Robust Partitioning Policies of Multiprocessor Systems

E. Rosti*, E. Smirni†, L.W. Dowdy†, G. Serazzi‡, B.M. Carlson§

Abstract

Various techniques for multiprogramming parallel multiprocessor systems have been proposed recently as a way to improve performance. A natural approach is to divide the set of processing elements into independent partitions, and simultaneously execute a different parallel program in each partition. Several issues arise, including the determination of the optimal number of programs allowed to execute simultaneously (*i.e.*, the number of partitions) and the corresponding partition sizes. This can be done statically, dynamically, or adaptively, depending on the system and workload characteristics.

In this paper several adaptive partitioning policies are evaluated. Their behavior, as well as the behavior of static policies, is investigated using real parallel programs. The policy applicability to actual systems is addressed, and implementation results of the proposed policies on an iPSC/2 hypercube system are reported. The concept of *robustness* (*i.e.*, the ability to perform well on a wide range of workload types over a wide range of arrival rates) is presented and quantified. Relative rankings of the policies are obtained, depending on the specific workload characteristics. A trade-off is shown between potential performance and the amount of knowledge of the workload characteristics required to select the best policy. A policy that performs best when such knowledge of workload parallelism and/or arrival rate is not available is proposed as the most robust of those analyzed.

1 Introduction

General purpose multiprocessor systems consisting of considerable numbers of processing elements, have recently become commercially available. Multiprogramming parallel machines has been recently proposed as a feasible way for improving their performance. It can be achieved by time sharing the multiprocessor among several applications in the traditional way used in uniprocessor systems [Ous82], [MEB88], [LV90]. Since parallel applications may not be able to efficiently utilize all the processors in the system, it may be better to partition the processor set among the programs in a space sharing fashion. With space sharing, subsets of processors are assigned to the various

*Dipartimento di Scienze dell'Informazione - Università di Milano, Italy (while on leave at Vanderbilt University)

†Department of Computer Science - Vanderbilt University - Tennessee, USA

‡Dipartimento di Elettronica e Informazione - Politecnico di Milano - Italy

§Computer Systems Research Institute - University of Toronto - Canada

This work was partially supported by Italian CNR "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo – Sottoprogetto 3" by grant N. 92.01615.PF69.115.23757, and under sub-contracts 19X-SH008V and 19X-SL131V administered through the Mathematical Sciences Section of Oak Ridge National Laboratory.

parallel applications, either in a dedicated mode (*i.e.*, one subset per program) [LDP89], or in time sharing mode (*i.e.*, several programs execute on the same partition) [ZB91]. How many processors to assign, when to compute the partition size, and whether to allow partition sizes to vary during the program execution are the issues addressed by *processor partitioning* or *allocation* strategies [Sev89], [TG89], [PD89], [DCDP90], [ZM90], [GGT91], [DL93]. Processor allocation is also known as *processor scheduling* [GST91]. The *task assignment* and *task scheduling* problems address the issues of which application tasks, considered as the smallest unit of allocation, to assign to which processor, and how to schedule the tasks within each processor. Several authors address the task scheduling problem in multiprocessor systems [Ous82], [LV90], [ZM90], [GTU91], [ZB91].

Static, *adaptive*, and *dynamic* processor allocation schemes are possible.

- Static strategies [Sev89], [MEB91], [GST91] assume that the number of processors allocated per job is a system configuration parameter which does not change. Static policies are simple and efficient to implement, but at a performance cost, since they do not adapt to various workloads. They work well when the workload is known and constant.

- Adaptive strategies [GGT91] adjust to various workloads by computing a job's partition size when it is scheduled. The partition size does not change until a job completes execution. These types of policies are referred elsewhere as static ones [Sev89], [ZM90], [MEB91], [ST91], [GST91]. The number of simultaneously executing jobs varies according to the specific strategy, the workload arrival patterns, and the service demands. The overhead is minimal, since the partition size computation is simple. Adjusting the partition size depending upon the current state of the processor set and the number of jobs requesting service increases the system's flexibility.

- Dynamic schemes allow the number of processors assigned to a job to vary during its execution [PD89], [TG89], [DCDP90], [ZM90], [LV90]. Processors can be taken away from an executing program and reassigned to a newly arriving program. Similarly when a job finishes, its freed processors can be distributed to the other jobs still running. Depending on the underlying operating system platform, the overhead for reallocating processors to processes during execution may be considerable [DCDP90] or not [TG89]. Furthermore, semi-dynamic policies are similar to dynamic ones but the processor repartitioning is restricted by several factors [MEB91].

In this paper various adaptive processor partitioning policies for a space sharing multiprocessor system are proposed and compared against known static policies. The policies are examined in a single class environment. Extensions to multi-class workloads are also presented. Jobs execute on dedicated partitions. The adaptive policies are characterized by their adaptability to the possible range of arrival rates and to the possible range of workload types. Various workloads with different parallel characteristics are analyzed. Each workload is characterized by its speedup curve which is derived from measured execution times for various numbers of processors of a real workload. The proposed adaptive policies do not assume any specific knowledge of workload characteristics that have been proved particularly useful in other scheduling schemes [EZL89], [Sev89], [GST91]. The performance metric used to evaluate the various policies is *power*, defined as the ratio of system throughput to response time [Klei79]. The concept of *robustness* is used to compare policies. Robustness is the ability of a policy to perform well on a wide range of workload types and a wide range of system loads. A robustness metric is associated with each policy by means of a scoring function. Relative rankings are obtained under various workload assumptions. An indication of which policy to use can be given based on the relative rankings, without *a priori* knowledge of the workload characteristics and/or the load the system operates on. This does not guarantee peak performance in any specific case. However, it avoids the system performing poorly under any arrival rate or workload type. Several of the policies have been implemented on an Intel iPSC/2 system. Experimental results are presented.

No specific assumptions are made with respect to the underlying hardware platform. In shared memory environments, dynamic schemes may be possible since the overhead cost of moving threads among processors during execution is low. However, the various policies are particularly appropriate for implementation in distributed memory systems.

This paper is organized as follows. A description of the policies is presented in Section 2. Section 3 gives an analysis of each policy. Section 4 presents analytical results to obtain the optimal partition size given the arrival rate and the parallel characteristics of the workload. A comparison of the performance of the policies is described in Section 5. Section 6 generalizes the analysis to multi-class workloads and to various distributions of the interarrival and job service times. Experimental results from implementing the policies on the iPSC/2 are reported in Section 7. Future work and conclusions are summarized in Section 8.

2 Policy Descriptions

In this section, several processor allocation policies are presented. It is assumed that parallel programs (*i.e.*, jobs) requesting service at the multiprocessor system are served in *first-come first-served* order. The processor allocator is activated every time a job arrival or completion occurs. A job is scheduled (*i.e.*, actually allocated processors) when there is a "sufficient" number of free processors available. "Sufficient" is defined according to the specific allocation policy. If the number of free processors is not sufficient, the job waits. In the pseudo-code description reported for each policy, the procedure *schedule_a_job* implements the processor allocation to the job, and initiates job execution. The constant TOT_PEs represents the total number of processing elements (*i.e.*, processors) in the system. The variable PARTITION_SIZE is the number of processors a job is assigned according to the policy. The specific descriptions of the policies follow.

2.1 Fixed Processors Per Job (PPJ)

This policy has been previously analyzed and is included here for comparison purposes [Sev89], [GST91], [ST91]. At configuration time the processor set is divided into a fixed number of equal sized partitions. When a job arrives, it waits for the next available free partition and then executes in that partition. The maximum number of simultaneously executing jobs is equal to the number of partitions, which is equal to the total number of processors divided by the partition size.

```
/* given: PARTITION_SIZE */
while (queue_length > 0) ∧ (free_partitions > 0) do
    free_partitions ← free_partitions −1
    queue_length ← queue_length −1
    schedule_a_job(PARTITION_SIZE)
od
```

Algorithm 1: Fixed Processors Per Job Policy.

The value of PARTITION_SIZE determines the specific version of the PPJ policy. When PARTITION_SIZE = TOT_PEs, the system is used in single user mode, and the jobs are scheduled sequentially. When PARTITION_SIZE = 1, as many jobs as the total number of processors may execute concurrently, but each executes on a single processor. Processors are left idle if the total number of jobs in the system is less than the number of partitions. The PPJ policy is static, since PARTITION_SIZE remains fixed throughout the lifetime of the system. It does not adapt to the load dynamically or to the specific parallel characteristics of the workload, but it does achieve good performance under particular conditions (see Section 3.3).

2.2 Equal Partitioning with a Maximum (EPM)

An improvement in flexibility is obtained if the partition size is computed at allocation time instead of at system configuration time. In the equal partitioning with a maximum (EPM) policy, the set of currently free processors is equally divided among the jobs in the waiting queue. To limit the maximum size of any allocated partition, an upper bound on the partition size is introduced as a configuration parameter (MAX in Algorithm 2). For example, if MAX = TOT_PEs/2, then no single job is allowed to execute on more than half of the system processors, regardless of the number of free processors and the number of waiting jobs. This allows a mechanism to restrict any one job from monopolizing the system by reserving some of the free processors for anticipated later arriving jobs.

```
/* given: MAX */
while (queue_length > 0) ∧ (free_processors > 0) do
    temp ← max(1,⌊free_processors/queue_length⌋)
    partition_size ← min(MAX,temp)
    free_processors ← free_processors − partition_size
    queue_length ← queue_length −1
    schedule_a_job(partition_size)
od
```

Algorithm 2: Equal Partitioning with a Maximum policy.

The proposed EPM policy is similar to the ASP policy presented in [ST91]. The difference is that the value of MAX in [ST91] is the workload's maximum parallelism. The maximum parallelism is defined as the maximum number of simultaneous busy processors during the execution of a program when a sufficiently large amount of processors is available [EZL89]. Here, since the workload's maximum parallelism may not be known, the value of MAX is set at system configuration time and may be assigned a value between 1 and TOT_PEs. It identifies a specific version of the policy. When MAX = 1, the policy is identical to PPJ1 (*i.e.*, PPJ with PARTITION_SIZE = 1). When MAX = TOT_PEs, no upper bound applies to the partition size.

