Performance of Hierarchical Processor Scheduling in Shared-Memory Multiprocessor Systems[*]

Sivarama P. Dandamudi
School of Computer Science
Carleton University
Ottawa, Ontario K1S 5B6, Canada

Samir Ayachi
Northern Telecom
Ottawa, Ontario
Canada

TECHNICAL REPORT TR-96-21

Abstract

Processor scheduling policies can be broadly divided into space-sharing and time-sharing policies. Space-sharing policies partition system processors and each partition is allocated exclusively to a job. In time-sharing policies, processors are temporally shared by jobs (e.g., in a round robin fashion). Space-sharing policies can be either static (processor allocation remains constant during the lifetime of a job) or dynamic (processor allocation changes in response to changes in job parallelism). Equipartition is a dynamic space-sharing policy that has been proposed and studied extensively. Among the time-sharing policies, job-based round robin policy (RRJob) has been shown to be a very good policy. Performance analysis of these two policies suggests that Equipartition policy performs well at low to moderate system loads and is extremely sensitive to system overheads and variance in service demand of jobs. RRJob performs better when there is a high variance in service demand and at high system loads. Furthermore, these policies have been proposed for small-scale shared-memory systems and require a central run queue and/or central scheduler. The central queue/scheduler poses serious scalability problems for large-scale multiprocessor systems.

In this paper we propose a new multiprocessor scheduling policy that combines the merits of space-sharing and time-sharing policies while eliminating the contention for the central queue/scheduler. The new policy — called hierarchical scheduling policy (HSP) — uses a hierarchical run queue organization to take advantage of both temporal and spatial partitioning to allocate processing power amongst jobs waiting for service. We show that the HSP policy is considerably better than the purely space-sharing and purely time-sharing policies over a wide range of system parameters.

Key words: Multiprocessor systems, Processor scheduling, Performance evaluation, Hierarchical scheduling, Space-sharing, Time-sharing.

1 Introduction

The problem of scheduling in multiprocessor systems has been widely investigated [1,3–6,9–15,17–25,27–32]. Shared-memory systems can be classified into uniform memory access (UMA) systems or non-uniform memory access (NUMA) systems. UMA architectures are suitable for small systems whereas NUMA architecture is typically used for large systems [24]. The focus of this study is on shared-memory NUMA systems. Some examples of multiprocessors that belong to this category are the BBN Butterfly, IBM RP3, the Stanford DASH, the Toronto Hector, and the Illinois Cedar.

Processor scheduling policies can be broadly divided into space-sharing and time-sharing policies. Space-sharing policies partition the system processors and each partition is allocated exclusively for a job. In time-sharing policies, processors are temporally shared by jobs (e.g., in a round robin fashion). Space-sharing policies can be either static or dynamic. Static policies allocate a fixed number processors for a job and this allocation remains constant during the lifetime of the job. In dynamic policies, processor allocation varies during the lifetime of a job as it responds to the varying degree of a job's parallelism. This

[*] A preliminary version of this paper appears in the *Proc. IEEE Symposium on Parallel and Distributed Processing,* New Orleans, 1996.

This report can be obtained form `http://www.scs.carleton.ca`.

implies that sometimes processors are taken away from a job, which is expensive for NUMA systems. While the dynamic policies are shown to perform well in UMA systems, dynamic policies involve high overhead in NUMA systems. Equipartition is a dynamic space-sharing policy that has been proposed and studied extensively. Among the time-sharing policies, the job-based round robin policy (RRJob) is shown to be a very good policy. Both policies try to allocate equal amount of processing power to jobs, which has been shown to be necessary in order to yield good performance in multiprocessor systems.

Performance analysis of these two policies suggests that Equipartition performs well at low to moderate system loads and is extremely sensitive to system overheads and variance in job service demand. RRJob performs better when there is high variance in service demand and at high system loads. Furthermore, these (and most other) policies have been proposed for small-scale shared-memory systems that use a central run queue and/or central scheduler. It has been shown that this central queue/scheduler tends to become a bottleneck at moderate to high system utilizations [6]. Thus the central queue/scheduler poses serious scalability problems for large-scale multiprocessor systems.

We propose a new multiprocessor scheduling policy that combines the merits of both space-sharing and time-sharing while minimizing/eliminating the disadvantages associated with these policies (e.g., the new policy eliminates the contention for the central queue/scheduler). The new policy — called the hierarchical scheduling policy (HSP) — uses a hierarchical run queue organization to take advantage of both temporal and spatial partitioning to allocate processing power amongst jobs waiting for service and uses self-scheduling. We show that the HSP policy is considerably better than the purely space-sharing and purely time-sharing policies over a wide range of system parameters. In the remainder of the section, we will briefly review the previous work that is directly related to our study.

1.1 Previous Work in Multiprocessor Scheduling

In this section some common time- and space-sharing policies are briefly reviewed. In [17], one of the earliest works in multiprocessor scheduling, a policy known as coscheduling has been proposed. In this policy a set of processors is allocated to all the computational units (which we refer to as tasks) of a parallel job at the same time. Tasks are allowed to execute on the processors for a fixed time slice and then preempted simultaneously — allowing the tasks of the next job to be scheduled on the same processors. Even though this policy reduces synchronization delays by scheduling communicating tasks together it has the drawback of low system utilization [1, 12]. Improvements have been suggested to the scheme to increase system utilization [12]. Also in [12] the performance of a round robin policy known as *RRjob* was studied. The *RRjob* policy was different from the usual round robin scheduling policy in that it allocates equal processing power on a per job basis rather on a per task basis. Studies in [12] indicated that this approach demonstrated considerable performance improvements over a number of policies like round robin (also referred to as *RRtask* or *RRprocess*), Smallest Number of Processes First (SNPF) and Shortest Cumulative Demand First (SCDF). The *RRjob* policy had first been proposed in [13] where the performance of SNPF, RRtask, SCDF and FCFS policies was compared to investigate whether policies that used information about job characteristics had any performance advantage over policies that did not. The study in [13] concluded that policies like *RRprocess* and First-Come-First-Served (FCFS) that did not use job characteristics had inferior performance to policies like SNPF and SCDF that used information about job characteristics to make scheduling decisions.

Processors can also be spatially partitioned amongst waiting jobs by the scheduler. Several studies have investigated the performance of space-sharing policies [14, 27, 29]. There is consensus in the literature that policies that can alter processor allocation to the tasks of a job dynamically have performance superior to that of static partitioning policies. It has also been shown that the best performance is observed when the number of processors allocated to a job is equal to the number of tasks in the job at that stage in the job's execution [27].

