# Benefits of Speedup Knowledge in Memory-Constrained Multiprocessor Scheduling

Eric W. Parsons and Kenneth C. Sevcik

March 19, 1996

**Abstract**

Recently, research on scheduling for large-scale multiprocessor systems has begun to take into consideration the memory sizes of jobs. In general, the approach used to ensure good performance is to increase the multiprogramming level as the load increases. But if jobs require significant amounts of memory, then there will be a point at which the system can no longer fit additional jobs in memory. The scheduler will then have to carefully choose the way it allocates processors to jobs that are running in order to optimize performance.

In this paper, we investigate the benefit of having speedup knowledge about individual jobs in making such a choice. We find that if memory sizes and speedup characteristics of jobs are uncorrelated, then there may be moderate benefits in having speedup information, but if large-sized jobs tend to have better speedup than small-sized ones, as might occur in real systems, then much more significant benefits can be obtained. We propose scheduling strategies that exploit speedup information to improve performance, and evaluate them under a variety of workloads.

## 1 Introduction

Large-scale multiprocessor scheduling has been studied extensively for several years, but only recently has the issue of memory requirements been taken into consideration. Even with large memory capacities (1–2GB per node) becoming more common, the resource demands of large-scale scientific applications are expected to continue to strain the resources of systems [Ast93, AKK$^{+}$95]. For multiprogrammed multiprocessor systems, it is crucial that the allocation of processors and memory be coordinated in order to make effective use of the system resources.

An important factor determining how well a scheduler can perform is the amount of information it possesses about jobs arriving in the system. In this paper, we investigate the benefit of having knowledge of the speedup characteristics of jobs. Although our primary objective is to minimize mean response time, we begin by examining the achievable system throughput. Our experience has been that, once it is understood how to adapt to increasing loads, it becomes easier to define scheduling disciplines that perform well at both low and high loads. If only throughput is of concern and memory is abundant, then having speedup information is much less important because it is possible to maximize system efficiency by reducing processor allocations to a single processor. When main memory space is limited, however, the system cannot run an arbitrary number of jobs together, and must therefore allocate processors to jobs carefully. In particular, the scheduler must choose between giving more processors to efficient jobs to make better use of processors or to memory-intensive jobs to free up memory more quickly.

Recently, we investigated bounds on the achievable system throughput when nothing is known about the speedup characteristics of jobs [PS96]. For this case, it was shown that an equi-allocation strategy, where processors are allocated equally among running jobs, is the best. The assumption that no speedup information is known is actually quite restrictive, however, as it implies that there cannot be any correlation between memory size (which is assumed to be known) and speedup characteristics.[1]

In fact, we believe that a typical workload is likely to exhibit correlation among memory size, speedup, and service time. It is certainly true that such correlations exist in many individual applications, as represented by a variety of speedup models (e.g., [SN93, GGK93]). Larger-sized problems usually require more memory and attain better efficiency on a given number of processors. Such correlations will also exist in mixed workloads, albeit to varying degrees.

To develop practical algorithms that make use of speedup information, it is necessary to first acquire this information. In an environment where most of the workload consists of a small set of well-understood applications, it is relatively easy to measure the speedup of those applications directly and record the information for scheduling. In more dynamic environments, it is usually possible to collect speedup information of a job at run time with little user intervention [NVZ95].

Our baseline discipline allocates processors evenly among the jobs chosen to run in a given scheduling quantum (i.e., equi-allocation [MVZ93]). We first show that if no correlation exists between the memory size and speedup characteristics of jobs, then there is a moderate benefit in having speedup knowledge. If correlation does exist, then the potential improvement increases, theoretically to an arbitrarily large degree. Given that our job-selection strategy does not utilize speedup information, using equi-allocation can lead to poor performance because it makes poor processor allocation choices for certain sets of jobs selected to run. We thus propose some scheduling disciplines that use speedup information in allocating processors and show that these disciplines perform very well compared to equi-allocation in the case where memory size and job speedup are correlated. Although our focus throughout this paper is on shared-memory systems, where processors can be allocated independently of memory, we believe that our results provide significant insight for scheduling distributed-memory systems.

In the next section, we summarize previous research that has been conducted on both *processor-only* scheduling, where memory is not considered, and *processor-memory* scheduling, where both resources are considered. In Section 3, we examine how knowledge of speedup information can improve throughput. In Sections 4 and 5, we propose and evaluate scheduling disciplines that make use of such information. We then present our conclusions in Section 6.

# 2 Previous Work

## 2.1 Processor-Only Scheduling

A well-known principle of multiprocessor scheduling is that the average number of processors that are allocated per job must decrease as the system load increases. Since jobs tend to use processors more efficiently when given smaller allocations, this permits the system to run at a higher overall efficiency, thus avoiding saturation below the maximum load [Sev94]. Between very light load, where jobs receive all the processors they need, and heavy load, where each job receives a single processor, the way in which processors are allocated to jobs with different speedup characteristics determines the average response time observed.

[1] Otherwise, the correlation could be used to obtain information regarding speedup curves.

The way in which processors can be allocated to jobs differs among systems. In this paper, we will assume that the system supports *dynamic repartitioning*, where a job's processor allocation can vary during its execution. Although not available in most current systems, this level of flexibility can greatly improve performance, and as such is likely to be provided in future systems.

There are two ways in which dynamic repartitioning can be implemented. First, the system can inform a job that its allocation has changed; the job can then restructure itself to make use of more or fewer processors [TG89]. Applications that are based on task queues or whose parallelism occurs in independent loop executions can easily handle such reallocations. A complication is that performance may be lost if data is no longer arranged in physical memory appropriately after a reallocation. The second way to achieve dynamic repartitioning is to have the system multiplex processes of a job on processors, a scheme that can be quite successful if jobs do not synchronize too frequently [FR92, GTU91]. In distributed memory systems, a technique called *folding*, where processes are multiplexed evenly among all processors, can be used to ensure good load balancing [MZ94]. An advantage of these latter two approaches is that it may be easier to keep data close to where it is being accessed, as the mapping between processes and data remains constant.

More constrained types of processor-allocation schemes have been studied, motivated by assumptions about the software or hardware. With *fixed partitioning*, the system is configured with a static set of processor partitions. A user specifies, at submission time, the partition in which a job is to be executed, so the scheduler has no flexibility in reducing the size of processor allocations as the load increases. With *adaptive partitioning*, the scheduler can give a job any processor allocation upon activation, but preserves the same allocation throughout the job's execution [EZL89, GST91, ST93, CMV94]. This type of scheme can perform very well if all jobs have nearly the same processing requirements.

These latter disciplines are generally based on the assumption that jobs must be run until completion once activated. However, it has been shown that preemption is required to minimize mean response times when processing requirements have a high degree of variability [CMV94, PS95], as is the case in multiprocessor workloads that have been measured [FN95]. Otherwise, long-running jobs can monopolize the system resources, thus delaying short-running jobs from activation.