2.3 Insurance Policies (IP)

The insurance policy generalizes the EPM policy. The motivation for MAX in the EPM policy is to prohibit any one job from monopolizing the system and possibly reserve some of the free processors for anticipated future job arrivals. However, the value of MAX is set at system configuration time. The motivation for the insurance policy is to always reserve a percentage of the available free processors for later job arrivals depending on the system load in a more flexible way. The specific number of reserved processors is a function of the insurance percentage PERCENT, and of the current system state (*i.e.*, the number of free processors and the number of jobs currently executing). PERCENT is a configuration parameter. In the insurance policy specified by Algorithm 3, at each allocation time, the number of processors to be allocated is computed and equally divided among the waiting jobs in the queue. Those processors not allocated constitute a system "*insurance*" against future arrival bursts. The value of PERCENT determines the specific version of the policy.

```
/* given: PERCENT */
alloc_proc ← ⌈free_processors*(1-PERCENT^queue_length)⌉
while (queue_length > 0) ∧ (free_processors > 0) do
    partition_size ← max(1,⌊alloc_proc/queue_length⌋)
    free_processors ← free_processors − partition_size
    queue_length ← queue_length − 1
    schedule_a_job(partition_size)
od
```

Algorithm 3: Insurance Policy.

As an illustration, suppose PERCENT is chosen to be 1/2. If 16 processors are available and a new job arrives (with no other jobs in the queue), 8 processors will be allocated to the job. If 8 processors become available with 2 jobs in the queue, the number of processors to be allocated is computed as $8 * (1 - (1/2)^2) = 6$. These 6 processors are then equally allocated among the 2 jobs and the remaining 2 processors are held as insurance for future job arrivals. When the total number of free processors is less than or equal to the queue length, each job is assigned one processor, as long as processors are available.

2.4 Adaptive Policies (AP)

Although more flexible than the PPJ policy, both the EPM and insurance policies have potential performance problems. For example, it is possible that both very small and very large partitions are allowed to coexist. This "fragmentation effect" may seriously hurt performance. Suppose that a system has N processors and consider either: a) EPM with MAX = N − 1; or b) the insurance policy with PERCENT = $\frac{1}{N}$. Under a low job arrival rate a frequent scenario would be: 1) a job arrives to an empty system and is allocated N − 1 processors; 2) during execution, a second job arrives and is allocated the remaining 1 processor; 3) the first job completes execution quickly because of its N − 1 processors, but the second job (with only one processor) remains in execution for a long time period while the remaining N − 1 processors are idle. This results in poor performance. Other adaptive policies, based upon current state information (*e.g.*, number of free processors, number of busy processors, number of jobs in service, number of jobs in queue), are possible which seek to avoid potential fragmentation problems. Several such adaptive policies, referred to as AP1, AP2, AP3, AP4, and AP5 are presented in this section. They attempt to generate equal sized partitions while adapting to transient workload behavior.

At any point in time, the number of equal size partitions that the system should be divided into would ideally be equal to the total number of jobs in the system (both in service and in the queue). However, this ideal is not realizable because of the system dynamics. Jobs arrive and depart and cannot change their partition sizes during the execution. Using the partition size to represent the system state, the ideal *target* states for a 16 node system are depicted in Figure 1. If there is only one job in the system, the ideal is to allocate all the processors to it. If two jobs are in the system, it would be best for each to have 8 processors. With three jobs, it would be best for each to have 5 (or 6) processors, etc. Other states are not shown in Figure 1 since they do not represent the target states of the policy design.

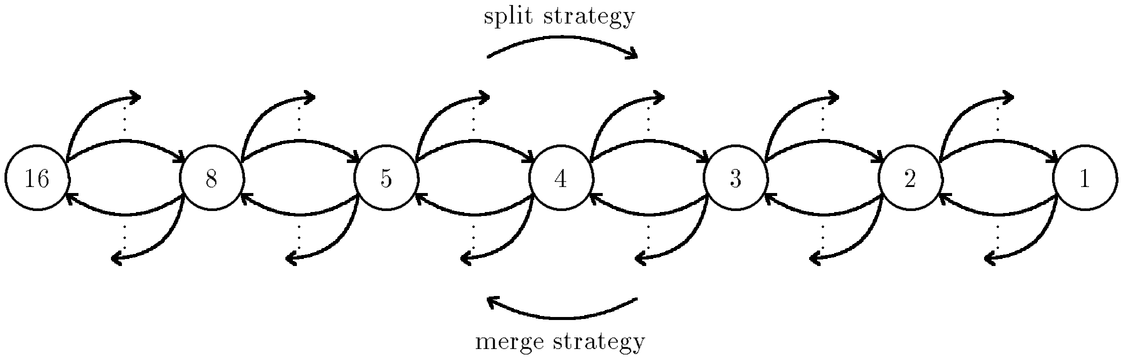

Figure 1: Target state diagram illustrating the adaptive policies.

Recalling that the states in Figure 1 represent policy targets, state transitions are determined by a "split" and a "merge" strategy. Suppose that the system allocation policy is currently in state i (*i.e.*, the partition size is i). When jobs arrive and depart such that all partitions are busy, and there is a steady queue of as many waiting jobs as partitions, each job will be scheduled on a partition of size i. If the queue length surpasses the number of partitions, it may be beneficial for the policy to split by going to some new state j, $j < i$. Similarly, if several jobs depart the system without arrivals, leaving idle partitions, it may be beneficial for the policy to merge by going to some state k, $k > i$, so that the next arriving job will be allocated more (*i.e.*, k) processors. No policy transition occurs when the queue length equals the number of partitions in the current state. The specific split and merge strategies determine different versions of the basic adaptive policy.

The basic version of this policy is shown in Algorithm 4. The system reacts to transient fluctuations by re-defining the target partition size upon job arrival and job departure, based upon the split and merge strategies. Recall that the allocator is called whenever a job arrives and there are free processors, or whenever a job departs and there are queued jobs.

/* given: the specific implementation of *compute_target_size*
 *   based upon the split and merge strategies */
target_size $\leftarrow$ *compute_target_size*
while (queue_length > 0) $\wedge$ (free_processors $>$ target_size) **do**
  free_processors $\leftarrow$ free_processors $-$ target_size
  queue_length $\leftarrow$ queue_length $- 1$
  schedule_a_job(target_size)
od

Algorithm 4: The basic Adaptive Policy.

Depending upon the specification of the procedure *compute_target_size*, which implements specific split and merge strategies, various adaptive policies are possible. Five adaptive policies, AP1, AP2, AP3, AP4, and AP5, are presented.

The *compute_target_size* specification for AP1 is

$$\text{target\_size} \leftarrow \max(1, \lfloor \frac{TOT\_PEs}{queue\_length} + 0.5 \rfloor)$$

In AP1, the split and merge strategies are directly dependent upon the current value of queue_length. Whenever the queue length increases, the target size decreases (*i.e.*, the split strategy) and whenever the queue length decreases, the target size increases (*i.e.*, the merge strategy).

As shown in the next section, AP1 tends to overreact to changes in the queue length. In the AP2 version, the effect on the target_size by the queue length is reduced. The *compute_target_size* specification for AP2 is

$$\text{target\_size} \leftarrow \max(1, \lfloor \frac{TOT\_PEs}{queue\_length+1} + 0.5 \rfloor)$$

The split strategy of AP2 (*i.e.*, when the queue length increases) tends toward premature fragmentation. For example, when a single job arrives at an empty system, only 50% of the processors are allocated. The merge strategy of AP2 (*i.e.*, when the queue length decreases) is more conservative than AP1.

In AP3, the split strategy of AP1 is combined with a quite conservative merge strategy that allows target_size to increase by only one state at a time. Specifically,

if TOT_PEs $\leq$ target_size$*$queue_length **then**
    target_size $\leftarrow \max(1, \lfloor \frac{TOT\_PEs}{queue\_length} + 0.5 \rfloor)$ /* split */
else
    new_nb_of_partition $\leftarrow \lfloor \frac{TOT\_PEs}{target\_size} - 1 \rfloor$
    target_size $\leftarrow \lfloor \frac{TOT\_PEs}{new\_nb\_of\_partition} \rfloor$ /* merge */

Slight modifications are made to AP3 in constructing two new policies, AP4 and AP5. The merge strategies in AP4 and AP5 are identical and are more conservative than AP3. The target_size is increased only when there are at least two free partitions of the previous target_size. AP5 adds the constraint that at least two processors are allocated to each job. In summary, the various adaptive policies are distinguished by their split and merge strategies. These distinctions lead to the observed differences in performance. The split strategy governs the rate at which the system adapts to a sudden arrival bursts. All the policies use the same split strategy except AP2. When a sudden arrival burst occurs, the system immediately adapts by targeting an equal number of processors to each job. The AP2 policy throttles this target by being slightly more conservative in its partition size. Similarly, the various policies vary according to their merge strategies. Policies AP1 and AP2 adapt immediately to long interarrival intervals, where several jobs depart before a new arrival occurs. In this event, the next arrival will likely be assigned more processors than would be assigned using AP3, AP4, and AP5. These latter three policies are more conservative in their merge strategies. If a long interarrival gap occurs, they are more likely to reserve processors for another anticipated burst of arrivals (*i.e.*, similar to the arrival burst that forced the system into its current target state). Thus, the key to an effective adaptive policy is a proper balance between split and merge strategies.

As an illustration, consider 4 jobs executing in partitions of 4 processors each. After a job finishes, 4 processors become available. If there are 6 jobs in the queue, the split strategy for all policies except AP2 sets the target size equal to 3 (AP2 sets the target to 2). If there is only one job in the queue, merging occurs and AP1 sets the target equal to 16, AP2 sets it equal to 8. AP3, being more conservative, sets the target to 5, while AP4 and AP5 do not change the target since no two partitions of the previous target are free.