However, as mentioned earlier, most of the above policies use the centralized queue organization (or some variation of it) to place jobs in before they can receive service. Usually the effect of queue access contention is not factored into the simulation experiments performed to study their performance. In real systems the overhead due to access contention is considerable [6]. Another factor that could lead to performance deterioration in these policies is that scheduling decisions are made by a central scheduler that also tends to become a bottleneck at moderate to heavy system loads.

The rest of this paper is organized as follows. The next section describes the new hierarchical scheduling policy (HSP) that is proposed in this paper. Section 3 describes the space-sharing policy (Equipartition) and the time-sharing policy (RRJob). Section 4 describes the system and workload models used in this paper. Sections 5 and 6 present the results of simulation experiments done to compare the performance of HSP with the time-sharing and space-sharing policies described in Section 3. Section 7 concludes the paper and provides some pointers to future work.

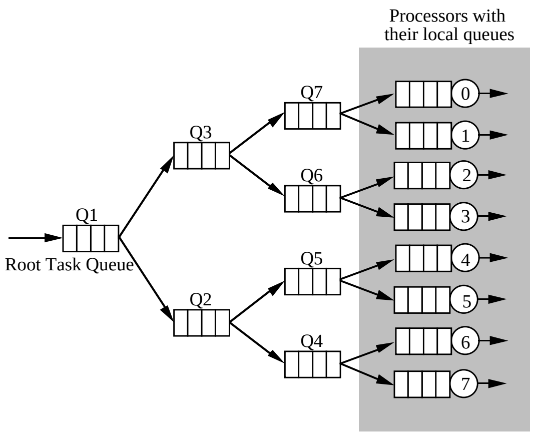

Figure 1: Hierarchical task queue organization for $P = 8$ processors with a branching factor $B = 2$

2 The Hierarchical Scheduling Policy

The objective of this study is to devise a multiprocessor scheduling policy that inherits the good features of the space- and time-sharing policies. It is also required that it should eliminate queue access contention and distribute scheduler functions among processors thus removing the major system bottlenecks that cause performance deterioration.

The hierarchical queue organization that was first proposed and analyzed in [6] is used as the queueing model for HSP. Figure 1 shows the hierarchical queue organization for a system with $P = 8$ processors. It has been shown in [6] that the performance of a hierarchical queue is very close to that of an ideal centralized queue[1]. It has also been shown that the hierarchical queue organization has the desirable load sharing property (as in the centralized organization) and is scalable (like the distributed organization).

Any hierarchical queue can be described using two parameters:

- The branching factor (B) which is the out-degree of each non-leaf node in the queue (see Figure 1)
- The transfer factor (Tr) which determines how many tasks are moved down from a parent queue to one of its children queues. The transfer factor is defined as follows:

$$Tr = \frac{\text{number of tasks moved one level down the tree}}{\text{number of processors below the child task queue}}$$

The remainder of this section describes the hierarchical scheduling policy in detail.

In the hierarchical queue the leaf nodes are the processors and the non-leaf nodes are shared memory modules. The memory modules contain task queues into which waiting tasks are placed before being scheduled to run. In any shared-memory system with P processors and P memory modules the maximum number of non-leaf nodes required will be $P - 1$ (when $B = 2$). Therefore each task queue can be placed in a different memory module. This is one of the key factors that reduces access contention.

As mentioned earlier, HSP is a self-scheduling policy. This implies that there is no central scheduler making processor allocation decisions. The policy works in the following manner. There is a system parameter called the multiprogramming level (MPL) that is associated with each processor. The MPL is the maximum number of tasks that can be present in a processor's ready queue. Whenever the number of tasks in a processor's ready queue is less than the MPL the processor polls the task queue immediately above it in the hierarchy. If the polled queue has tasks then a certain number of tasks is moved down to the processor's ready queue. This number is determined using:

min(MPL $-$ number of tasks in ready queue, number of tasks in polled queue)

If the parent node of a processor does not have any tasks in its queue then the processor locks this node and polls the next node in the hierarchy, i.e., its parent's parent. If tasks are found at this level then a certain number of tasks are moved down one level of the tree. This number is give by:

[1] An "ideal" centralized queue is one in which there is no access contention overhead.

min($Tr \times$ number of processors under this node, number of tasks in polled queue)

However if no tasks are found then the search process is repeated till a node with waiting tasks is reached. If the root node is reached and no tasks are found then it indicates that the system is waiting for new jobs to arrive.

Let us look a little more closely at the process of task transfer. Depending on the branching factor B of the queue each node may be polled by at most B processors at the same time. While this is better than the centralized queue in which all P processors might try to access the queue simultaneously, there is still the problem of maintaining mutual exclusion between processors. This is achieved by associating a locking mechanism with each task queue. A modified version of the MCS lock [16] is used to lock the task queues. The reason the MCS queue model was chosen is that it has negligible overheads since each processor waiting on the lock needs to poll only a local flag to determine whether the lock is available or not. Refer to [16] for further details on the MCS model. Therefore, whenever a processor requires access to a task queue it tests the lock. If the lock is available it is acquired and set to 'locked'. Other processors that try to access the same queue will find it locked and add themselves to a list of processors waiting for the queue to become free.

This locking mechanism is essential for the functioning of the scheduling algorithm and it also has the beneficial effect of reducing the number of processors that access nodes at higher levels. This is because a processor that was not able to acquire a lock waits on the same lock till it becomes free and is not allowed to poll other nodes in the hierarchy, thus reducing the network traffic and access contention.

It can be seen from this description that the number of processors polling nodes in the tree decreases as the distance from the leaf level increases. Processors that find tasks at a particular node carry down more tasks than they need to process so that other processors that are waiting on queues at lower levels need not search the entire tree. A processor that retraces its path down the tree checks the wait queue at each node and sets a flag informing the processor at the head of the queue that the lock is now available. It also leaves behind some tasks at each level. The number of tasks left behind is given by :

Total tasks being carried down $-$ ($Tr \times$ Number of processors in child subtree)

When a processor is informed of the availability of a lock that it has been waiting on it acquires the lock and checks to see if there are any tasks at the current level. If not the process of searching the hierarchy is started again. If tasks are found they are moved down the tree in the manner described above.

There are several overheads associated with the actions described above, and some of these overheads are now described. The main overheads associated with the processors' scheduling actions are due to:

- Adding and removing tasks from a queue.
- Polling a queue to see if tasks are available.
- Context switching between tasks in processor ready queues.
- Delay required to inform waiting processors of queue availability.

In the hierarchical policy, the contention for the root queue is explicitly modelled. Measurements in actual systems indicate that memory access overheads and context switching overheads lie in the range of 2%–8% of the average task service demand [14]. For the simulation experiments performed in this study the total overhead is an important factor in order to compare the performance of HSP with previously studied policies.

3 Other Scheduling Policies

This section briefly describes the two other scheduling policies — one space-sharing policy (Equipartition) and one time-sharing policy (modified RRJob) used to compare the performance of the hierarchical policy.