### 2.2 Processor-Memory Scheduling

In contrast to processor allocation, consideration of memory has only recently gained attention in multiprocessor scheduling research. McCann *et al.* consider how the minimum processor allocation requirements due to memory sizes affect scheduling in the context of the Intel Paragon [MZ95]. Their strategy is to balance the processing time received by each job as evenly as possible, with the goal of minimizing the mean processor allocation. Implicit in this goal is that lower mean processor allocations imply higher processor efficiencies; as such, they do not consider the use of speedup characteristics of individual jobs in allocating processors.

Peris *et al.* describe a technique to model the paging behaviour of parallel jobs when memory is constrained [PSN94]. They use this model to evaluate the performance of a real parallel job on a message-passing system as the number of nodes allocated to the job (and hence memory) is varied. Although the problem of scheduling jobs is not investigated, this work provides a method for modeling the effects of reducing the memory allocated to a job. It is, however, still unclear whether most (or even many) parallel scientific applications possess working set sizes much smaller than their total memory size [BHMW94].

Setia considered the use of paging in running parallel applications on distributed memory sys-

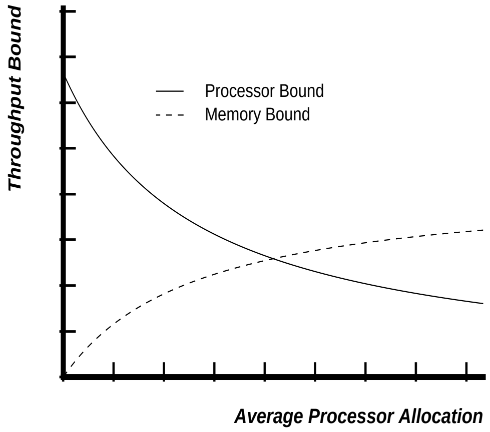



Figure 1: This graph illustrates the bounds on system throughput due to memory or processing as a function of the average processor allocation.

tems [Set95]. He compared a discipline that always allocates to a job as many nodes as required to satisfy its total memory requirements against one that uses paging to increase the multiprogramming level. He concluded that, with the overheads found in current systems, the paging approach was not suitable for typical parallel scientific applications.

An actual implementation of a multiprocessor scheduler that takes into account the memory sizes of applications is that of the Tera MTA [AKK+95]. This scheduler employs an algorithm for swapping jobs in and out of memory that minimizes the time memory is unavailable for running jobs. Moreover, it allows a priority to be associated with each job in the form of an expected space-time demand value; when a job's actual space-time usage drops below the assigned value, its priority is increased.

In our earlier work, we showed that there exists a simple relationship between the average processor allocation in the system and the bounds on throughput due to memory and processing [PS96]. This relationship is illustrated in Figure 1. As the average processor allocation increases, the throughput bound due to processing decreases as jobs execute with less efficiency. In contrast, as the average processor allocation increases, the throughput bound due to memory increases as jobs occupy memory for less time. The intersection of the two bounds represents the point at which maximum throughput can be obtained.

# 3 Throughput Analysis

In this section, we examine the difference in achievable throughput between an equi-allocation strategy having no information regarding the speedup characteristics of jobs and a non-equi-allocation strategy having complete information. Our system model consists of a simple shared-memory multiprocessor having $M$ units of memory and $P$ processors. (We generally normalize $M$ to 1 so that allocations can be thought of as fractions of the total memory size.) We assume that all jobs belong to a small number of classes which are characterized by the parameters listed in Table 1.

The memory size of a job can be interpreted either as the total amount of physical memory that will be accessed by the job (in a non-paging environment like the Tera MTA) or the size of the working set needed to keep the page replacement rate at a tolerable level.

| Workload Model Parameters | |
|---|---|
| $f_i$ | fraction of jobs of class $i$ ($\sum f_i = 1$) |
| $m_i$ | memory size of jobs of class $i$ ($m_i \leq M$) |
| $T_i(w,p)$ | execution time of jobs of class $i$ on $p$ processors given work $w$ |

Table 1: This table lists the parameters used in defining workloads.

To simplify the presentation of this section, which is concerned solely with throughput, we will assume that $w = 1$ for all jobs, and will use the notation $T_i(p)$ to denote $T_i(1,p)$. This does not influence the results of this section, because it is the amount of offered work in each class (as specified by $f_i$) rather than the number of jobs that matters in comparing the throughput of two disciplines.[2]

It has been previously shown that an equi-allocation strategy offers near-maximum performance if (1) no correlation exists between memory requirement and speedup curve, and (2) the scheduler cannot distinguish between jobs [PS96]. It is relatively easy to show, however, that this is not true if either of the two conditions is not met. In fact, if memory size and speedup characteristics are correlated, then techniques described in earlier work will not accurately predict the performance of even an equi-allocation discipline. As a result, new approaches are required to analyze the throughput, as we describe in the following sections.

### 3.1 General Case for Uncorrelated Workloads

To determine a bound on the maximum throughput of the system, we assume that all the job parameters described above are known. We also assume that there is a large queue of jobs waiting to be executed (such as would be the case in a heavily-loaded system) and require that jobs be consumed according to the fractions $f_i$.

To obtain the maximum throughput in the general case, we first enumerate all possible *configurations* of jobs to run at the same time, each one corresponding to a particular combination of jobs that fit in memory together, along with their processor allocations. (A simplified example is given in Figure 2.) There may be many different allocation choices for each combination of jobs, depending on the scheduling strategy being considered. We then specify a linear programming problem in which each configuration, $j$, is associated with a free variable $t_j$ representing the time for which that configuration is executed in a schedule.[3] The number of jobs of class $i$ consumed per unit time, given a processor allocation $p$, is $1/T_i(p)$; therefore, running a configuration $j$ for time $t_j$ will consume $t_j/T_i(p)$ jobs from class $i$ for each job in that configuration (substituting the appropriate value of $p$ for each job).

If $N$ is the number of jobs that pass through the system, then the scheduler must consume $f_j N$ jobs from each class $j$. But to obtain the optimal solution, we relax the constraint to be that *at least* $f_j N$ jobs be consumed; if a configuration is run and there are no jobs of a particular class left, then the processors corresponding to that job class would be left idle. The objective function

[2]For example, consider one workload in which two job classes have work values of 1 and 2, respectively, and another workload in which both classes have the same work values, but twice as many jobs in the second class. The amount of work generated in each class per unit time is the same for both workloads. Since the disciplines in this section ignore job boundaries, the amount of work completed by a discipline will be the same under both workloads. Thus, the ratio of throughputs of two different disciplines will be the same under both workloads.

[3]Note that we do not advocate the use of linear programming problems within a practical scheduler. In this section, the goal is to explore the limits on gains in performance from using application knowledge. More practical scheduling algorithms are proposed in Section 4.