3 Policy Analysis

In this section the behavior of each policy is investigated over the entire range of arrival rates for various workload types. A simulation model of a multiprocessor system was constructed using QNAP2 [Sim89]. Due to the adaptive discipline of several of the proposed scheduling policies, analytic analysis of all policies is difficult. Although the PPJ policies can be modeled analytically, they are analyzed by simulation for consistency purposes with the adaptive policies. The interarrival time between successive jobs is assumed to be exponentially distributed. Arriving jobs wait until scheduled. With static and adaptive policies, the overhead for processor scheduling is assumed to be negligible. This assumption was validated via experimentation (described in Section 7) on the Intel iPSC/2. Once scheduled on a particular partition size, a job runs to completion. The job service time distribution is exponential. It is a function of the number of allocated processors and of the speedup characteristics of the workload type, from which the mean of the service time distribution is derived. After completing service at the multiprocessor, jobs exit the system. A system of 16 processing elements is considered. All simulations were run at the 95% confidence level.

This paper focuses on partitioning policies. A detailed knowledge of the internal structure of each job is not necessary. Architectural features such as shared rather than distributed memory are not explicitly modeled.

For each policy, the performance curves are reported in Sections 3.3–3.6. Each considered workload type has different characteristics. Thus, the maximum feasible arrival rate depends on the saturation point. The saturation arrival rate, λ_s, occurs at $p * \mu(1)$, where p is the total number of processors and $\mu(1)$ is the execution rate of the job when allocated a single processor. For sake of comparison, the arrival rates are normalized over $[0, \lambda_s]$ (*i.e.*, from 0% system load to 100% system load). A description of the various workload characteristics and of the behavior of the policies follows.

3.1 The Workload

In order to obtain a better understanding of the policy sensitivity to the parallel workload characteristics, a single class environment is initially assumed. An extension to multi-class workload and various distributions of job interarrival and service times is presented in Section 6. The workload is characterized by each program's speedup, (*i.e.*, $S(p) = \frac{T(1)}{T(p)}$ where $T(1)$ is the execution time of an application program when allocated one processor, and $T(p)$ is the execution time when allocated p processors). Four diverse speedup functions (see Figure 2) are used to represent the behavior typical of parallel applications.

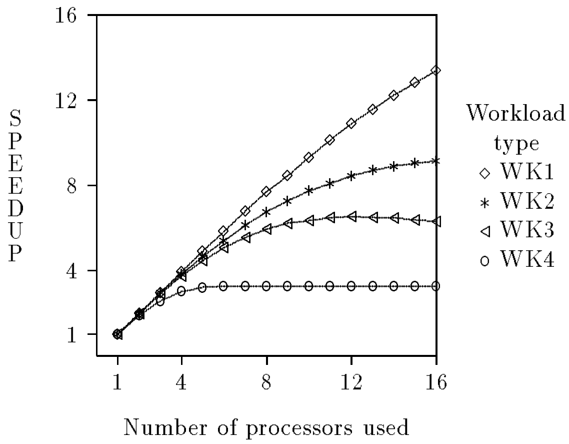

Figure 2: Observed speedup curves for transposition sort application.

These speedup functions are derived from measurements on an actual multiprocessor system. A parallel implementation of the *odd-even transposition sort* was executed on a distributed memory machine, a MEIKO40 with 16 T800 transputers. The application was executed with different numbers of processors, varying from 1 to 16, and the execution times were measured. Different speedup characteristics result from different input string lengths. A reduction in speedup occurs when the input size is decreased, due to the relative increase of communication time compared to computation time. As more processors are allocated, the potential reduction in computation time is offset by increased communication time. String lengths of 8K, 2K, 1K and .25K items yield WK1, WK2, WK3, and WK4, respectively.

- WK1 is a workload exhibiting nearly linear speedup; its maximum speedup occurs when $p = 16$, the maximum number of processors.
- WK2 is a workload exhibiting increasing concave speedup; its maximum occurs at $p = 16$.
- WK3 also exhibits concave speedup, but with a maximum speedup when $p = 10$, when $p > 10$ the speedup slowly decreases.
- WK4 is a workload with limited parallelism potential. The speedup function increases to its maximum at $p = 6$ and remains flat thereafter.

A related workload parameter is the *processor working set* [GST91], or PWS. The PWS of a workload is the partition size that, in a single class system, maximizes $\frac{S(p)^2}{p}$. The PWS for each workload is equal to 16, 10, 8 and 4, respectively.

3.2 Objective Function

In order to compare the various allocation policies uniformly across various workloads, an appropriate objective function is needed. The objective function selected is the power of the system, Φ, defined as the ratio of throughput to response time [Klei79]. For an open system and a given system load, that is for a given job arrival rate, maximizing power is the same as minimizing response time. Here, power is preferred to response time, because it captures the trade-off between system and application performance. Furthermore, power is minimized under very light load, since throughput is low, as well as under very high load, since response time tends to infinity. The optimal point is easier to identify than for response time, which requires the analysis of the "knees" of the curve [Sev89].

3.3 Fixed Processors Per Job

Fixed, equal sized partitions of sizes 1, 2, 4, 8 and 16 processors each are considered. The comparisons across various workloads are shown in Figure 3. As expected [Sev89], with workload WK1, PPJ16 performs best under light to medium load. This results from the nearly linear speedup characteristics of this workload. However, as the system load tends towards 100%, first PPJ8, then PPJ4, PPJ2, and finally PPJ1 perform best. This trend is characteristic across all workloads. For example, for WK2, which exhibits good speedup, PPJ16 is the best policy at low load, under 20% of system capacity. From 20% to 60%, the best policy is PPJ8, and for capacity values greater than 60%, PPJ4 outperforms the others. Only at near 100% capacity PPJ1 is the best policy. With WK3 and WK4, since their speedup characteristics are poor, it is not advantageous to assign large partition sizes.

The evident disadvantage of PPJ policies is the lack of flexibility and adaptability. Usually, it is not possible to know *a priori* the speedup characteristics and the system load. For example, PPJ8 is good at 50% load level under WK2. However, if the load were 80% or if the workload were WK4, PPJ8 would perform poorly. Thus the PPJ policies do not adapt well to changes in the load or changes in the workload type. Ideally, the policy should be able to switch dynamically to a different partition size depending on the load conditions and the workload type.

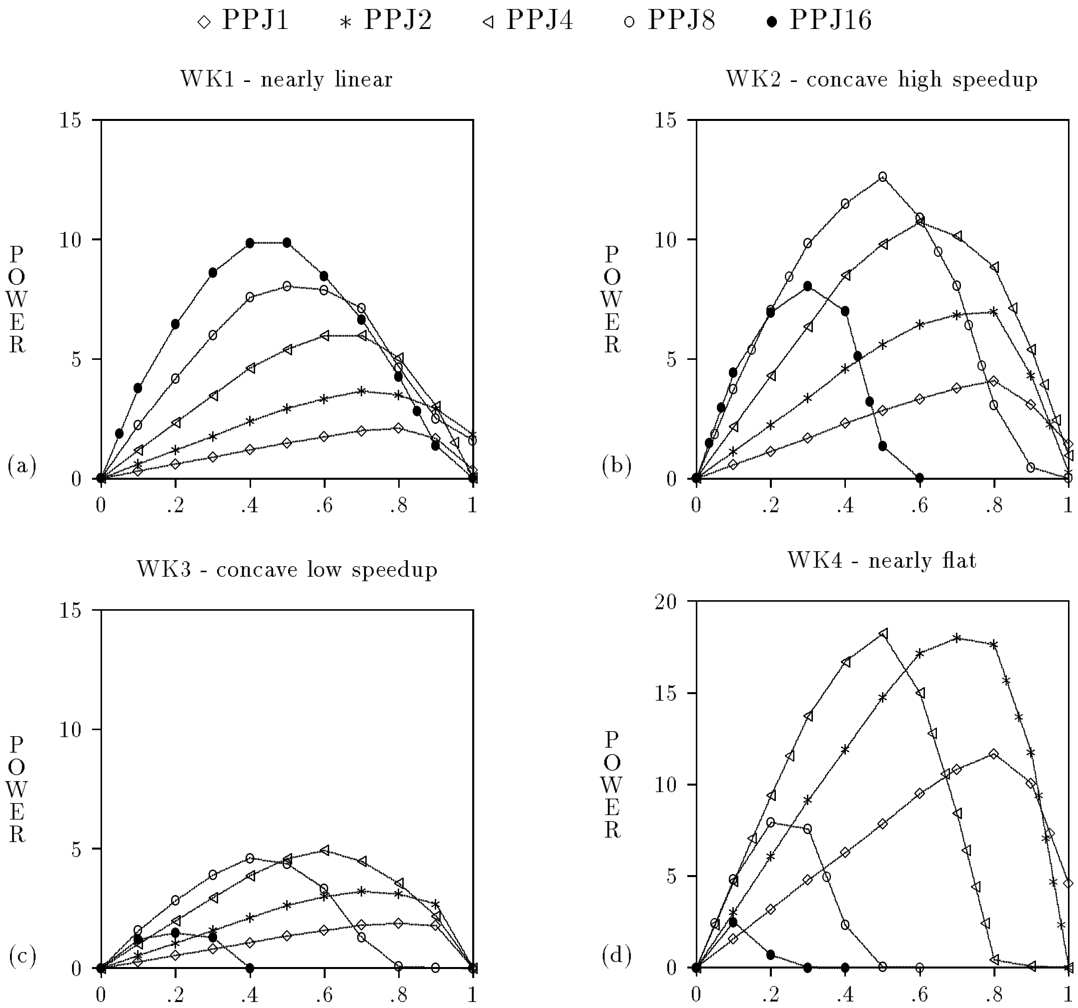

Figure 3: Power curves *vs* system load for the various workloads for the PPJ policies.

The maximum value of the power attained for each workload depends on its PWS and the partition size, as shown in Section 4. The PPJ policy that achieves the maximum power for a given workload is the one whose partition size is closest to the workload's PWS.