3.1 Equipartitioning

This is a modified version of the dynamic policy first outlined in [29], where it was compared with static allocation policies. This policy uses a central queue and a central scheduler for making allocation and placement decisions.

Each job informs the system of its processor requirement. This requirement is usually equal to the parallelism of the job. The scheduler attempts to satisfy the job's requirement if there are processors available. If no processor is available, one is taken away from a job that is holding more than one processor.

At moderate to high system loads this leads to a situation where each job has at least one processor. If there are more jobs in the system than there are processors then the extra jobs have to wait in the central queue. This policy is a dynamic space-sharing policy. It is dynamic in that the processor allocation changes with job arrival, job departure, or a change in a job's requirement. It is a space-sharing policy since each job is exclusively allocated a certain number of processors.

A modification[2] has been made in this study from the original scheme in [29]. In the original scheme, when a new job arrived and there were no processors available, a processor was taken away from an executing task and allocated to the new job. This led to a high frequency of pre-emption at all but low system loads. In this study, all tasks that are allocated a processor, run on that processor to completion. This is expected to improve performance, especially if pre-emption overheads are not negligible. However, to ensure that new jobs do not have to wait for too long, the central scheduler gives preference to jobs that have not been allocated any processors so far. This implies that whenever processors become available, the scheduler first checks to see if there are any jobs in the queue that have not been allocated any processors. If such jobs are present then the processor is given to that job, which might not necessarily be at the head of the queue. Note also that a job's processor allocation changes during its execution. For instance, if a job required n processors, but was allocated only p, where $n > p$, then once the p tasks of the job are done, it is not necessary that the same processors will be reallocated to the same job — because, in the meantime, new jobs might have arrived in the system and these jobs are given preference over older jobs that have received some service already.

The equipartition policy was included in this study for several reasons. It is a purely space-sharing policy. It exhibits excellent performance at low system overheads for all load levels, but is extremely sensitive to the scheduling overhead especially as system load increases.

All details regarding the actual simulation are identical to the other policies. Note that since this is a run-to-completion, or RTC policy, there is no multiprogramming level associated with the processors. Two main overheads are associated with this policy:

- An allocation delay D_{alloc} is incurred when a task is allocated to an idle processor and,
- A reallocation delay $D_{realloc}$ is incurred if a processor is taken away from a job and allocated to a new job.

3.2 Modified RRjob

In the modified RRjob policy, the centralized queue organization is used. Unlike the original scheme of [12], however, there is no scheduler that makes allocation decisions. Instead of having a central scheduler, the demand driven approach of self scheduling is used, similar to HSP. Also, in the original scheme of [12] there was no MPL associated with the processors, as it is in this case.

When a job arrives at the system, it is added to the queue behind all waiting jobs. Processors pick up tasks from the queue, equal in number to their multiprogramming level. Depending on the parallelism of the job, the quantum of service given to tasks from different jobs is different. Smaller jobs are given larger processor quanta in order to allocate equal processing power to each job. The quantum size Q for the tasks of a job are calculated using the formula:

$$Q = \frac{P}{\mathit{Avg}} \times q,$$

where P is the number of processors in the system, Avg is the parallelism of the job and q is the basic quantum size. The parallelism of the job is bounded by the system size so that q becomes the minimum quantum length for the system. At moderate to high load levels processors are busy most of the time and check the job queue only when a task blocks or finishes execution. Whenever the job queue is checked, it is locked to prevent other processors from accessing the queue at the same time. Each time a processor accesses the job queue, it attempts to remove enough tasks so that it maintains the number of tasks in its ready queue at its MPL. However, when the system is in a steady state, each access brings down only one task, since queue accesses are made each time a task departs.

[2] This modification does not significantly affect the performance of the policy as demonstrated in Section 5.1. For example, notice the flat response time before saturation occurs in Figure 3

4 System and Workload Models

With numerous kinds of parallel architectures and an even greater variety in the kinds of parallel applications existent today, it is difficult to devise a scheduling policy that is "optimal" in all cases. It is necessary, therefore, to define clearly the class of parallel systems and the kind of parallel workload for which a particular policy is intended. The following paragraphs describe the system and workload models used in this study.

4.1 System Model

We assume that the system consists of P homogeneous processors that are interconnected with P memory modules using a multistage interconnection network. Note that each of the P memory modules is local to a processor in the system. Jobs arrive for service at the system in a Poisson stream with rate λ and are queued.

Several overheads need to be factored into the system for a realistic analysis. Some system specific overheads that are incorporated into the model are: the delay involved in searching the task queues, the delay involved in removing a set of tasks from the queue and the preemption overhead when a task is descheduled at the end of its quantum.

These delays are usually very system specific and difficult to generalize, but normally do not exceed a small fraction of the average task service time. The performance of HSP is studied for a range of overheads, varying between 2% to 8% of the average task service time. It is believed that this is a realistic estimate for the system model used [14, 29].

If the scheduling policy being studied uses the round robin mechanism to schedule tasks in the processor ready queues, then another system specific parameter that comes into play is the length of the round robin quantum. Simulation experiments have also been performed to examine the effect of the round robin quantum length on the average job response times.

4.2 Workload Model

Parallel applications can be modelled at various degrees of detail [22]. The workload used in this study is similar to the workloads of [4], [12] and [13]. The two main features of parallel applications that are of interest in this study are the average parallelism and the total service demand. We have considered two types of workload classes — the independent workload model in which the tasks of a job execute independently of each other and synchronize only once at the end, and the lock accessing workload model in which some form of communication or shared data access is required among the tasks of a job.

For the independent job model or Fork-Join type of job, the job is described completely by the following parameters (lock accessing workload model is described in Section 6):

1. The cumulative service demand for the job, D (the actual job service demand is taken from a two-stage hyperexponential distribution depending on the coefficient of variation of the service time)
2. The average parallelism of the job Avg i.e., the average number of tasks in a job (the actual number of tasks is generated from a two-stage hyperexponential generator)
3. The job's ERF, or *execution rate function*, first described in [7], and used by us to divide the cumulative service demand among the tasks of the job.

Another feature of the workload model is that, it is assumed that the cumulative demand of the job is not split equally among all tasks. This is because, there are always overheads associated in splitting the total work of an application into smaller units of computation. Therefore, the execution rate function described in [7] and also used in [4], is used here to determine the service demand of each task in a job, given the cumulative service demand of the job. The calculation is done using the following formula:

$$\gamma(k) = \frac{(1+\beta)k}{k+\beta}, k = 1, 2, \cdots, P.$$

In the above equation, β determines how the total service demand is divided among the tasks. Lower β values lead to higher task service demands and vice versa. k is the parallelism of the job. The task service demand d is obtained using:

$$d = \frac{D}{\gamma(k)}$$

where D is the job service demand. Figure 2 shows a graph of $\gamma(k)$ as a function of the average parallelism k for various values of β. From the graph it can be seen that small values of β result in small values of $\gamma(k)$, resulting in large task service demands. The dashed line in the figure shows the case when the cumulative demand of the job is split equally among all the tasks.