**Classes:**

$m_1 = 0.9$
$m_2 = 0.1$

**Configurations:**

$t_1 : C_1(100)$
$t_2 : C_1(50), C_2(50)$
$t_3 : C_2(10), C_2(10), \cdots, C_2(10)$

**Linear Programming Problem:**

$$\begin{aligned} t_1/T_1(100) + t_2/T_1(50) &\geq f_1 N \\ t_2/T_2(50) + 10t_3/T_2(10) &\geq f_2 N \\ t_1 + t_2 + t_3 &= z(\text{minimize}) \end{aligned}$$

Figure 2: Predicting the maximum throughput of a workload given $P = 100$. The configurations are listed as class, followed by processor allocation. For this example, we used an equi-allocation strategy; in general, the second configuration would expand to all possible processor allocations, from $C_1(1), C_2(99)$ to $C_1(99), C_2(1)$.

is simply $z = \sum_j t_j$, which should be minimized. As $z$ gives the time needed to consume $N$ jobs, the throughput is $N/z$ (which is independent of $N$).

Recall that our goal is to investigate the throughput difference between a naive equi-allocation scheduler and one that possesses full knowledge of the characteristics of individual jobs. Clearly, the linear program models the latter perfectly, producing the best possible throughput. Modeling a naive equi-allocation discipline is a little more difficult because, by favouring some configurations over others, the solution to a linear programming problem can implicitly make use of the speedup characteristics of jobs, even if within each configuration processors are allocated evenly among jobs. For example, consider the system from Figure 2, and assume that the large jobs have perfect speedup and the small ones very poor speedup. The optimal solution to the linear programming problem will have $t_2 = 0$. This does not model naive equi-allocation accurately as knowledge of speedup characteristics has been used in avoiding the second combination.

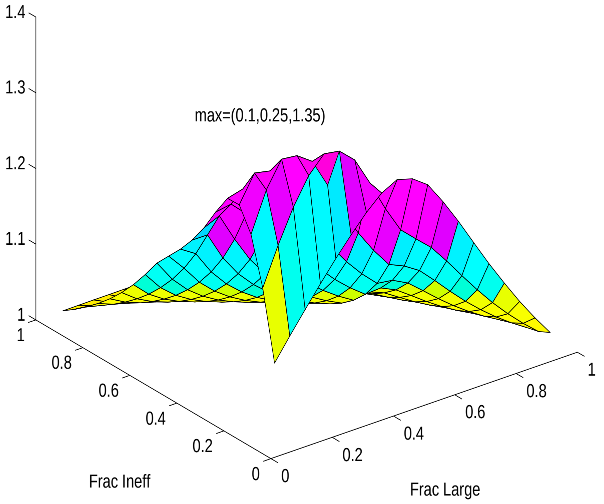
If there is no correlation between memory size and speedup characteristics, however, then the average execution time of a unit of work on $p$ processors will be the same in all memory classes, say $\overline{T}(p)$. In this case, the naive equi-allocation scheduler (called **naive equi**) can be modeled by aggregating all classes having the same memory size into a single class, having average execution time $\overline{T}(p)$. The full-knowledge scheduler (called **smart non-equi**) is still modeled by distinguishing between jobs having different speedup characteristics.

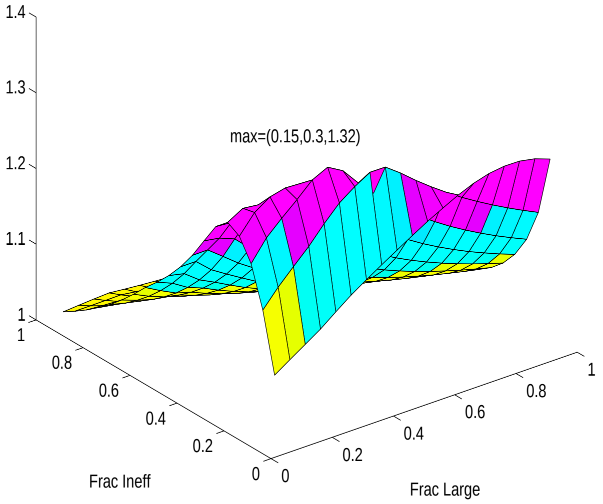
The relative performance of these two disciplines is shown for two two-memory-class workloads in Figure 3, assuming no correlation between memory size and speedup characteristics. In all graphs, we use execution-time functions of the form proposed by Dowdy [Dow88]:

$$T_i(w, p) = w(\alpha_i + (1 - \alpha_i)/p) \tag{1}$$

The values of $\alpha$ chosen for study, from 0.001 to 0.2, correspond to realistic speedup curves for jobs that speedup very well (91 on 100 processors) and that speedup very poorly (5 on 100 processors), respectively.

In the first example, the workload is composed of either very large or very small jobs. The graph




| **Uncorrelated Workload 1(a)** | | | |
|---|---|---|---|
| **Class** | $m_i$ | $\alpha_i$ | $f_i$ |
| 1 | 0.9 | 0.001 | $(1-g_{\rm I})\ g_{\rm L}$ |
| 2 | 0.9 | 0.2 | $g_{\rm I}\ g_{\rm L}$ |
| 3 | 0.1 | 0.001 | $(1-g_{\rm I})\ (1-g_{\rm L})$ |
| 4 | 0.1 | 0.2 | $g_{\rm I}\ (1-g_{\rm L})$ |

| **Uncorrelated Workload 1(b)** | | | |
|---|---|---|---|
| **Class** | $m_i$ | $\alpha_i$ | $f_i$ |
| 1 | 0.5 | 0.001 | $(1-g_{\rm I})\ g_{\rm L}$ |
| 2 | 0.5 | 0.2 | $g_{\rm I}\ g_{\rm L}$ |
| 3 | 0.1 | 0.001 | $(1-g_{\rm I})\ (1-g_{\rm L})$ |
| 4 | 0.1 | 0.2 | $g_{\rm I}\ (1-g_{\rm L})$ |

Figure 3: Throughput of smart non-equi relative to naive equi for two job-classes in an uncorrelated workload. The x-axis represents the fraction of jobs that are inefficient, $g_{\rm I}$, while the y-axis represents the fraction of jobs that are large, $g_{\rm L}$. The maximum point is shown in $(x, y, z)$ coordinates.