3.4 Equal Partitioning with a Maximum

The comparison results of five variants (where MAX = 1, 4, 8, 12 and 16) of the EPM policy are shown in Figure 4. Other variants, whose results are not reported here, were investigated to better understand the observed behavior. For example, a MAX value of 14 causes quite poor performance, due to fragmentation effects. It is possible for 14 processors to be allocated to one job and the remaining 2 to another job. When the job with 14 processors completes, the system becomes very unbalanced. Only the more balanced partition schemes are presented in Figure 4.

The most noticeable improvement of the EPM policies over the PPJ policies is that the EPM policies do not saturate before the system load reaches 100% (see Figure 4). Thus, the EPM policies are more robust and are effective over the full range of possible arrival rates. However, for a given load, the EPM policies do not achieve power values as high as the best PPJ policy.

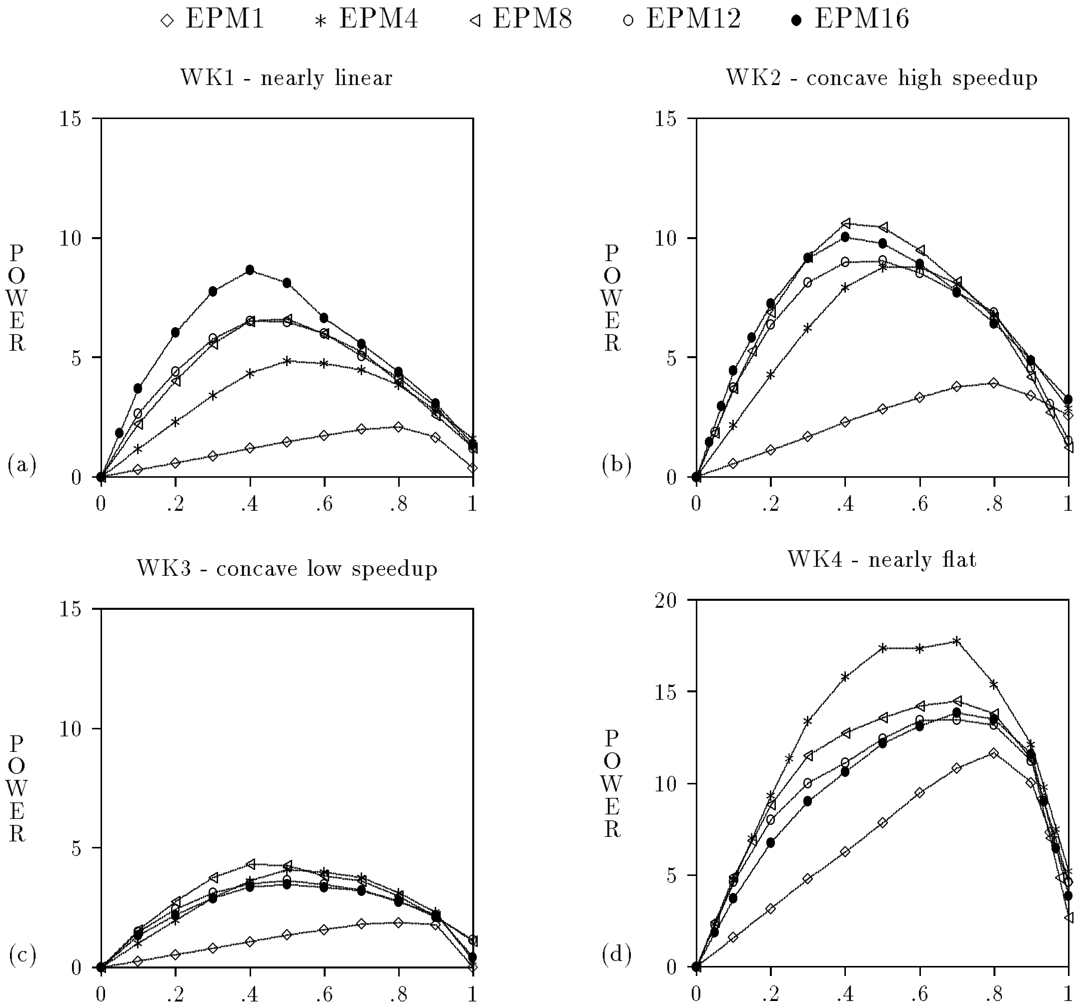

Figure 4: Power curves *vs* system load for the various workloads for the EPM policies.

Thus, there is a trade-off between robustness and absolute maximum power. As the load increases, the different versions converge to the same PPJ1 behavior (*i.e.*, MAX = 1). The best EPM policy depends upon the workload characteristics. For example, EPM16 is best for workloads with near linear speedup (*e.g.*, WK1) while EPM4 is better for workloads with limited speedup (*e.g.*, WK4). EPM8 is well suited for WK2 and WK3. A good heuristic is to set MAX to the PWS of the workload. Another good heuristic is to set MAX to the maximum parallelism [ST91]. The primary disadvantage of the EPM policies is that the parameter MAX must be properly tuned to the workload. The primary advantage that EPM policies have over PPJ policies is their adaptability and robustness. A PPJ policy is characterized by exhibiting very good performance over a relatively narrow range of load levels and workload types. An EPM policy is characterized by exhibiting adequate performance over a relatively wide range of load levels.

3.5 Insurance Policies

Performance comparisons of five versions (*i.e.*, with PERCENT = 0, .10, .25, .50, .75) of the insurance policy are shown in Figure 5. As with the EPM policies, the insurance policies are robust with respect to the capacity level. Comparing Figures 4 and 5, the performance of the insurance policies is inferior to EPM policies.

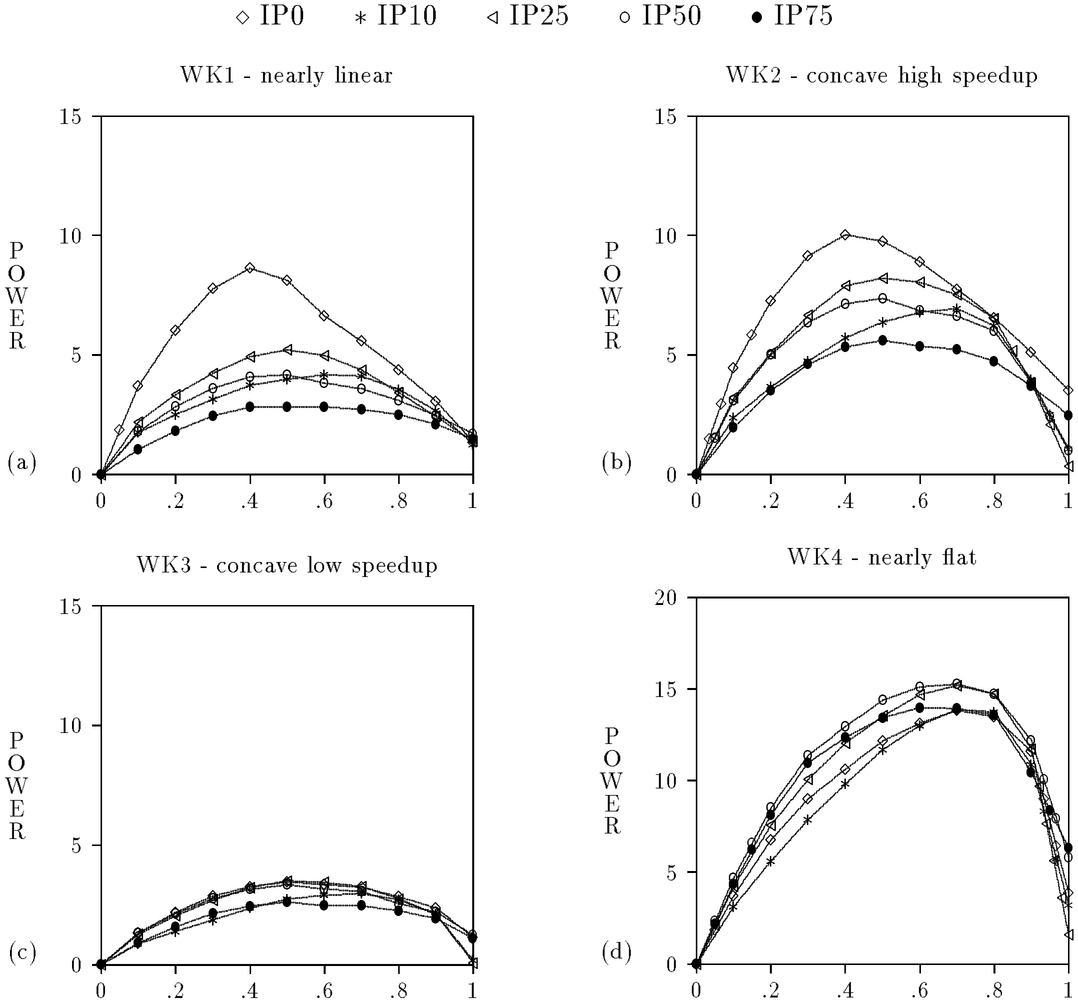

Figure 5: Power curves *vs* system load for the various workload for the insurance policies.

Only EPM1 achieves lower power than the several insurance policies. For example, for WK1 and WK2, the best insurance policy is when PERCENT is 0% (*i.e.*, no insurance). This 0% policy is identical to the EPM policy with MAX = 16. The results indicate that saving processors for future arrivals is effective only when the application's speedup is low. As the load level increases, the relative performance between the five versions remains fixed, (*i.e.*, the curves have few crossover points). However, the best version depends upon the specific workload, as with the EPM policy.

3.6 Adaptive Policies

The performance of the adaptive policies is shown in Figure 6. All five variants of the basic adaptive policy perform well, within a certain range of power, over the entire system load and all workload types.