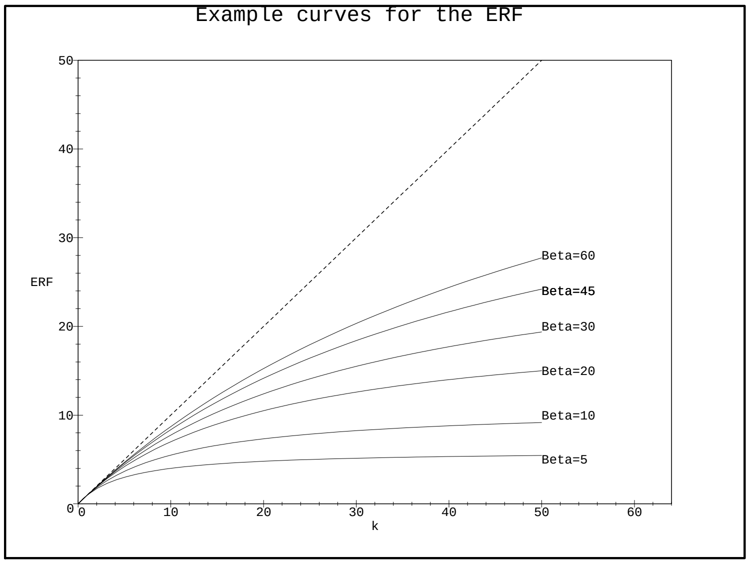

Figure 2: Example curves for ERF $\gamma(k) = \frac{(1+\beta)k}{k+\beta}$

5 Performance Analysis

In this section the results of simulation experiments carried out to evaluate the performance of HSP are presented. The performance of HSP under different simulation parameters, was studied and compared to the performance of Equipartition, and the modified RRjob.[3] Several classes of experiments were performed; these include experiments that study the effect of scheduling and contention overheads on the response time, the effect of the coefficient of variation of the cumulative job service demand C_{v_d}, and the effect of other factors like the round robin quantum, the ERF factor (β), and the average parallelism *Avg*.

The default system and workload parameters for the simulation experiments are given in Table 1. Note that in this table, all overhead values are expressed as a percentage of the average task service time. The *notification overhead* is the delay involved in informing a processor in a wait queue of the availability of the lock it was waiting on. The arrival rate λ is increased, keeping μ constant, to increase the load on the system.

5.1 Effect of scheduling overhead

Since the main structural difference between HSP and the other policies is that the latter use a centralized queue organization, the most important aspect that needs to be studied is the effect of overheads on policy performance. Most studies that have analyzed policies that use centralized queues do not take into account the queue access contention. In this study queue access contention was factored into the simulation modelling of all policies that were investigated. Apart from the access contention other overheads were also involved and these have been summarized in Table 1. Experiments were carried out for three levels of total overhead — low, medium and high. The low overheads are shown in Table 1. The medium and high overheads were twice and three times the low values, respectively. The medium overhead results are omitted for the sake of brevity — see [2] for details. The results of these experiments are shown in Figure 3. Note the utilization reported in this section is computed as follows:

$$\text{utilization} = \frac{\lambda}{\mu \times \gamma(avg) \times P}$$

The actual system utilization will be higher than this because of the ERF factor bloating the task service times.

[3] Queue access contention is included in the analysis

Para-meter	**Description**	**Default value**
P	System size (# of processors)	64
D_{qp}	Queue polling overhead	0.04%
D_{qa}	Queue access overhead	0.04%
D_{cs}	Context switching overhead	0.02%
D_{no}	Notification overhead	0.001%
D_{alloc}	Processor allocation delay	1%
$D_{realloc}$	Processor reallocation delay	2.5%
Tr	Transfer factor	1
B	Branching factor	8
μ	Job service time	10.00
avg	Avg. parallelism (# of tasks)	8
β	ERF factor	30
C_{v_d}	Coeff. of var. of demand	1
C_{v_n}	Coeff. of var. of parallelism	1
MPL	Multiprogramming Level	2
q	Round robin quantum	0.025
N_j	Number of jobs per batch	10000

Table 1: Default parameter values for simulation experiments

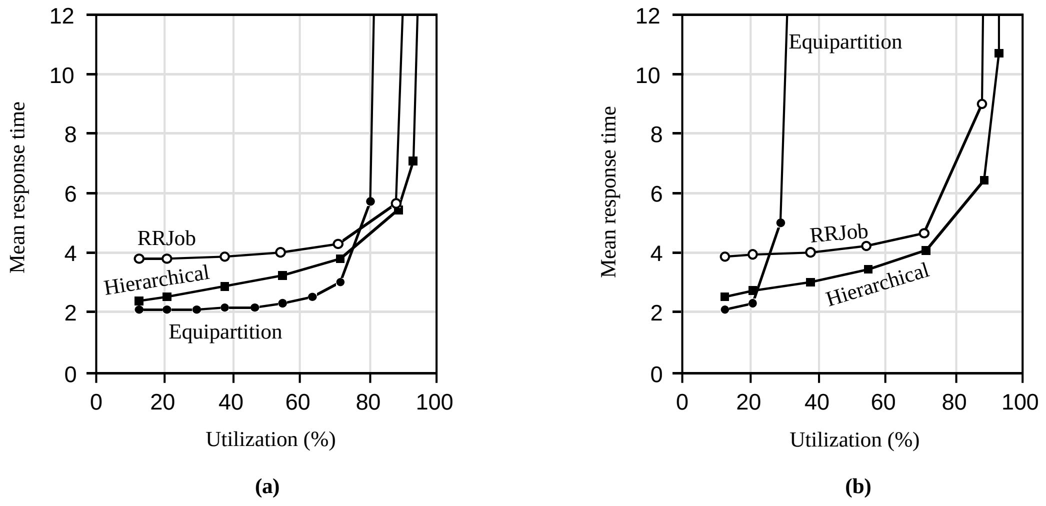

Figure 3: Response time versus utilization for (a) low overhead; (b) high overhead

We can make several observations from the data presented in Figure 3.