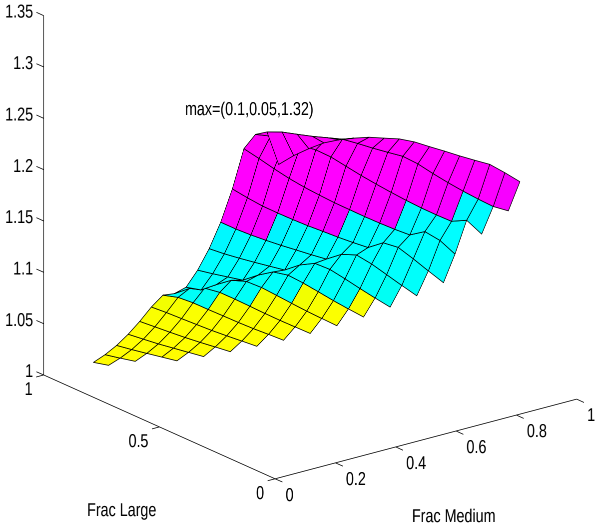
shows that moderate benefit can be obtained from using speedup information, yielding up to a 35% increase in throughput. As the discrepancy in speedup curves decreases, the maximum performance difference between naive equi and smart non-equi decreases (e.g., 16% if $\alpha_1 = \alpha_3 = 0.01, \alpha_2 = \alpha_4 = 0.2$, not shown). In this first example, however, it is necessary for the workload to consist of mostly small, inefficient jobs for a significant benefit to be obtained. This is contrary to our intuition regarding actual workloads, as we believe large jobs will generate the most work [FN95].[4] But if we decrease the size of large jobs, so that two large jobs can fit in memory together, we can observe increases in throughput across a wider variety of workloads. In particular, we can observe up to a 25% increase in throughput if most jobs are large and efficient.

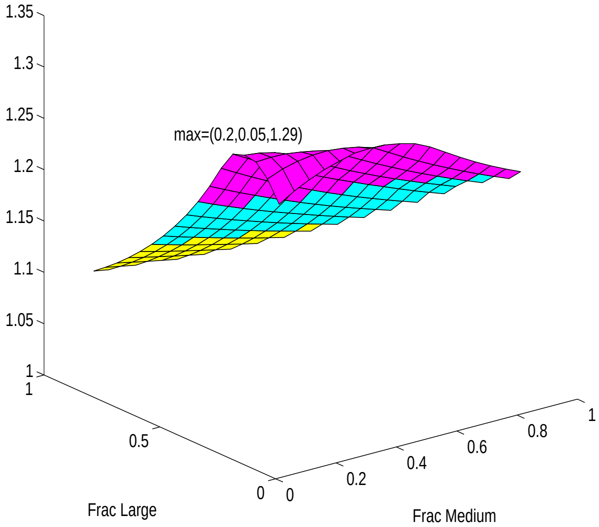
In Figure 4, we show similar graphs for a three-memory-class workload. These graphs are for the case where only 20% of jobs have poor speedup, since our two-memory-class case showed that it was necessary to have a small fraction of poor-speedup jobs in order to obtain any performance improvement.[5]

In the first graph, the potential benefit increases as the fraction of large-sized jobs decreases,

[4]We actually expect there to be fewer large-sized jobs than small-sized ones, but the amount of work to be generated by the large-sized ones to be much greater. Since, in this section, we assume all jobs require unit work, this corresponds to most jobs having large memory sizes.

[5]Because the total number of possible job configurations increases dramatically over the two-class case, it was necessary to limit the processor allocation choices to be a multiple of four (other than 1 and $P$). We do not believe this significantly affected the results as compared to allowing all possible processor allocation choices.




| **Uncorrelated Workload 2(a)** | | | |
|---|---|---|---|
| **Class** | $m_i$ | $\alpha_i$ | $f_i$ |
| 1 | 0.9 | 0.001 | $0.8g_{c1}$ |
| 2 | 0.9 | 0.2 | $0.2g_{c1}$ |
| 3 | 0.25 | 0.001 | $0.8g_{c2}$ |
| 4 | 0.25 | 0.2 | $0.2g_{c2}$ |
| 5 | 0.1 | 0.001 | $0.8(1-g_{c1}-g_{c2})$ |
| 6 | 0.1 | 0.2 | $0.2(1-g_{c1}-g_{c2})$ |

| **Uncorrelated Workload 2(b)** | | | |
|---|---|---|---|
| **Class** | $m_i$ | $\alpha_i$ | $f_i$ |
| 1 | 0.5 | 0.001 | $0.8g_{\mathrm{L}}$ |
| 2 | 0.5 | 0.2 | $0.2g_{\mathrm{L}}$ |
| 3 | 0.25 | 0.001 | $0.8g_{\mathrm{M}}$ |
| 4 | 0.25 | 0.2 | $0.2g_{\mathrm{M}}$ |
| 5 | 0.1 | 0.001 | $0.8(1-g_{\mathrm{L}}-g_{\mathrm{M}})$ |
| 6 | 0.1 | 0.2 | $0.2(1-g_{\mathrm{L}}-g_{\mathrm{M}})$ |

Figure 4: Throughput of smart non-equi relative to naive equi for three job-classes in an uncorrelated workload. The $x$ and $y$ axes represent the fraction of jobs that are large, $g_{\mathrm{L}}$, and medium, $g_{\mathrm{M}}$, respectively. For all cases, the fraction of jobs that have poor speedup is 20%.

being relatively insensitive to the fraction of medium-sized jobs. In the second graph, the range of potential benefit increases such that any choice of fractions can yield at least a 10% increase in performance. The reason is that large-sized jobs in the first graph can only run with small-sized jobs, while in the second, they can run with jobs of any size. This increases the opportunity for running inefficient large jobs with efficient ones.

## 3.2 Specific Case for Correlated Workloads

Next, we consider the situation where there exists a correlation between execution-time function and memory size. As described earlier, it is not possible to model the naive equi discipline using the linear programming approach for the general case. It is possible, however, to investigate a special two-job-class case analytically.

Let there be two classes of jobs to be scheduled such that $m_1 + nm_2 = M, m_1 > M/2$. We assume that a naive equi scheduler will try to maximize the use of memory whenever possible, as this has been shown to be crucial to achieving high throughputs and good response times in many other workloads [PS96]. This is the reason why naive equi will perform badly; if it could use the speedup characteristics of each job, the scheduler could trade off memory usage for processor efficiency. Given no knowledge of speedups, however, one must assume that maximizing memory usage at high loads is the best strategy.

There are only three possible combinations of jobs from which to choose: one class 1 job and $n$ class 2 jobs; $l = \lfloor M/m_2 \rfloor$ class 2 jobs; or (only if needed) one class 1 job on its own. Again, let there be $N$ jobs to schedule. The fraction of class 1 jobs that have $n$ class 2 jobs with which to run can be computed by taking the ratio of the time to execute all class 2 jobs ($n$ at a time) over that for all class 1 jobs:

$$\gamma = \min\left(\frac{(N/n)f_2T_2(P/(n+1))}{Nf_1T_1(P/(n+1))}, 1\right)$$

The time $\tau(N)$ required to execute all $N$ jobs is then:

$$\tau(N) = \gamma N f_1 T_1(P/(n+1)) + (1-\gamma)Nf_1T_1(P) + \left(1 - \frac{\gamma N f_1 T_1(P/(n+1))}{(N/n)f_2T_2(P/(n+1))}\right)(N/l)f_2T_2(P/l)$$

The first term is the time that class 1 jobs are executed with class 2 jobs; the second term is the time that leftover class 1 jobs are executed alone in the case that there are insufficient class 2 jobs to run with class 1 jobs; the final term is the time to execute leftover class 2 jobs because there are insufficient class 1 jobs. (Either the second or third term will be zero for any given choice of parameters.) This expression leads to a throughput of $N/\tau(N)$, which is independent in $N$.

