduling policies under two types of workload models, without and with lock synchronization.

Regarding the policies compared in our study, we have reached the following general conclusions. First, in contrast to earlier work, we find that SNPF does not perform well for workloads where coefficient of variation of service demand is larger than 1 or 2, even if there is a linear correlation between job service demand and number of processes. Second, for independent processes or lock synchronization, the RRjob and RRprocess policies generally have higher performance than the coscheduling policies, although they are not specifically designed for coscheduling. We expect that for barrier synchronization coscheduling or intelligent thread scheduling will be more important. Third, we find that the RRjob and dynamic-partition policies, which allocate an approximately equal fraction of the processing power to each job in the system, rather than to each process in the system, perform best under nearly all workload assumptions we considered. We note that the RRjob policy has not previously been examined. Determining which of the RRjob and dynamic-partition policies is preferable requires further investigation.

We have seen that workload assumptions can greatly affect the performance of the policies studied. For example, it is possible to devise workloads such that the SNPF scheduling policy performs very well (e.g, the correlated_high_C_n workload), and to devise workloads where the dynamic-partitioning policy performs poorly (e.g., the uncorrelated_high_C_n workload). However, overall, and considering workload models which we believe are more realistic, we conclude that SNPF is generally not a promising policy whereas dynamic-partition is one of the two most promising policies studied.

More generally, it appears to be important to use a scheduling policy that is relatively insensitive to the coefficient of variation in job service demand. These policies include the coscheduling, round-robin, and dynamic partitioning policies studied in this paper. Minimizing spin-waiting appears to improve performance at very high lock demands, but even then not as effectively as handling the variance in job demand. For barrier synchronization we expect minimizing spin-waiting to be more important.

References

[BeLL 88] Bershad, B.N., Lazowska, E.D., Levy, H.M., "PRESTO: A System for Object- Oriented Parallel Programming," Software: Practice and Experience 18,8, August 1988, pp. 713-732.

[Doep 87] Doeppner, T.W., "Threads: A System for the Supporst of Concurrent Programming," Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[EaZL 86] Eager, D.L, Zahorjan, J., Lazowska, E.D., "Speedup Versus Efficiency in Parallel Systems," Research Report 86-12, Dept. of Computational Science, University of Saskatchewan, Saskatoon, Canada, 1986 [to appear in IEEETC].

[Gust 88] Gustafson, J.L., "Reevaluating Amdahl's Law," Communications of the ACM, May 1988.

[Klei 76] Kleinrock, L., Queueing Systems, Vol. II: Computer Applications, John Wiley and Sons, 1976.

[LiCh 89] Lionel, M.Ni., Ching-Farn, E.WU, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems," IEEE Transactions on Software Engineering, March 1989, pp. 327-334.

[Livn 88] Livny, M., DeNet User's Guide, Version 1.0, Computer Sciences Deptartment, University of Wisconsin, Madison, 1988.

[NeTT 87] Nelson, R., Towsley, D., Tantawi, A.N., "Performance Analysis of Parallel Processing Systems," IEEE Transactions on Software Engineering, 1987.

[Maju 88] Majumdar, S., "Processor Scheduling in Multiprogrammed Parallel Systems," Ph.D. Thesis, Research Report #88-6, Department of Computational Science, Saskatoon, Saskatchewan, Canada, April 1988.

[MaEa 89] Majumdar, S., Eager, D., Private communications, 1989.

[MaEB 88] Majumdar, S., Eager, D., Bunt, R., "Scheduling in Multiprogrammed Parallel Systems," Proc. of ACM SIGMETRICS 1988 Conf. on Measurment and Modeling of Computer Systems, Santa Fe, NM, May 24-27, 1988, pp. 104-113.

[Oust 82] Ousterhout, J., "Scheduling Techniques for Concurrent Systems," Proc. of Distributed Computing Systems Conference, 1982, pp. 22-30.

[Shra 68] Shrage, Linus, "A Proof of the Optimality of the Shortest Remaining Processing Time Discipline," Operations Research, 1968, pp. 687-690.

[SaCh 89] Sauer, C.H., Chandy, K.M., "Computer System Performance Modeling", Prentice-Hall, 1981, page 16.

[TuGu 89] Tucker, A., and Gupta, A., "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," Proc. 12th ACM Symposium on Operating System Priciples, December 1989.

[ZaLE 89] Zahorjan, J., Lazowska, E.D., Eager, D.L., "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems," Technical Report 89-07-03, Department of Computer Science, University of Wshington, Seattle, July 1989.

[ZaLE 88] Zahorjan, J., Lazowska, E.D., Eager, D.L., "Spinning Versus Blocking in Parallel Systems with Uncertainty," Proc. of International Seminar on Performance of Distributed and Parallel Systems, Kyoto Japan, December 7-11, 1988, pp. 445-462.