- When the overhead is low (as exemplified by the data shown in Figure 3a), Equipartition policy provides the best performance unless the system load is high. Since the average parallelism used in this experiment is 8, it is possible to schedule 8 such jobs, on the average, using space-sharing Equipartition policy. At low to moderate system loads this is the best policy. At these system loads, the hierarchical and RRJob policies suffer in performance because of the multiprogramming level (MPL) of 2 used here. This leads to a situation where some processors have two tasks to process while others are idling. We need to have at least 2 for MPL in order to give preemptive priority to smaller tasks[4]. This

[4] At MPL=1, it degenerates to FCFS policy.

preemption property becomes more important when there is high variance in job service demands. Since we are using exponential job service demand (i.e., $C_{v_d} = 1$) in this experiment, the results are biased in favour of Equipartition policy. Section 5.2 discusses the impact of the variance in job service demand.

- At high system loads, even if the overheads are low, the performance of Equipartition policy deteriorates rapidly. This is mainly due to access contention for the central scheduler and run queue. As shown in Figure 3, Equipartition policy is sensitive to scheduling overheads. For example, when the overheads are low, system saturation occurs at about 80% system utilization; this reduces to about 30% when the overhead is high. This can be seen by noting that the job arrival rate λ_{sat} at which the Equipartition scheduler becomes the bottleneck when only allocations are performed is given by

  $$\lambda_{sat} = \frac{\mu \times \gamma(avg)}{D_{alloc} \times avg}$$

  Eliminating γ from the above equation leads to

  $$\lambda_{sat} = \frac{\mu \times (1+\beta)}{D_{alloc} \times (avg+\beta)}$$

  The corresponding value when reallocations are performed is given by

  $$\lambda_{sat} = \frac{\mu \times (1+\beta)}{D_{realloc} \times (avg+\beta)}$$

  We have observed that the scheduler performs approximately 40% allocations and 60% reallocations. Given these values, the above two equations predict that saturation occurs at about 82% utilization when the low overhead values are used; the corresponding value for high overhead values is about 27%. This agrees with the saturation values shown in Figure 3.

  The other two policies show substantial resilience to scheduling overheads. In particular, the hierarchical scheduling policy exhibits substantial performance superiority over RRJob policy over a wide range of system loads and scheduling overheads. RRJob suffers in performance as the scheduling overheads increase because of the central queue access contention.

- The data presented in Figure 3 suggests that the hierarchical policy does inherit the merits of the space-sharing and time-sharing policies. At low system loads it behaves like a space-sharing (Equipartition) policy and at high system loads it behaves like a time-sharing (RRJob) policy. The policy as implemented now does have the disadvantage of inferior performance as compared to that of Equipartition policy at low to moderate system loads (depending on scheduling overheads). Part of the reason is that we have used a static transfer factor Tr value in these experiments. This causes load imbalance at low to moderate system loads by moving more tasks to one part of the system leaving the other part underloaded. This situation can be remedied by using a dynamic task movement policy (effectively using a dynamic Tr value) that is dependent on the number of tasks present at the parent node. In this policy, the number of tasks moved one level down the tree is proportional to the number of tasks present in the parent queue. Thus, if there are fewer tasks at the parent queue, correspondingly fewer tasks are moved to its child queue. This task transfer policy also implements a load-dependent MPL. With these modifications, we expect that the performance of the hierarchical scheduling policy would be close to that of Equipartition policy.

5.2 Impact of variance in service demand

Often jobs that arrive for service at a multiprocessing system have coefficient of variation of demand (C_{v_d}) much greater than one [4]. It is, therefore, important to study the effect of service demand variance on the performance of any scheduling policy. This section presents the results of the simulation experiments that show the impact of C_{v_d} on the performance of the three scheduling policies. All these experiments were done at the low system overhead level in order to include Equipartition in the analysis[5].

Figure 4 shows the performance of the three policies as a function of system utilization when the service demand variance C_{v_d} is 3. The impact of changing C_{v_d} from 1 to 3 can be seen by comparing Figures 3a and 4. It can be observed from these

[5] Real system overheads are usually greater than this [14].

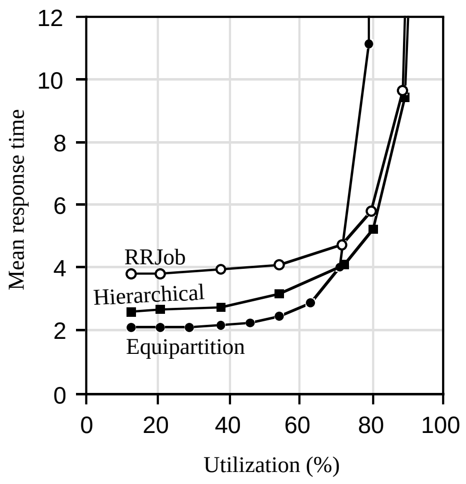

Figure 4: Response time versus utilization for service demand $C_{v_d} = 3$

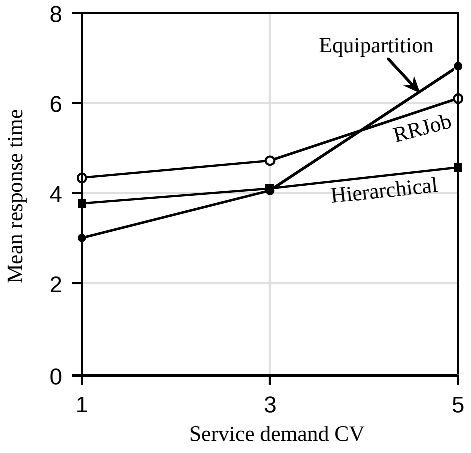

Figure 5: Response time versus service demand C_{v_d} at 72% utilization

two figures that, while all three policies experience performance deterioration, Equipartition suffers the most. This sensitivity is further demonstrated in Figure 5. The sensitivity of Equipartition can be mainly attributed to the fact that Equipartition uses a non-preemptive, run-to-completion FCFS policy. As a result some jobs that arrive for service might have very large service demands and so they might not allow smaller jobs to be serviced for longer times.

In addition, both Equipartition and RRJob also suffer from queue access contention as the variance in service demand increases. Note that a high CV implies that there is proportionately a large number of tasks with small service times and a comparatively small number of tasks with very large service times. As a result, the central run queue is accessed in a "bursty" fashion. This, therefore, increases access contention for the central queue on a temporal basis (even though the time-averaged behaviour might not significantly change). The hierarchical policy successfully avoids these two problems and, therefore, shows the least sensitivity to service time variance as demonstrated in Figure 5.

5.3 Effect of task granularity

For a given job, we can increase the job parallelism by moving toward finer task granularity. In our workload model, we use Avg parameter to model job parallelism. The previous results have assumed that the average parallelism Avg is 8. This section presents the impact of task granularity on the performance of the three scheduling policies.