Given this, the performance of naive equi relative to smart non-equi is shown in Figure 5. In the first graph, we let $\alpha_1 = 0.001$ and $\alpha_2 = 0.2$. As can be seen, up to a 75% improvement in throughput can be obtained for workloads comprised mostly of large jobs (i.e., most work is associated with large jobs). This corresponds to the type of workload that one might observe in a real system. The second graph shows that the potential performance improvement is larger in a case where the disparity in speedup behaviour between the two classes is extreme, but the improvement is never more than 100%. This limiting behaviour is captured by the following proposition.

**Proposition 1** *Given a workload consisting of two job classes such that $m_1 + nm_2 = M$, $m_1 > M/2$, an optimal scheduler given full knowledge of the speedup characteristics of jobs may have a maximum throughput that is, in the limit, $(n+1)$ times that of a naive equi-allocation scheduler that seeks to maximize memory usage.*

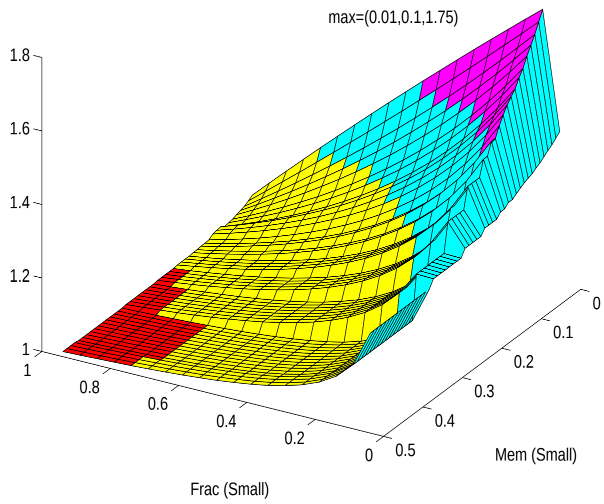
**Proof Overview:** Assume class 1 jobs have perfect speedup and class 2 jobs have no speedup. Let $m_2 \to 0$, $P = M/m_2$. Now, choose $f_1$ such that $f_1T_1(P/(n+1)) = f_2T_2(P/(n+1))$ (i.e., $\gamma = 1$). Since $T_1(P/(n+1)) \to 0$ and $T_2(P/(n+1)) = 1$ (i.e., a constant), $f_2 \to 0$ and $f_1 \to 1$. Thus, for naive equi, $\tau(N) = NT_1(P/(n+1)) = N(n+1)/P$. Smart non-equi, on other hand, will schedule class 1 jobs on their own, leading to an execution time $\tau'(N) = f_1NT_1(P) + f_2Nm_2/MT_2(1) = NT_1(P) = N/P$. □

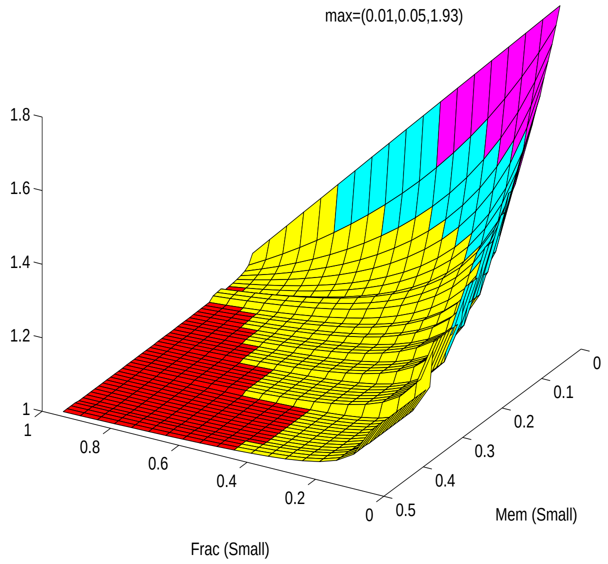
The cases considered in this section represent an extreme situation where naive equi will perform very poorly. Later, in the context of our simulation studies, we study naive equi given workloads with realistic degrees of correlation, where it compares more favourably to the optimal schedule.

## 4 Scheduling Disciplines

We now turn our attention to developing practical scheduling disciplines that exploit knowledge of speedup characteristics to improve throughputs and mean response times.

There are two essential steps that a processor-memory scheduler must perform. First, it must select jobs from those that are available to run during the next scheduling quantum. This selection is based on the objective chosen for the system, which in our case is to minimize mean response time. Second, the scheduler must assign processors to the jobs that have been selected. The




| **Correlated Workload 1(a)** | | | |
|---|---|---|---|
| **Class** | $m_i$ | $\alpha_i$ | $f_i$ |
| 1 | $1 - m_S$ | 0.001 | $1 - g_S$ |
| 2 | $m_S$ | 0.2 | $g_S$ |

| **Correlated Workload 1(b)** | | | |
|---|---|---|---|
| **Class** | $m_i$ | $\alpha_i$ | $f_i$ |
| 1 | $1 - m_S$ | 0.00001 | $1 - g_S$ |
| 2 | $m_S$ | 0.99999 | $g_S$ |

Figure 5: Throughput of smart non-equi relative to that of naive equi, given different memory sizes for small jobs, $m_S$, and different fractions of small jobs, $g_S$.

previous section showed that making poor processor allocation choices (given that job selection does not utilize speedup information) can lead to poor performance; in this section, we describe two scheduling disciplines that, by making better processor allocation choices, offer better performance when memory size and speedup are correlated.

### 4.1 Selecting Jobs

The basic strategy we use for job selection is to scan the list of jobs that are ready to run in order of increasing remaining service time, choosing those that can fit in memory (i.e., first fit). As the load increases, however, job selection is increasingly based on improving memory utilization, thereby increasing the bound on throughput due to memory. At heaviest load, remaining service time is not considered at all, thus allowing the greatest freedom to maximize memory utilization.

For this purpose, we use a subset-sum algorithm to find the set of jobs that maximizes memory utilization given a certain availability of memory. Although this problem is in general NP-complete, the size of problem with which we are concerned (i.e., up to 1000 jobs) can be easily solved by branch-and-bound algorithms that have been proposed. The one that we use is presented elsewhere, and will be referred to simply as the *subset-sum algorithm* [MT90].

The job selection algorithm, invoked at the beginning of each quantum, is defined as follows:

- Let the load $L$ be the number of jobs in the system, $N_{\text{ff}}$ be the number of jobs selected by the first-fit algorithm, and $\delta$ be a tunable parameter that determines how aggressively the scheduler seeks to maximize memory usage as the load increases. The first $N'$ jobs from first-fit are chosen for activation, where

$$N' = \left\lfloor N_{\text{ff}} * \max(1 - \frac{L}{\delta N_{\text{ff}}}, 0) \right\rfloor$$

We choose $\delta = 100$, which means that the scheduler will gradually decrease the value of $N'$ as the load increases, until $L$ is 100 times greater than the number of jobs selected using first-fit (after which point $N' = 0$).