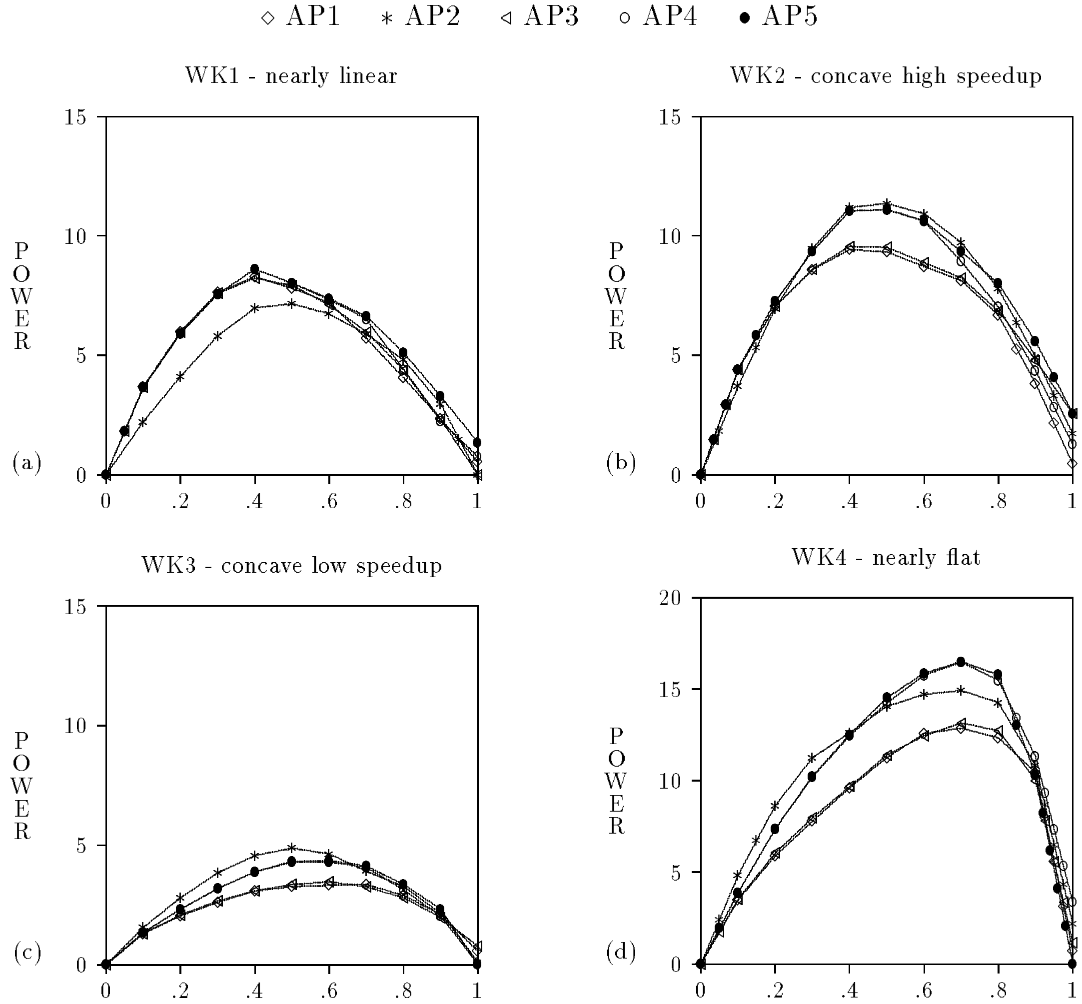

Figure 6: Power curves *vs* system load for the various workloads for the adaptive policies.

The absolute values of power are close to the best power values achieved by the PPJ policies. With respect to the various workloads, the policies tend to group into two classes of behavior. Policies AP2, AP4, and AP5 exhibit consistently better performance than AP1 and AP3 over the various capacities and various workload types. Merge strategies that are more sensitive to interarrival gaps determine the inferior performance of AP1 and AP3 compared to the other adaptive policies. The adaptive policies compare favorably with EPM and IP policies. The important advantage of the AP family over the other policies is that they are parameter independent and they consistently achieve among the highest power values of the examined policies.

4 Theoretical Motivation

In this section, an analytical expression is given that provides justification for the empirical observation that as the job arrival rate increases, the optimal equipartitioning policy should reduce the partition size. The assumption that the workload is single class implies that all jobs should be treated equally. This suggests an equipartitioning policy, namely the PPJ policy. This observation applies also in the context of dynamic policies [LV90].

A system using the PPJ policy can be modeled as a simple $M/M/x$ queue, where x is the number of partitions. For instance, if the total number of processors is 16, the PPJ8 policy can be modeled as an $M/M/2$ queue. In general, for an $M/M/x$ queue the power Φ is given by

$$\Phi = \frac{X}{R} = \frac{\lambda^2}{\frac{\lambda}{\mu_y} * p_0 * \sum_{i=0}^{x-1} \frac{1}{i!}(\frac{\lambda}{\mu_y})^i + p_0 * \sum_{i=x+1}^{\infty} \frac{i}{x^{i-x} * x!} * (\frac{\lambda}{\mu_y})^i} \qquad (1)$$

where X represents throughput, R is system response time, λ is the arrival rate, x is the number of partitions, p_0 is the probability that the system is idle, and μ_y is the execution rate of a job if y processors are assigned to it. By definition, $y = total\_number\_of\_processors/x$. Similarly, if the service time distribution is deterministic, simple $M/D/x$ analysis is possible.

As the job arrival rate changes, the target PPJ policy should also change, since the performance of any single PPJ policy is good only in a limited subrange of arrival rate λ (*i.e.*, system load). This is illustrated in Figure 7(a). A better policy would guarantee the assignment of an equal number of processors to each job in the system, with the partition size adapting to the arrival rate.

If the speedup curve of the workload is known (*i.e.*, the μ_y's), the value of λ that maximizes Φ can be derived for all values of x. For instance, by analyzing (1) when $x = 1$, Φ is maximized when: $\lambda = \mu_y/2$. Likewise, if $x = 2$, Φ is maximized when: $\lambda = \frac{2}{\sqrt{3}}\mu_y$. As x increases, so does the arrival rate λ that maximizes power. For higher values of x, the expression for λ that maximizes Φ becomes complex. When $x = \infty$, $M/M/\infty$ yields $\Phi = \frac{X}{R} = \frac{\lambda}{\frac{1}{\mu_y}} = \lambda\mu_y$, hence Φ is maximum when $\lambda = \infty$.

The crossover points which define the outer envelope of the various curves in Figure 7(a) indicate the specific values of λ where the allocator should switch to a different target partition size to keep Φ maximum. Given the specific workload characteristics, the λ subranges and the corresponding partition sizes that maximize Φ can be analytically derived. Figure 7(b) presents the number of

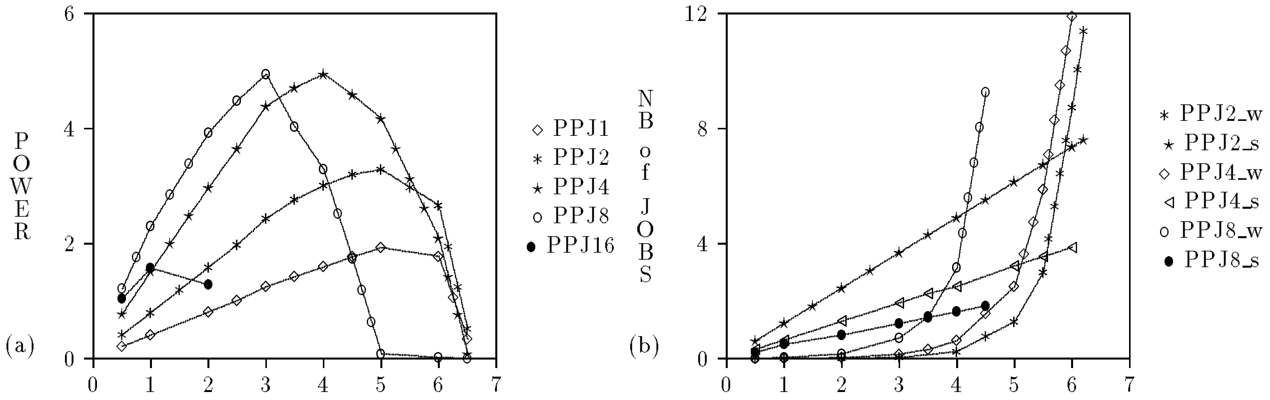

Figure 7: PPJ policies on WK3: power (a), waiting -_w- and in service -_s- jobs (b) *vs* the arrival rate λ.

waiting and in service jobs for the PPJ8, PPJ4, and PPJ2 policies for WK3 versus the arrival rate λ. The number of jobs in service increases linearly with λ, approaching the number of partitions. The number of waiting jobs increases convexly. For each policy, this pair of functions behaves similarly. For low values of λ, the average number of jobs in the waiting queue is less than the average number of jobs in service. After the crossover point, the number of jobs in the waiting queue surpasses the number in service and the distance between the two functions grows exponentially. The value of λ where the two functions intersect is approximately that λ value where switching between target partition sizes is required to maximize Φ. This observation is used as a simple control mechanism for the *split* and *merge* strategies. From variations of this simple mechanism, the specific versions of AP1–AP5 were derived. Figure 8 shows that AP4, one of the best versions of the adaptive policies, approximates closely the outer envelope of the fixed PPJ policies.

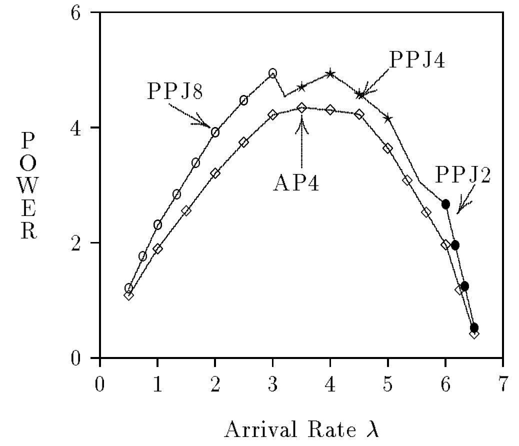

Figure 8: Outer envelope of the PPJ policies and the AP4 policy comparison (WK3).