The results are presented in Figure 6. It can be seen from this plot that the initial increase in job parallelism results in

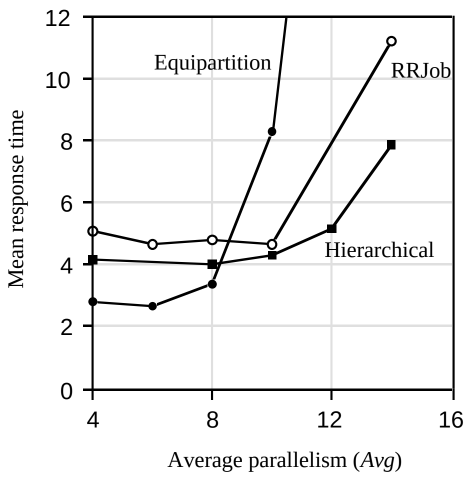

Figure 6: Response Time Vs. Avg. Parallelism,(avg), at 75% utilization

marginal performance improvement for all three scheduling policies. For example, the hierarchical policy benefits when the job parallelism Avg is increased from 4 to 8 while Equipartition policy benefits when Avg is changed from 4 to 6. This initial improvement in performance is due to reduced task sizes (i.e., finer task granularity). However, there is a cost associated with moving toward finer task granularity — the scheduling overhead increases. The amount of scheduling overhead depends on the type of policy considered. The initial improvement in performance is due to the fact that the performance gains (because of smaller sized tasks) more than offset the increase in the scheduling overhead.

However, if we increase job parallelism further, the scheduling overhead dominates and this deteriorates the performance. From Figure 6 it can be seen that Equipartition policy is much more sensitive to this overhead than the other two policies. Among the three policies considered here, the hierarchical policy exhibits the least sensitivity to this overhead.

An important conclusion that we can draw from this figure is that it is more critical to have an "appropriate" job parallelism when Equipartition policy is used. If the job parallelism is off this acceptable value, there will be severe performance penalty. On the other hand, the hierarchical policy exhibits a more robust behaviour to changes in job parallelism.

5.4 Effect of ERF Factor

The effect of the ERF factor (that is, β) is shown in Figure 7. Note that as the β value decreases, the task size increases for the same job service demand. That is, we introduce more overhead for parallel processing the job (see Figure 2). As a result, the actual utilization increases with decreasing β values. Recall that the utilization 75% stated in the figure legend is obtained form the input parameters as :

$$\text{utilization} = \frac{\lambda}{\mu \times \gamma(avg) \times P}$$

using $\beta = 30$. For β values lower than about 20, Equipartition performs worse than the hierarchical policy. This is because lower β values push the system toward higher system utilization and the performance of Equipartition is better at low to moderate system loads. Equipartition, however, maintains its performance superiority for larger β values.

5.5 Effect of Quantum Size

Quantum size is a parameter that is used in the hierarchical and RRJob policies but not in Equipartition policy. The sensitivity of these two policies to this parameter is shown in Figure 8. Both policies show initial improvement in performance as the quantum size increases even though the effect is more pronounced on the hierarchical policy. This is because too small a quantum size causes the context switch overhead to be high. However, increasing the quantum size causes the RRJob's performance t̊o deteriorate rapidly while the hierarchical policy exhibits a stable behaviour. The sensitivity of the RRJob policy to the quantum size is well-known. The surprising robustness of the hierarchical policy is due to its space-sharing aspect of the policy.

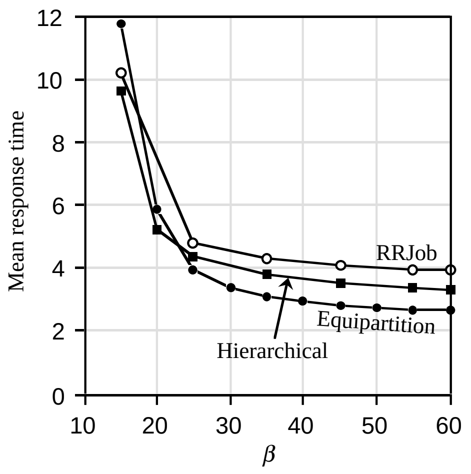

Figure 7: Sensitivity to ERF factor at 75% utilization

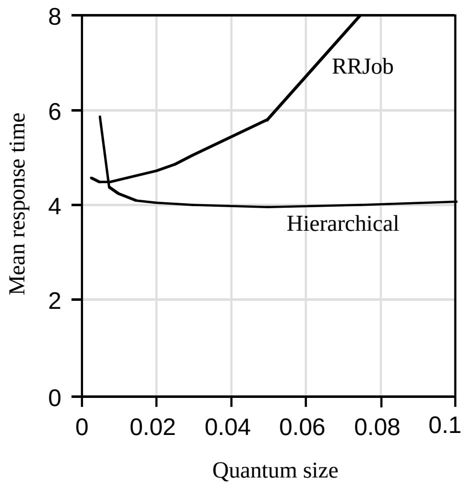

Figure 8: Sensitivity of hierarchical and RRjob policies to quantum size at 75% utilization

6 Performance with Lock Accessing Workload

This section first describes the lock accessing workload model and then discusses the results. Due to space limitation, only a few simulation results are presented here (complete results are available in [2]).

6.1 Lock accessing workload

In this workload model, each task is assumed to consist of a certain number of iterations. During each iteration, a task requires exclusive access to a critical section (e.g., shared data structure) that is shared among all the tasks of a job. In order to do this it must exclusively acquire the lock on the critical section. If the lock is not available immediately, the task is blocked and another task in the processor's ready queue is scheduled to run in its place. Figure 9 shows the structure of a single iteration of a task in this workload. Other job characteristics like Avg and D are generated in a manner similar to the independent workload model (described in Section 4).

Table 2 gives the additional parameters and their default values used in the simulation experiments for the results reported here. All other parameter values are fixed as in Table 1. The number of iterations per task is generated from a bounded uniform distribution between 1 and I_{max}. There is no delay associated accessing the critical section lock (i.e., $D_{lock} = 0$) because

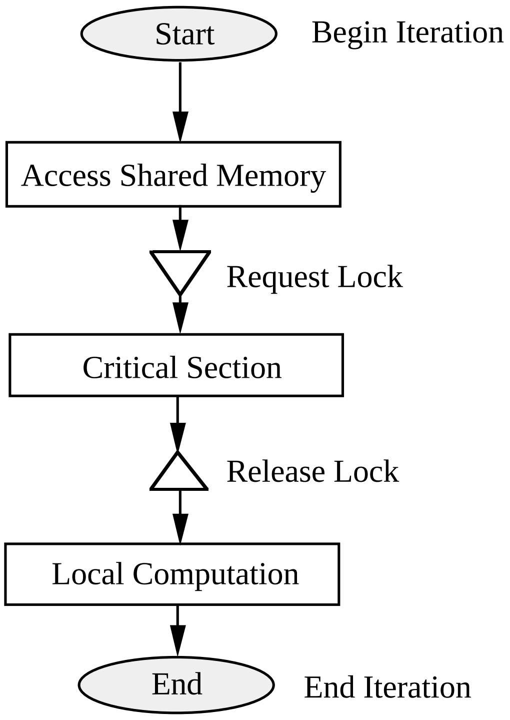

Figure 9: Task structure for one iteration

Parameter	Description	Default Value
I_{max}	Maximum number of iterations	15
D_{lock}	Delay to access critical section lock	0
T_{lock}	Critical section lock hold time	0.1 $\times q$

Table 2: Additional parameters for the lock accessing workload

each processor needs only to test a local flag to determine if the lock is available [16].