- Given the amount of memory remaining after the first $N'$ jobs, the subset-sum algorithm is invoked to choose for activation a subset of the remaining jobs that maximizes the total memory usage.

An example of where such a selection algorithm is beneficial is a system with a two-class workload where one class requires a small amount of memory and the other a large amount. With first-fit, a steady-state situation can arise where there are always a small number of small jobs at the beginning of the run queue. These small jobs are scheduled with enough processors to keep their number relatively steady; as a result, the large jobs never get a chance to run. With the selection algorithm just described, if the load is sufficiently high (as indicated by a long queue of large jobs), the subset sum algorithm is invoked, allowing one of the large programs to be selected.

Under workloads for which first fit can achieve high memory utilization, the benefit of using a rule to maximize memory usage at high loads decreases. But even in the case of uniformly distributed memory sizes, we have found that using our job-selection algorithm can noticeably increase the throughput attainable before the system saturates.

### 4.2 Allocating Processors

Once the set of jobs to run has been chosen, the scheduler must allocate processors to these jobs. There are several approaches to making this allocation, two of which are described in this section.

The first approach is based on our earlier work on throughput bounds [PS96], which uses average processor and memory occupancies ($\theta_p$ and $\theta_m$, respectively) to determine the overall capacity of a system. Given a set of jobs $\mathcal{J} = \{j_1, j_2, \cdots, j_n\}$, such that job $i \in \mathcal{J}$ requires $m_i$ units of memory, is allocated $p_i$ processors, and has an execution-time function of $T_i(w_i, p)$,

$$\theta_m = \sum_{i \in \mathcal{J}} m_i T_i(w_i, p_i); \quad \theta_p = \sum_{i \in \mathcal{J}} p_i T_i(w_i, p_i)$$

In order to permit the highest possible system throughput ($X$), one must minimize the occupancies of memory and processors relative to the amounts available in the system, since throughput is bounded by:

$$X \leq \min(M/\theta_m, P/\theta_p) \tag{2}$$

Given a set $\mathcal{J}$, finding the processor allocations $\{p_1, \cdots, p_n\}$ that maximizes the throughput can be expressed as the following integer optimization problem:

$$\begin{array}{rll} \textbf{minimize:} & \max(\sum_{i \in \mathcal{J}} p_i T_i(w_i, p_i), \sum_{i \in \mathcal{J}} m_i T_i(w_i, p_i)) & \\ \textbf{subject to:} & \sum_{i \in \mathcal{J}} p_i & \leq P \end{array}$$

With our assumed execution-time functions (equation (1)), any increase in $p_i$ for any job $i \in \mathcal{J}$, will increase the processor occupancy but decrease the memory occupancy. But given that we may not use more than $P$ processors, any increase in allocation for one job must be balanced by a decrease for one or more others. Figure 6 illustrates the shape of the processor throughput and memory throughput surfaces for a three-job situation, assuming $p_1 + p_2 + p_3 = 100$. The memory throughput surface is always concave, as a low processor allocation will lead to a high memory occupancy for any given job. The processor throughput surface is always convex, as processor occupancy increases faster and faster with processor allocation.

Since service-time distributions are highly variable, using a job's service demand in computing processor allocations that minimize occupancies can lead to extremely varied results. Thus, we perform the optimization using the same value of work for all jobs (e.g., 1). Our reasoning is that in each quantum, the scheduler will schedule one unit of work from each selected job; if a job reappears in a later quantum, then this corresponds to another unit of work from that job. Another option would be to perform the optimization using the expected fraction of work appearing in each class, but we did not wish to include this type of knowledge in our scheduler.

Thus, to find the point of maximum throughput, we use the following approach (which requires a few milliseconds of computation time):

- Determine the processor allocation that minimizes the memory occupancy. Using Lagrange multipliers, the critical points of:

$$\sum_{i \in \mathcal{J}} m_i T_i(1, p_i) + \lambda \sum_{i \in \mathcal{J}} p_i - P$$

  occur where $\lambda = \left( \sum_{i \in \mathcal{J}} \sqrt{m_i(1-\alpha_i)/P} \right)^2$ and $p_i = \sqrt{m_i(1-\alpha_i)/\lambda}$.

- Check if the processor occupancy is larger than memory occupancy. If so, choose that pair of jobs for which reallocating a processor from one to the other leads to the greatest increase in $\min(M/\theta_m, P/\theta_p)$. Repeat this step until improvements are no longer possible.[6]

- Compare this allocation against equi-allocation; if equi is better (this happens on some rare occasions), then use it instead.

The occupancy-based processor allocation discipline is thus as follows:

**OCC** Given the set of jobs selected to run, apply the heuristic to obtain the processor allocation that maximizes the throughput based on occupancies from equation (2).

The second approach is to try to maximize the efficiency of the system, which is equivalent to maximizing the amount of work consumed per unit time: $\sum_{i \in \mathcal{J}} 1/T_i(1, p_i)$. In this case, a simple greedy algorithm can be used where the next processor is given to the job which increases the work consumption the most per unit time. Formally, begin by setting $p_i = 1$ for every $i$. Choose a job $j$ such that, for all jobs $k \neq j$,

$$\sum_{i \in \mathcal{J}, i \neq j} \frac{1}{T_i(1, p_i)} + \frac{1}{T_j(1, p_j + 1)} \geq \sum_{i \in \mathcal{J}, i \neq k} \frac{1}{T_i(1, p_i)} + \frac{1}{T_k(1, p_k + 1)}$$

and assign the next processor to $j$. Repeat this step until no processors are left. The efficiency-based processor allocation discipline is thus defined as follows:

[6]The nonlinear optimization formulation of this problem can actually be shown to be convex in positive $R^n$. We are currently investigating the use of standard nonlinear programming techniques to find the minimum occupancy value more efficiently and accurately.