5 Policy Comparison and Robustness

In this section, the policies are compared against each other under various workload assumptions. Several criteria can be adopted to compare the quality of each policy: the performance objective function Φ, the system load, and the workload type. As an example, for a given λ and for a given workload, a specific PPJ policy may yield the highest Φ value. However, that policy is optimal only over a limited load and workload type. The EPM policies, by contrast, do not reach power values that are as high, but they are effective over the entire range of λ. The insurance 75% policy performs well on certain workloads (*e.g.*, WK4), but performs poorly on other workloads (*e.g.*, WK1). *Robustness* is used as an overall evaluation criterion. A robust policy is one that achieves good performance over the entire load range and over the entire range of workload types. A specific scoring function is defined to be the sum of the values of Φ over a finite set of load levels. This function is used to classify and rank the different policies under various assumptions[1]. The values of the scoring function are normalized. Thus, the policy with the highest scoring function value is given a score of 100. The lowest policy's score is zero. By considering the various workload types (WK1-WK4) and the various loads (0%-100%), the overall robustness of each policy can be quantified. This provides a simple method for identifying the most robust policies when the workload type and the arrival rate are unknown.

In Figure 9 plain bars represent the rankings across all workloads, and dotted bars represent the rankings for WK1. For workload WK1, the best policy is PPJ16. This is because WK1 exhibits nearly linear speedup.

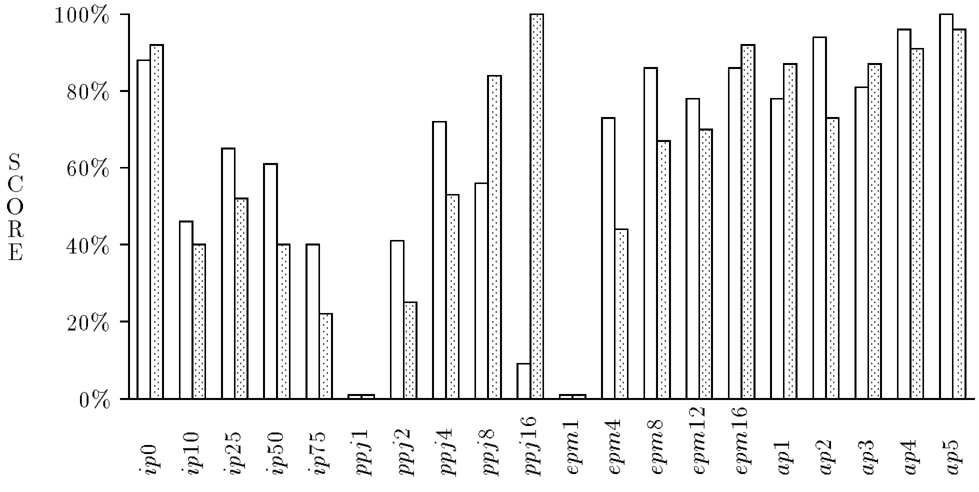

Figure 9: Policy rankings across all workloads (plain bars) and for WK1 (dotted bars).

If the workload type is unknown, however, the overall ranking suggests AP5 is best (*i.e.*, is the most robust). Thus, PPJ16 is optimal for some workloads but is volatile and is not suitable if

[1]Other functions could have been selected (*e.g.*, the sum of the distance of each policy's power curve from the optimal power curve) but the chosen function was selected for its computational simplicity and effectiveness in quantifying the idea of robustness.

the workload type or arrival rate are unknown. In contrast, AP5, although not optimal on WK1, performs adequately across a wide range of workloads and arrival rates. The difference between optimal and adequate performance depends upon the assumed knowledge about the workload and the arrival rate characteristics. In actual systems such knowledge is rarely available *a priori*, and it is also expensive to acquire. Thus, a relative ranking of the policies across a broad range of workload types and load levels is useful. If the system operates at a known load level or on a known workload type, that information can be used in the policy ranking. In Figure 10, plain bars represent the policy rankings assuming that the load level is known to be 10% and the workload is unknown. The dotted bars represent the rankings assuming that only the load level of 50% is known. The solid bars represent the rankings assuming that both the load level is known to be 50% and that the workload is of type WK4.

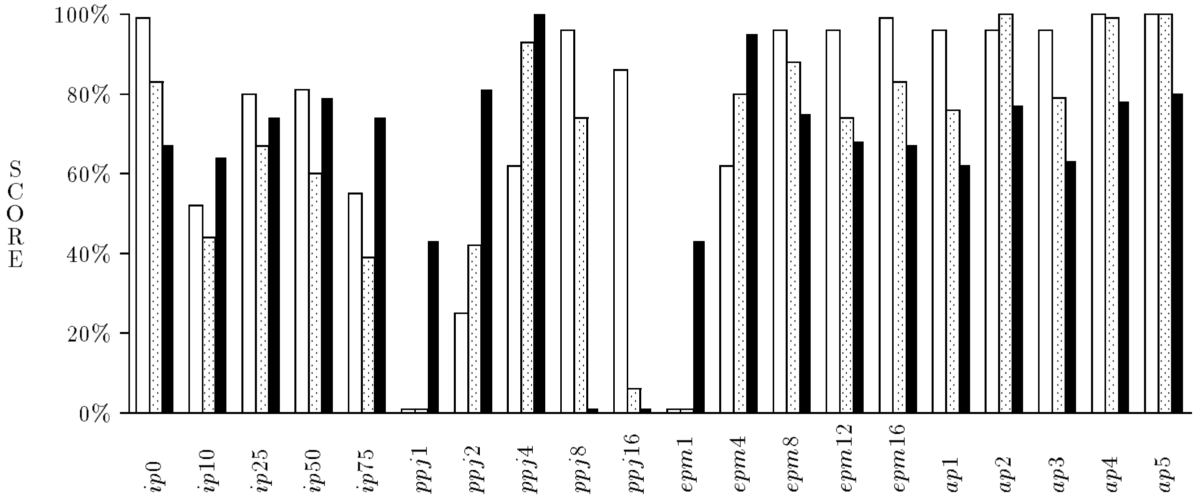

Figure 10: Policy rankings across all workloads at the 10% load level (plain bar), at the 50% load level (dotted bar), and at the 50% load level for WK4 only (solid bar).

Assuming that only a 10% load level is known (*i.e.*, the plain bars in Figure 10), PPJ16 and EPM16 are reasonable policy choices. However, PPJ16 is a poor choice when the load rises to 50% (*i.e.*, the dotted bars). Knowing also that the workload is of type WK4 (*i.e.*, the solid bars), the PPJ2 policy is a reasonable choice. However, PPJ2 would result in poor performance if such knowledge were not assumed. In fact, PPJ4 is the best policy for WK4 at the 50% load level, while AP4 and AP5 are the best at the 10% and 50% levels when the workload is unknown. It is evident that a more accurate workload characterization (*i.e.*, knowing the workload type and the system load level) improves performance. However, the cost of obtaining this information may be significant. The trade-off between such a cost and the quality of the performance is the primary motivation for the rankings.

6 Generalizations

Generalizations to policy analysis are presented here. Instead of comparing all policies, a specific policy from each policy family is selected. Namely, PPJ8, EPM8, IP50, and AP4 are examined. Various interarrival and service time distributions, and multi-class workload mixes are considered.

Interarrival time distributions

Figure 11 presents a sensitivity analysis on several interarrival time distributions for a single class workload (WK2). Four distributions with the same mean and different coefficient of variation (CV) are examined, namely a deterministic distribution (CV = 0), a hypoexponential (CV = 0.5), an exponential (CV = 1), and a hyperexponential (CV = 3). When the CV is low, the policies achieve higher power values as expected since the behavior is more regular and queueing effects are minimized. As before, PPJ8 saturates faster than the other policies which have broader (*i.e.*, more robust) power curves. The relative rankings of the policies remain the same.

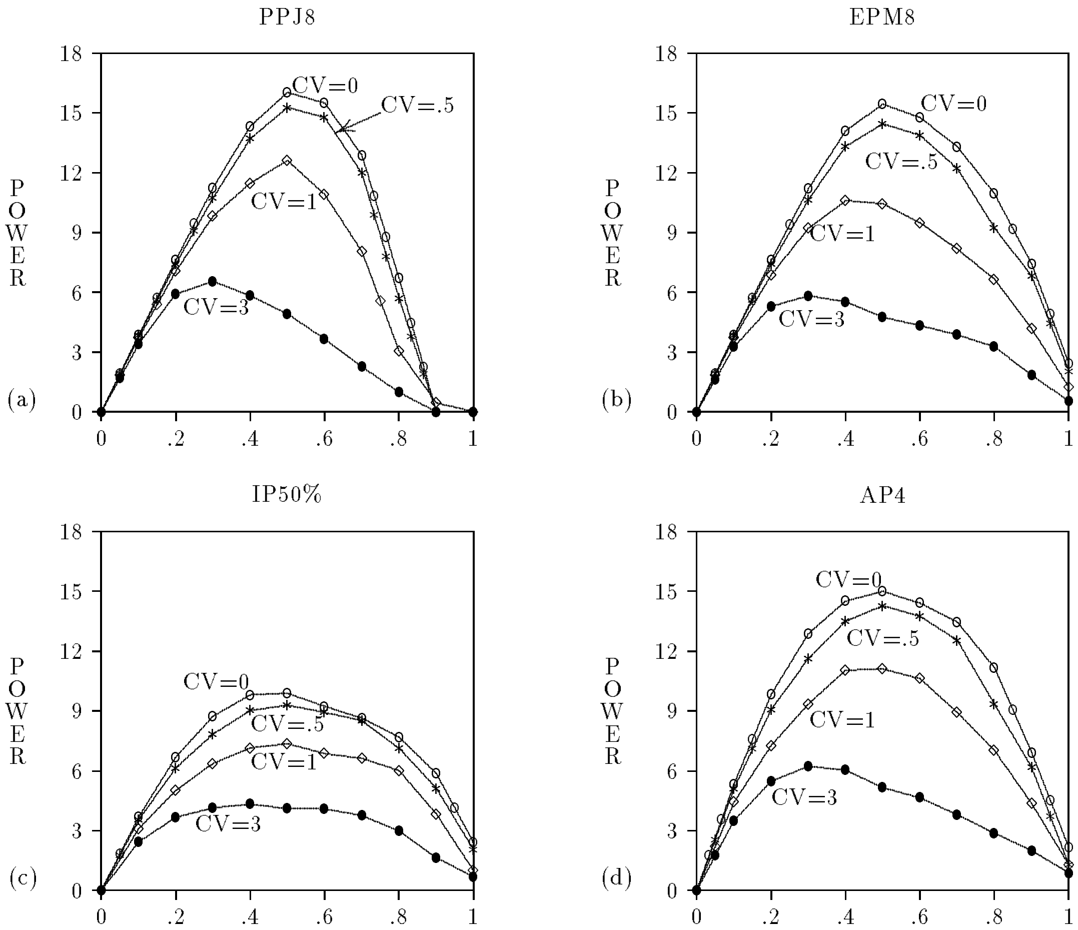

Figure 11: Power curves for WK2 *vs* system load for several interarrival time distributions.