6.2 Results

Figure 10 shows the performance of the three scheduling policies as a function of system utilization for low and high scheduling overheads. The impact of synchronization can be seen by comparing the plots of Figure 10 with the corresponding plots of Figure 3 on page 8. All three scheduling policies exhibit increased sensitivity to scheduling overhead. This is mainly due to the inherent serialization involved in accessing the critical section. This serialization causes tasks of a job to wait longer causing the response times to increase (compared to that in the independent workload). The higher the overhead the more severe this problem is.

Another noticeable trend is that, when the system load is low, the difference in performance between Equipartition and the other two policies is more significant than that in the independent workload. This is because the hierarchical and RRJob policies maintain a multiprogramming level MPL of 2. As a result a task that is not able to acquire the lock would have to wait a longer time than in the space-sharing Equipartition policy. Thus, when the central scheduler and the run queue do not become a bottleneck, Equipartition policy performs better than the other two policies.

As discussed in Section 5.1, the performance of the hierarchical policy can be improved by using an adaptive task movement policy. Such a policy would have a greater impact on the lock accessing workload than on the independent workload.

Figure 11 shows the impact of the service time variance on the performance of the three scheduling policies. We have

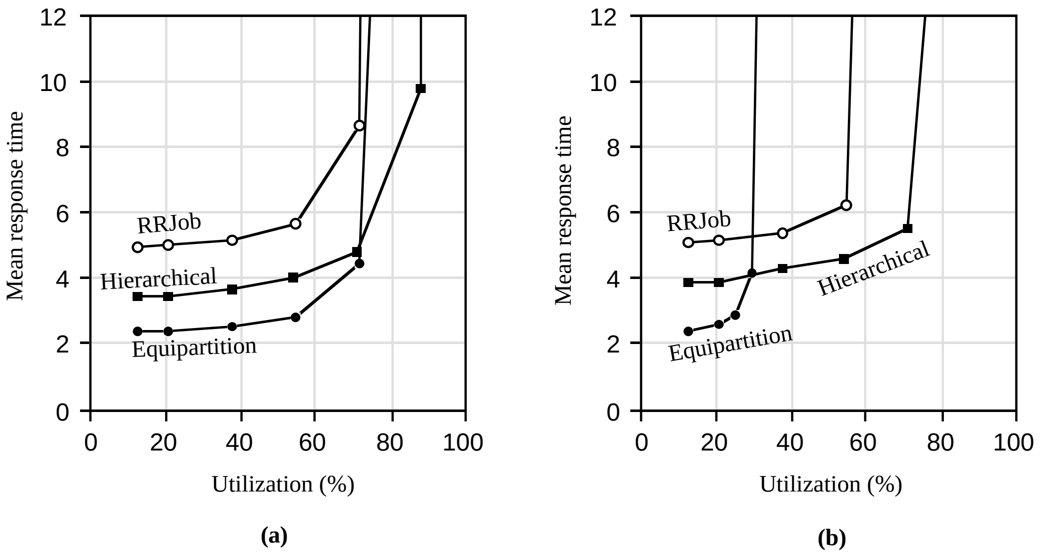

Figure 10: Response time versus utilization for (a) low overhead; (b) high overhead

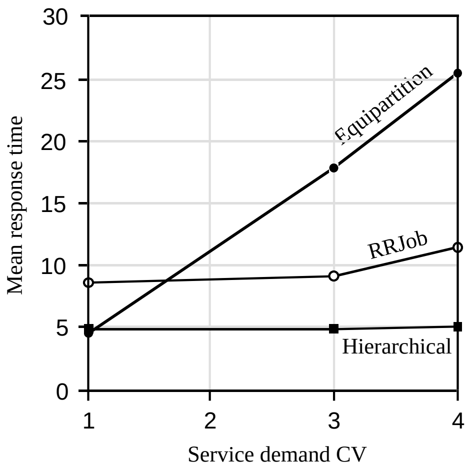

Figure 11: Response time versus C_{v_d} at 72% utilization

used low scheduling overhead in order to include Equipartition policy. The general trends observed in Section 5.2 for the independent workload are valid for the synchronization workload as well. The only surprise in this plot is the high degree of sensitivity exhibited by Equipartition policy to service time CV. This is mainly because, in this workload, the tasks wait for obtaining access to the user lock, which involves variable amount of queueing delay. Since Equipartition is non-preemtive, the processor is not released for other tasks.

7 Conclusions

We have presented a new multiprocessor scheduling policy that is suitable for large-scale multiprocessor systems. This policy is based on a hierarchical run queue organization. The proposed hierarchical scheduling policy (HSP) eliminates the contention for a central queue and a central scheduler and thus suitable for large systems. In addition, it combines the merits of space-sharing and time-sharing policies.

The main features of the new policy are the following.

- HSP eliminates contention for the central run queue that becomes a performance bottleneck in other common scheduling policies. Thus the HSP is suitable for large multiprocessor systems.
- HSP is a self-scheduling policy and processors look for work whenever they are idle. This eliminates the need for a central scheduler, making the system more scalable. Another advantage of self scheduling is that at moderate to high system loads a central scheduler might get saturated, which is avoided in self scheduling.
- Purely time-sharing policies are in general inferior to purely space-sharing policies when the service time variance is not high. On the other hand, space-sharing policies perform poorly when variance in service times is high. HSP combines space-sharing with time-sharing, and the results presented here indicate that as a result its performance is superior to the other policies over a wide range of system parameters. It should be noted, however, that Equipartition policy performs better than HSP policy at low system loads and low overheads.

In the preceding sections the performance of HSP was studied for a wide a range of system parameters. Observations regarding the performance of HSP are presented below.

- HSP is relatively less sensitive to scheduling and other system overheads compared to the other two policies studied here. Acceptable response times are obtained with HSP over a wide range of system utilization levels and overhead levels for both the independent and lock accessing workloads.
- HSP is also relatively less sensitive to the variance in job service demands for both workloads and over a wide range of system load levels. This is unlike the RRjob and Equipartitioning policies.
- The behaviour of HSP under varying conditions of job parallelism, round robin quantum size and the execution rate function was also studied and it was observed that in general, if the scheduling overheads were not negligible and the system load is not low, HSP exhibits better performance than Equipartition and RRjob policies.