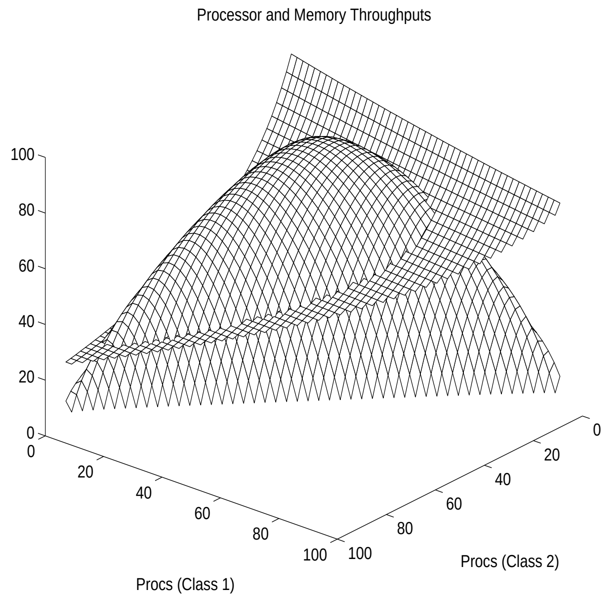



Figure 6: This graph illustrates memory and processor throughputs for a three-job situation. In this case, the maximum throughput occurs along the intersection of the concave memory throughput surface and the convex processor throughput surface.

| **Sample Processor Allocations — 2 Job Classes** | | | |
|---|---|---|---|
| | $\alpha_1 = 0.001, \alpha_2 = 0.2$<br>$m_1 = 0.8, m_2 = 0.1$ | $\alpha_1 = 0.2, \alpha_2 = 0.001$<br>$m_1 = 0.8, m_2 = 0.1$ | $\alpha_1 = 0.1, \alpha_2 = 0.2$<br>$m_1 = 0.5, m_2 = 0.25$ |
| **OCC** | 80-10-10 | 56-22-22 | 50-25-25 |
| **EFF** | 98-1-1 | 1-49-50 | 51-24-25 |
| **EQUI** | 34-33-33 | 34-33-33 | 34-33-33 |

Table 2: This table illustrates allocation choices made by different disciplines, assuming one class 1 job runs with two class 2 jobs. The allocations are shown in order of the class 1 job followed by the two class 2 jobs.

**EFF** Given the set of jobs selected to run, apply the greedy algorithm to maximize the consumption of work per unit time.

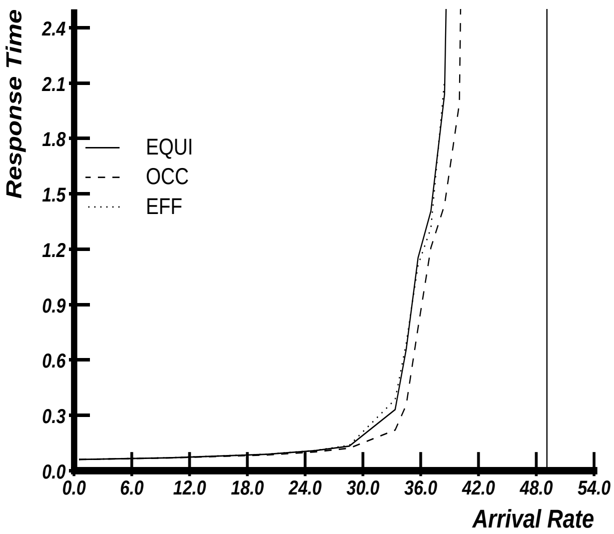
Finally, our baseline processor allocation discipline is:

**(Naive) EQUI** Given the set of jobs selected to run, allocate processors as evenly as possible.

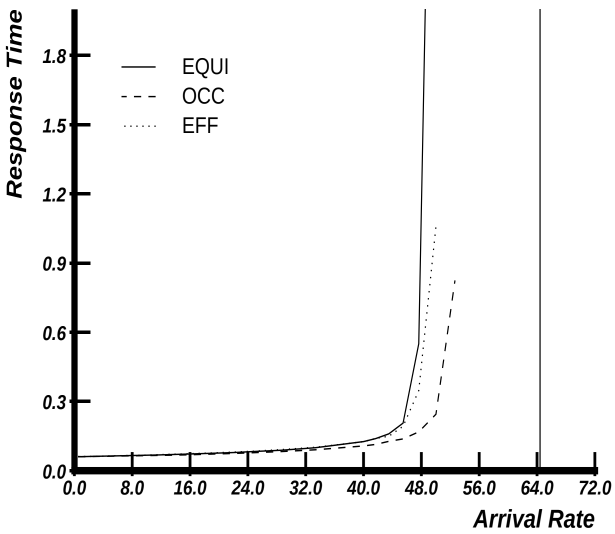
To illustrate the difference in processor allocations made by each of these algorithms, Table 2 illustrates the resulting choices for the case when one class 1 job is running with two class 2 jobs, given various choices of $\alpha_i$ and $m_i$.

## 5 Simulation Study

The performance of the disciplines described in the previous section has been examined by simulation under various workload conditions. First, we consider the workloads that were used as examples in the analysis section. We find that there are minimal benefits in the case of the uncorrelated workloads from Section 3.1, but that there are great benefits in the case of the correlated




Uncorrelated Workload 1(a)
($g_{\mathrm{I}} = 0.25, g_{\mathrm{L}} = 0.25$)




Uncorrelated Workload 1(b)
($g_{\mathrm{I}} = 0.25, g_{\mathrm{L}} = 0.25$)

Figure 7: Performance of scheduling disciplines for two-job-class uncorrelated workloads from Figure 3. The vertical line represents the maximum sustainable load, as predicted by our model.

workloads from Section 3.2. Then we consider a variety of correlated workloads that are likely to be more representative of real systems, where we can also find significant performance improvement.

The simulation study assumes that there is no scheduling overhead. In particular, it assumes that jobs can be suspended, restarted, and repartitioned with no overhead. It also assumes that the execution time of a job is accurately modeled by an execution-time function of the form given by equation (1).

The inter-arrival-time distribution is assumed to be exponential, and the service-time distribution hyper-exponential with coefficient of variation of five. This type of distribution is consistent with observed workloads at high performance computing centers [CMV94, FN95]. For the most part, a sufficient number of independent trials were performed for each data point to obtain a confidence interval that was within 5% of the mean. A trial terminated when the first 500000 jobs that entered the system (after a short warm-up period) had departed. The mean response time for a trial was based only on these 500000 jobs.

## 5.1 Uncorrelated Workloads

In Figure 7, we show the performance of the scheduling disciplines for the two-job-class systems studied in Figure 3. We chose parameters for the memory sizes, execution-time functions, and fractions such that performance gains were achievable.

In these workloads, the improvements in mean response times are minor, occurring only at very high loads. The EQUI discipline saturates in both cases very close to the value predicted by our model (40.45 and 51.63, respectively); the OCC discipline saturates at an arrival rate that is 4% and 5%, respectively, greater than that of EQUI. Improving the performance beyond what is shown has proven difficult, short of embedding the optimal solution in the scheduler and ensuring that different job configurations are selected in the correct proportions.

To illustrate the difficulty of achieving better performance, we consider the first of the two

| Optimal Schedule for Two-Job-Class Problem | | | | |
|---|---|---|---|---|
| time | lg ineff | lg eff | sm ineff | sm eff |
| $t_1 = 23.8\%$ | | | 10; 100 | |
| $t_2 = 10.1\%$ | | 1; 96 | 1; 4 | |
| $t_3 = 44.5\%$ | 1; 35 | | | 1; 65 |
| $t_4 = 21.6\%$ | 1; 74 | | 1; 26 | |

Table 3: This table gives the optimal schedule for the workload studied in Figure 7. Time is in terms of a fraction of the total, and entries correspond to the number of jobs selected to run from the given class, followed by the total number of processors allocated to those jobs.

workloads ($m_1 = 0.9, m_2 = 0.1$). The optimal solution as given by the linear programming model is given in Table 3.