Service time distributions

Figure 12 illustrates a sensitivity analysis on service time distributions for WK2. Four different service time distributions with the same mean (derived from the workload speedup curve, as in Section 3.1) are selected: deterministic (CV = 0), hypoexponential with (CV = 0.5), exponential (CV = 1), and hyperexponential (CV = 3). Again, better performance is achieved when the coefficient of variation is small. The relative performance of the policies remain the same as that presented in Section 5.

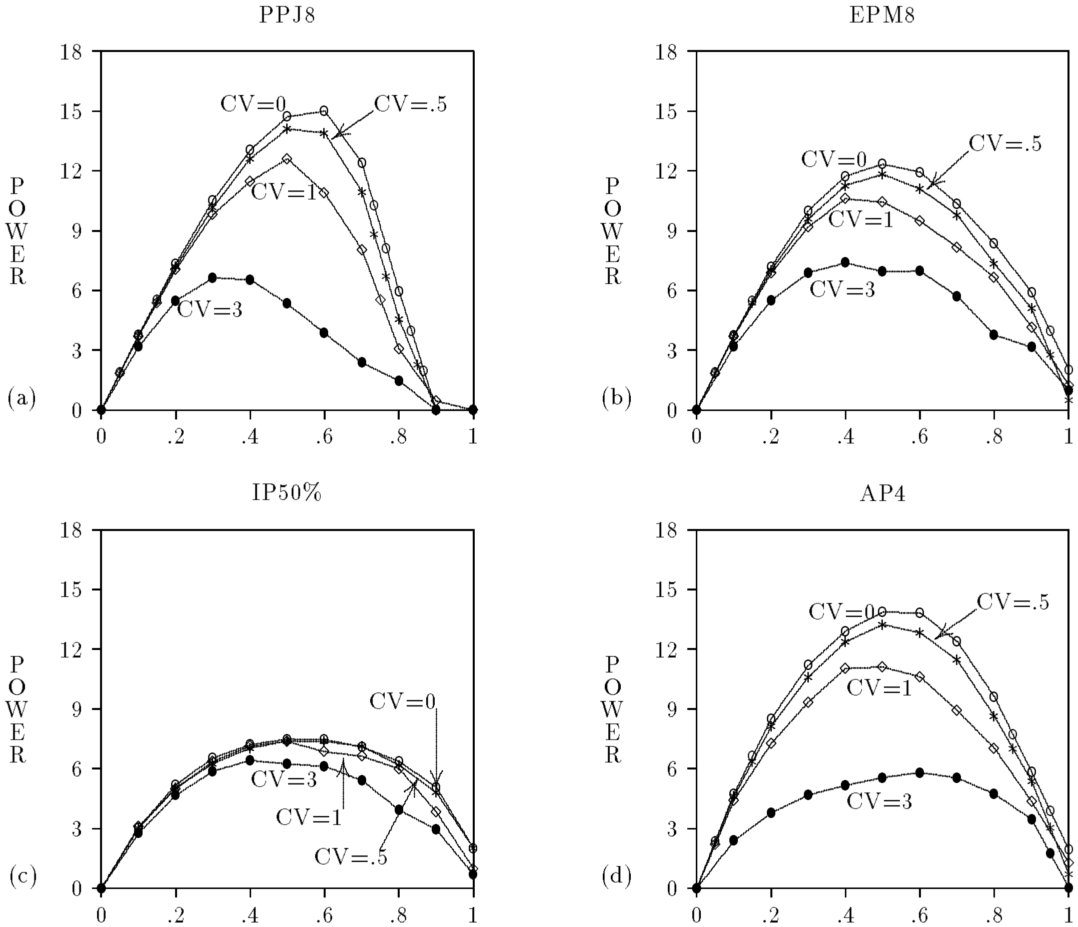

Figure 12: Power curves for WK2 *vs* system load for several service time distributions.

Multi-class workloads

The behavior of three different workload mixes is investigated as representative of multi-class workloads. Each mix contains a proportion of the four workloads presented in Section 3.1 and is described by the tuple [%WK1,%WK2,%WK3,%WK4]. The examined workload mixes are the following: MIX1 = [100%,0%,0%,0%], MIX2 = [40%,40%,10%,10%], MIX3 = [25%,25%,25%,25%], and MIX4 = [10%,10%,40%,40%].

Figure 13 illustrates the achieved power values for these mixes. PPJ8 is not robust, but still achieves good power values for a limited system load, depending on the workload mix. The IP50 policy is still the poorest of the adaptive policies examined, while AP4 and EPM8 achieve high power, with AP4 being the overall policy of choice.

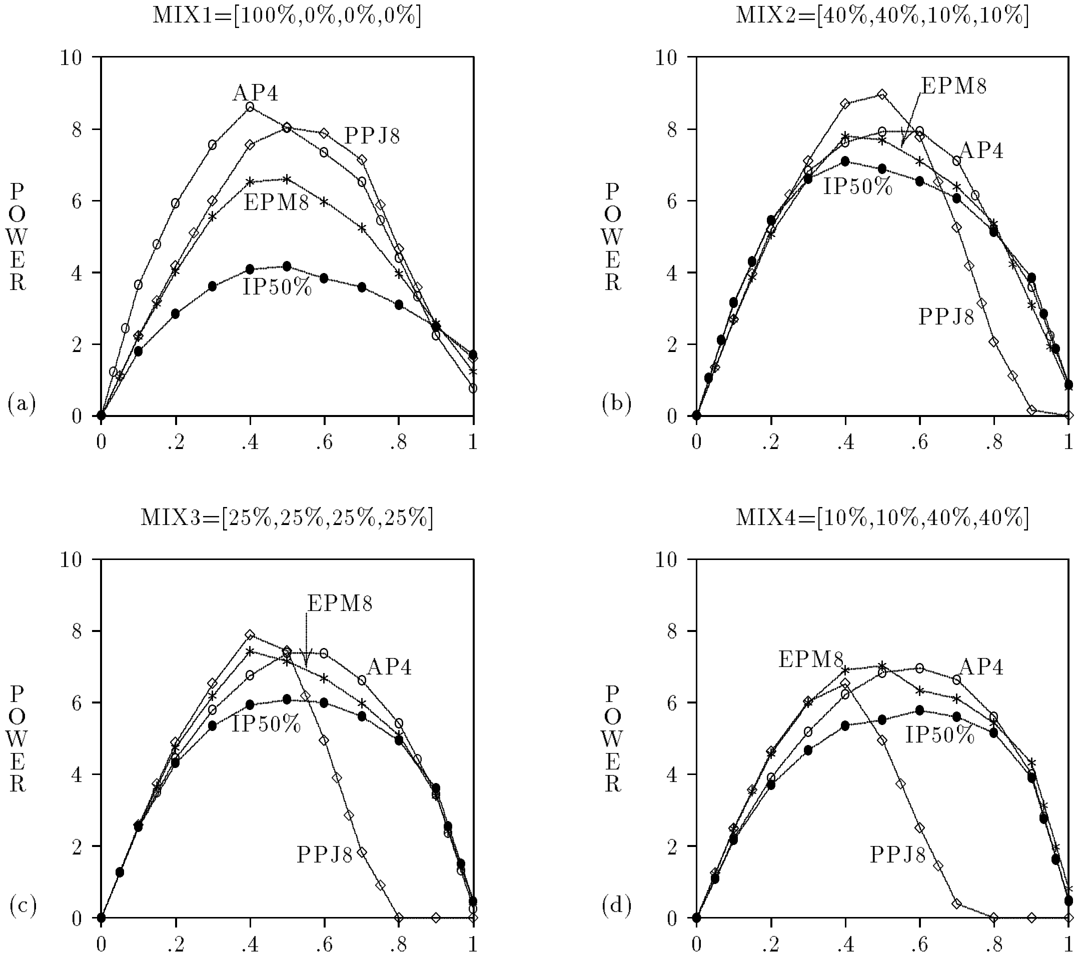

Figure 13: Power *vs* system load for several workload mixes.

Figure 13(a) shows the relative performance of these policies for MIX1 (*i.e.*, WK1). Their rankings are the same as those reported in Figure 9. The behavior of MIX2 is roughly similar to the behavior of WK2, the behavior of MIX3 is roughly similar to that of WK3, and the behavior of MIX4 is roughly similar to WK4. Indeed, looking at the solid bars in Figure 10, the relative ranking of the policies for MIX4 is similar to those for WK4. In summary, the relative ranking of the different policies (see Figure 9) is preserved across the various workload mixes. Thus, the robustness results apply to multi-class workloads.