The concepts of HSP can also be easily extended to distributed-memory systems and networks of workstations and this is intended to be the main focus of future research in the area of hierarchical scheduling.

Acknowledgement

We gratefully acknowledge the financial support provided by the National Sciences and Engineering Research Council and Carleton University. This work was done while the second author was at the School of Computer Science, Carleton University. A shorter version of this paper appears in *IEEE Symp. Parallel and Distributed Processing*, New Orleans, 1996.

References

[1] S. L. Au and S. P. Dandamudi, "The Impact of Program Structure on the Performance of Scheduling Policies in Multiprocessor Systems," *Int. J. Computers and Their Applications*, Vol. 3, No. 1, April 1996, pp. 17–30.

[2] S. Ayachi, *A Hierarchical Processor Scheduling Policy for NUMA Systems*, Master's Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 1995.

[3] S. P. Cheng, and S. P. Dandamudi, "Scheduling in Parallel Systems with a Hierarchical Organization of Tasks", *Proceedings of the 12th International Conference on Distributed Computer Systems*, June 1992.

[4] S. Chiang, R. K. Mansharamani and M. K. Vernon, "Use of Application Characteristics and Limited Pre-emption for Run-To-Completion Parallel Processor Scheduling Policies", *Proceedings of the ACM SIGMETRICS Conference*, 1994, pp. 33-44.

[5] S. P. Dandamudi, "A Comparison of Task Scheduling Strategies for Multiprocessor Systems", *IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, Dec. 1991, pp 423-426.

[6] S. P. Dandamudi and S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems", *IEEE Trans. Parallel and Dist. Syst.*, Vol. 6, Jan. 1995, pp. 1-16.

[7] L. Dowdy, “On the Partitioning of Multiprocessor Systems”, Technical Report, Vanderbilt University, July 1988.

[8] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park, “Dynamic Partitioning in a Transputer Environment”, *Proceedings of the ACM SIGMETRICS,*

[9] D. G. Feitelson and L. Rudolph, “Distributed Hierarchical Control for Parallel Processing”, *IEEE Computer*, 23(5), May 1990, pp. 65-77.

[10] D. Ghosal, G. Serazzi and S. K. Tripathi, “The Processor Working Set and Its Use in Scheduling Multiprocessor Systems”, *IEEE Transactions on Software Engineering,* Vol. 17, No. 5, May 1991.

[11] A. Gupta, A. Tucker and S. Urushibara, “The Impact of Operating System Policies and Synchronization Methods on the Performance of Parallel Applications”, *Proceedings of the ACM SIGMETRICS Conference*, May 1991, pp. 120-132.

[12] S. T. Leutenegger and M. K. Vernon,“The Performance of Multiprogrammed Multiprocessing Scheduling Policies”, *Proceedings of the ACM SIGMETRICS Conference,* May 1990, pp. 226-236.

[13] S. Majumdar, D. L. Eager and R. B. Bunt, “Scheduling in Multiprogrammed Parallel Systems”, *Proceedings of the ACM SIGMETRICS Conference,* May 1988, pp.104-113.

[14] C. McCann, R. Vaswani and J. Zahorjan, “A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors”, *ACM Transactions on Computer Systems,* Vol. 11, No. 2, May 1993, pp. 146-178.

[15] C. McCann and J. Zahorjan, “Processor Allocation Policies for Message-Passing Parallel Computers”, *Proceedings of the ACM SIGMETRICS Conference,* May 1994, pp. 19-32.

[16] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, University of Rochester, Computer Science Technical Report 342, Jan. 1991.

[17] J. K. Ousterhout, “Scheduling techniques for Concurrent Systems”, *Proceedings of the 3rd International Conference on Distributed Computing Systems,* 22-30, Oct. 1982, pp. 22-30.

[18] E. Rosti, E. Smirni, L. W. Dowdy, G. Serrazi, and B. M. Carlson, “Robust Partitioning Policies of Multiprocessor Systems”, *Performance Evaluation 19,* 1994, pp. 141-165.

[19] S. K. Setia, M. S. Squillante, and S. K. Tripathi, “Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Computers”, *Proceedings of the ACM SIGMETRICS Conference,* May 1993, pp. 158-170.

[20] S. K. Setia, M. S. Squillante, and S. K. Tripathi, “Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems”, *IEEE Transactions on Parallel and Distributed Systems,* Vol. 5, No. 4, Apr. 1994, pp. 401-420.

[21] S. K. Setia, and S. K. Tripathi, “An Analysis of Several Processor Partitioning Policies or Parallel Computers”, Technical Report CSTR-2684,University of Maryland, May 1991.

[22] K. C. Sevcik, “Characterizations of Parallelism in Applications and their Use in Scheduling”, *Proc. ACM SIGMETRICS Conf.*, 1989, pp. 171-180.

[23] K. C. Sevcik, “Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems”, *Performance Evaluation 19*, 1994, pp. 107-140.

[24] K. C. Sevcik and S. Zhou, “Performance Benefits and Limitations of Large NUMA Multiprocessors”, *Performance Evaluation 20,* 1994, pp. 185-205.

[25] M. S. Squillante, and R. D. Nelson, “Analysis of Task Migration in Shared Memory Multiprocessor Scheduling”, *Proceedings of the ACM SIGMETRICS Conference,* May 1991, pp. 143-155.

[26] D. Towsley, C. G. Rommel, and J.A. Stankovic, “Analysis of Fork-Join Program Response Times on Multiprocessors”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, Jul. 1990, pp. 286-303.

[27] A. Tucker and A. Gupta, “Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors”, *Proc. ACM Symp. Operating System Principles*, Dec. 1989, pp. 159-166.

[28] R. Vaswani, and J. Zahorjan, “The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared-Memory Multiprocessors”, *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, ACM, New York, May 1990.

[29] J. Zahorjan and C. McCann, “Processor Scheduling in Shared Memory Multiprocessors”, *Proc. ACM SIGMETRICS Conf.*, May 1990, pp. 214-225.

[30] J. Zahorjan, E. D. Lazowska, and D. L. Eager, “Spinning versus Blocking in Parallel Systems with Uncertainty”, *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, Kyoto, Japan, Dec. 1988.

[31] J. Zahorjan, E. D. Lazowska, and D. L. Eager, “The Effect of Scheduling Discipline on Spin Overhead in Shared-Memory Parallel Systems”, *IEEE Transactions on Parallel and Distributed Systems*, 2, 2, Apr. 1991, pp. 180-198.

[32] S. Zhou and T. Brecht, “Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors”, *Proc. ACM SIGMETRICS Conf.*, May 1991, pp. 133-142.