7 Policy Implementation

The results of policy implementation and comparison on an Intel iPSC/2 hypercube system are presented in this section. The experiments were run on a dedicated system of 16 processors. A parallel Gaussian elimination algorithm for solving systems of linear equations was implemented and used as the benchmark for these experiments. The speedup curves for this application were derived by executing the algorithm on various numbers of allocated processors. The execution times of the algorithm are deterministic. Different matrix sizes were chosen to provide a range of speedup curves (see Figure 14) similar to those in Figure 2.

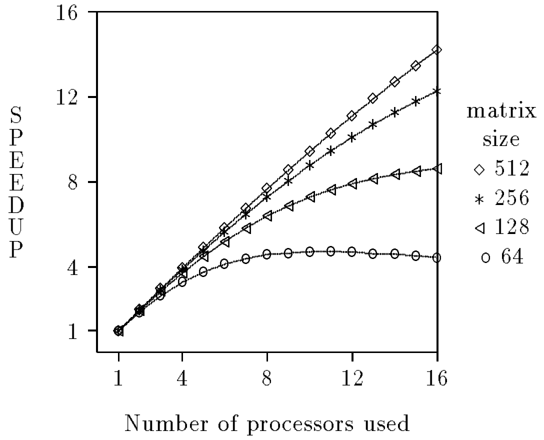

Figure 14: Speedup curves for the Gaussian benchmark.

The same policies that are selected in Section 6 are chosen for implementation: PPJ8, EPM8, IP50, and AP4. The results shown in Figure 15 represent typical comparisons between the policy simulations and observed performance. The simulations presented here were run with deterministic job execution times, to adequately model the measured ones. These experimental results confirm the behavior of the policies analyzed in Section 3.

8 Conclusions and Future Work

Several adaptive processor allocation policies have been introduced, analyzed, and compared. Each policy has been analyzed over a range of workload types and over the range of possible arrival rates. Factors influencing the performance have been identified. The comparisons illustrate the trade-off between system performance and the amount of known information. For example, if the speedup characteristics of the workload and the arrival rate are known, the allocation policy that yields the maximum power can be determined. However, that policy may have a narrow range of effectiveness and may perform poorly on other workloads or on other arrival rates. Robustness is a measure of how well a policy performs, even when limited or no knowledge of the workload type or arrival rate is available. Robust policies are especially useful on actual systems, since the parallel behavior of the workload and the arrival rate are unknown.

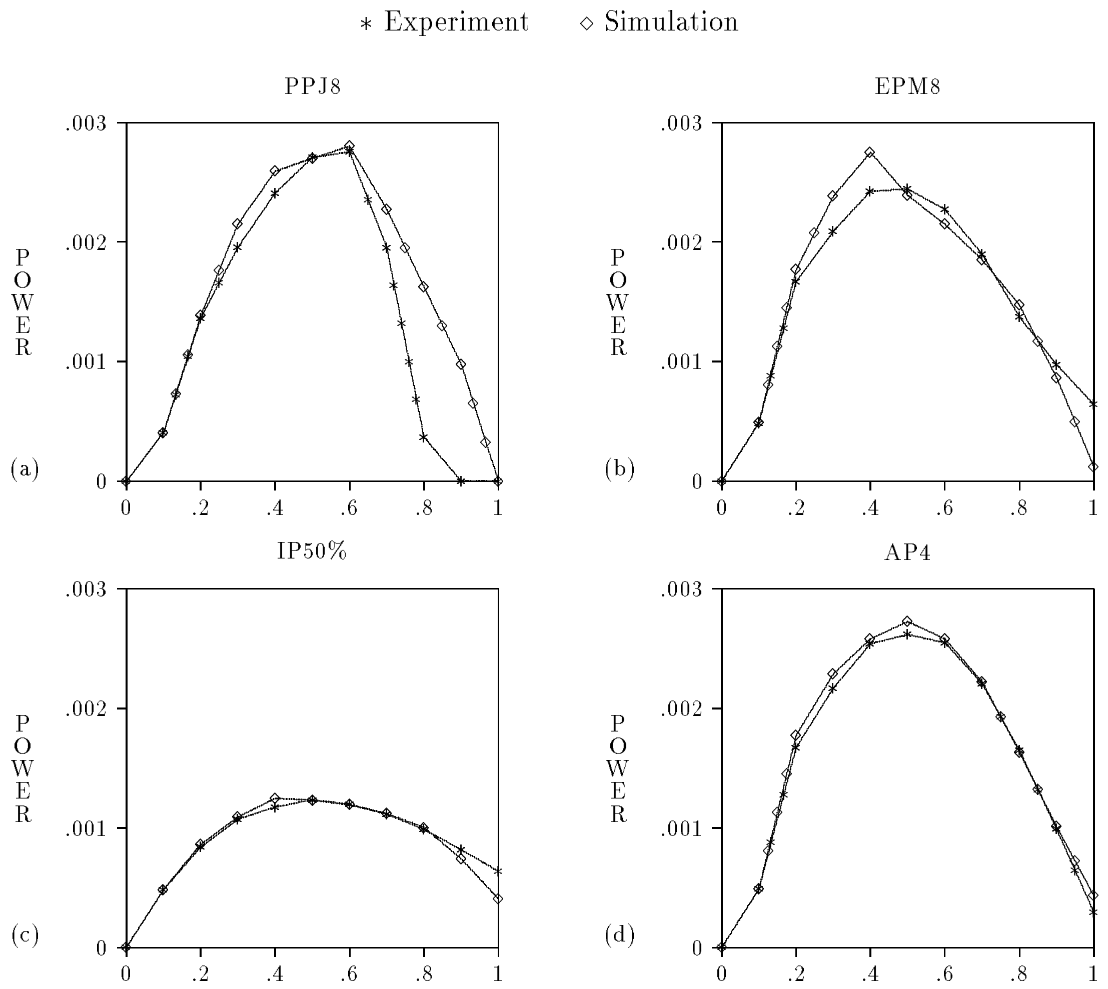

Figure 15: Power curves *vs* percentage of capacity for four policies on the iPSC/2 (workload type 2 – matrix size 256).

In this paper, adaptive space sharing policies where each job executes in a dedicated subset of processors are presented. Other partitioning schemes, that consider non-disjoint partitions, are subject of future work. A related issue concerns the degree of connectivity of the parallel machine. One extension of this analysis is to consider general (*i.e.*, not fully) connected machines, since the physical location of the processors assigned to an application affects the communication time of the application. A more detailed model of the system and a more detailed characterization of the workload are necessary in this case.

Acknowledgements
The authors would like to thank the Mathematical Sciences Research Section at Oak Ridge National Laboratory for providing access to the Intel iPSC/2 system.

References

[DL93] Dowdy L.W., Leuze M.R., "On modeling partitioned multiprocessor systems," to appear in *International Journal of High Performance Computing*, 1993.

[DCDP90] Dussa K., Carlson B., Dowdy L., Park K.-H., "Dynamic partitioning in a transputer environment," ACM SIGMETRICS 1990, pp 203-213, May 1990.

[GST91] Ghosal D., Serazzi G., Tripathi S.K., "Processor working set and its use in scheduling multiprocessor systems," *IEEE Transactions on Software Engineering*, Vol 17(5), pp 443-453, May 1991.

[EZL89] Eager D.L., Zahorjan J., Lazowska E.D., "Speedup versus efficiency," *IEEE Transactions on Software Engineering*, Vol 38(3), pp 408-423, March 1989.

[GGT91] Gelenbe E., Ghosal D., Tripathi S.K., "Analysis of processor allocation in large multiprocessor systems," *Proc. of the International Conference on the performance of distributed systems and integrated communication networks*, Kyoto, Japan, September 1991.

[GTU91] Gupta A., Tucker A., Urushibara S., "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications," ACM SIGMETRICS 1991, pp 120-132, May 1991.

[Klei79] Kleinrock L., "Power and deterministic rules of thumb for probabilistic problems in computer communications," *Proc. International Conference on Communications*, pp 43.1.1-43.1.10, June 1979.

[LDP89] Leuze M.R., Dowdy L.W., Park K.-H., "Multiprogramming a distributed memory multiprocessor," *Concurrency: Practice and Experience*, Vol 1(1), pp 19-33, September 1989.

[LV90] Leutenegger S.T., Vernon M.K., "The performance of multiprogrammed multiprocessor scheduling policies," ACM SIGMETRICS 1990, pp 226-236, May 1990.

[MEB88] Majumdar S., Eager D.L., Bunt R., "Scheduling in multiprogrammed parallel systems," ACM SIGMETRICS 1988, pp 104-113, May 1988.

[MEB91] Majumdar S., Eager D.L., Bunt R., "Characterization of programs for scheduling in multiprogrammed parallel systems," *Performance Evaluation*, Vol 13(2), pp 109-130, October 1991.

[Ous82] Ousterhout J. "Scheduling techniques for concurrent systems," *Proc. 3rd. International Conference on Distributed Computing Systems*, pp 22-30, October 1982.

[PD89] Park K.-H., Dowdy L.W., "Dynamic partitioning of multiprocessor systems," *International Journal of Parallel Programming*, Vol 18(2), pp 91-120, December 1989.

[Sim89] Simulog Corp., **The QNAP2 Reference Manual**, France, 1989.

[ST91] Setia S., Tripathi S.K., "An analysis of several processor partitioning policies for parallel computers," Computer Science Department, University of Maryland, College Park, Tech. Rep. CS-TR-2684, UMIACS TR-91-78, May 1991.

[Sev89] Sevcik K.C., "Characterization of parallelism in applications and their use in scheduling," ACM SIGMETRICS 1989, pp 171-180, May 1989.

[TG89] Tucker A., Gupta A., "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," *Proc. of the 12th ACM Symposium on Operating Systems Principles*, pp 159-166, 1989.

[ZM90] Zahorjan J., McCann C., "Processor scheduling in shared memory multiprocessors," ACM SIGMETRICS 1990, pp 214-225, May 1990.

[ZB91] Zhou S., Brecht T., "Processor Pool-based Scheduling for Large-Scale NUMA Multiprocessors," ACM SIGMETRICS 1991, pp 133-142, May 1991.