The maximum throughput is very sensitive to the way in which large inefficient jobs are scheduled; any deviation from the indicated values in the processor allocations for this class leads to significant decrease in achievable throughput. For example, using the processor allocation choices made by the OCC discipline for the above configurations (which only really differs in the third line where large jobs receive 75 processors and small ones 25) results in a maximum throughput that is 6.2% worse than the optimum. More importantly, increasing the fraction of large inefficient jobs to 25% (from 6.25%) results in a drastically different processor allocation in the optimal case for the third configuration, giving more processors to the large inefficient jobs than the small efficient ones.

In an attempt to improve the performance of the disciplines, we experimented with a different job selection strategy which avoided running inefficient jobs without efficient ones, and which avoided running more than one efficient job at any time. The intent was to maximize the efficiency of the system by always having an efficient job available. This change did not lead to any noticeable improvement, however, as efficient jobs were still being consumed too quickly for there to always be one available. (Note that the optimal solution runs inefficient jobs together 21% of the time.)

We conclude that for this uncorrelated workload, it is difficult to obtain better performance given only information about the jobs currently in the system. If there is a large backlog of jobs such that the fraction of work in each class is representative of the workload, then it would be possible (albeit expensive) to use a linear programming approach to find the best solution. If this is not the case, however, then any job selection and processor allocation strategies based on knowledge of jobs in the system are likely to be far from optimal.

## 5.2 Correlated Workloads

In Figure 8, we show the performance of the scheduling disciplines for the special-case two-job-class correlated workload where $m_1 = 0.9, m_2 = 0.1, \alpha_1 = 0.001, \alpha_2 = 0.2$. In the first graph, we set the fraction of large, efficient jobs to 94% and in the second to 83%.[7] As can be seen, in both cases, OCC and EFF reach the maximum possible load. EQUI, on the other hand, only reaches about half the attainable load, as predicted by our earlier analysis in Section 3.2. (EFFMULT10 and EFFPWR represent disciplines similar to EFF, but with which processor allocations are constrained. The behaviour of these disciplines are discussed later, in Section 5.3.)

[7]These fractions correspond to the best case for non-equi allocation and worst case for equi-allocation, respectively. They also happen to be in the range one might expect for actual workloads.

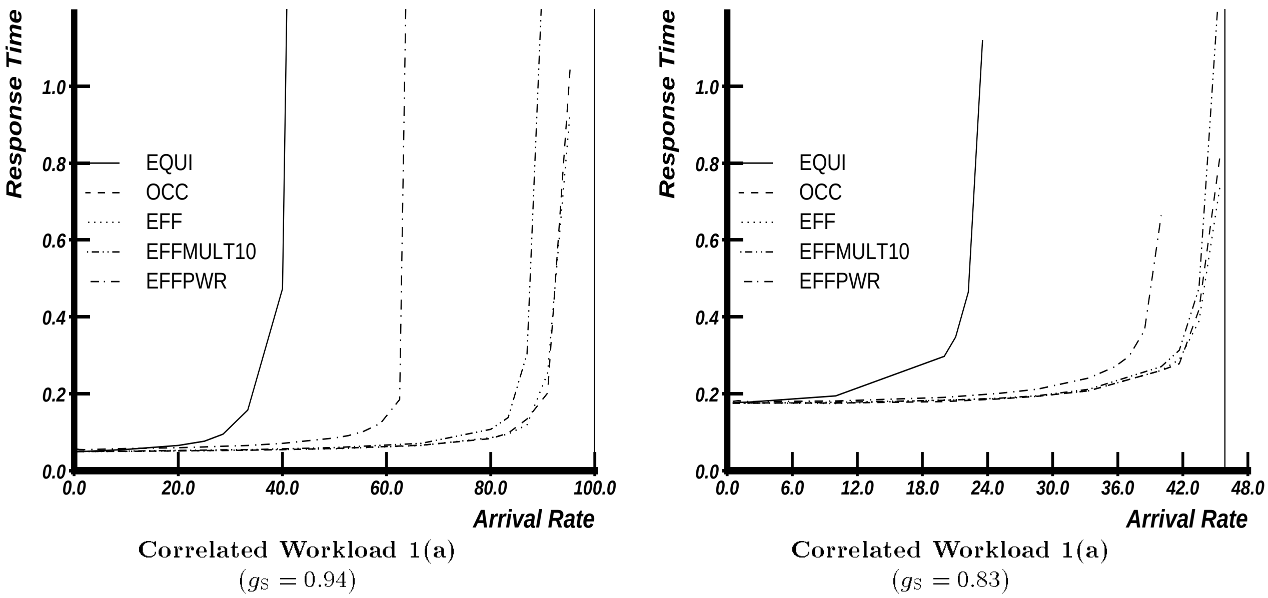



Figure 8: Performance of scheduling disciplines for two-job-class correlated workloads from Figure 5(a), using two different workload mixtures.

Next, we investigate the performance of the scheduling disciplines given more general models of correlation between memory size and speedup characteristics. As jobs in real workloads can be of any size, we draw the memory size of a job from a uniform distribution. This distribution is relatively consistent with that observed in real workloads [FN95]. However, this workload study also shows that there exists a correlation between memory size and service demand. In investigating such correlations, we found that there was little difference in the results. We thus only present the case where memory size and speedup are correlated.

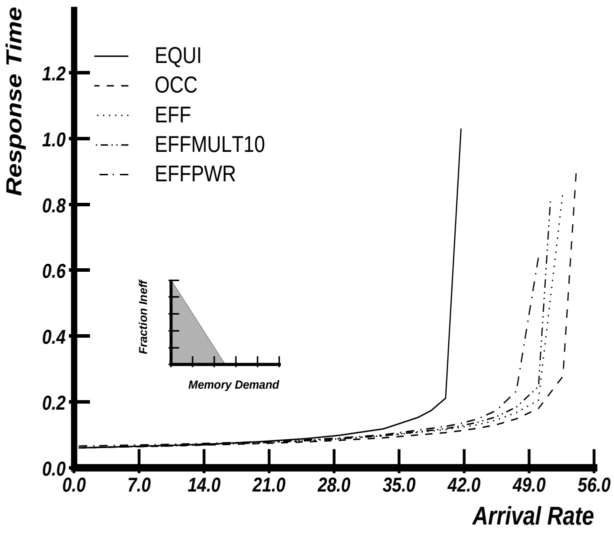
The memory-speedup correlation is defined by a function $F : (0,1) \mapsto (0,1)$, used in the following manner:

$$\begin{aligned} m &= \text{Unif}(0,1) \\ \alpha &= \begin{cases} 0.2 & \text{with probability } F(m) \\ 0.001 & \text{otherwise} \end{cases} \end{aligned}$$

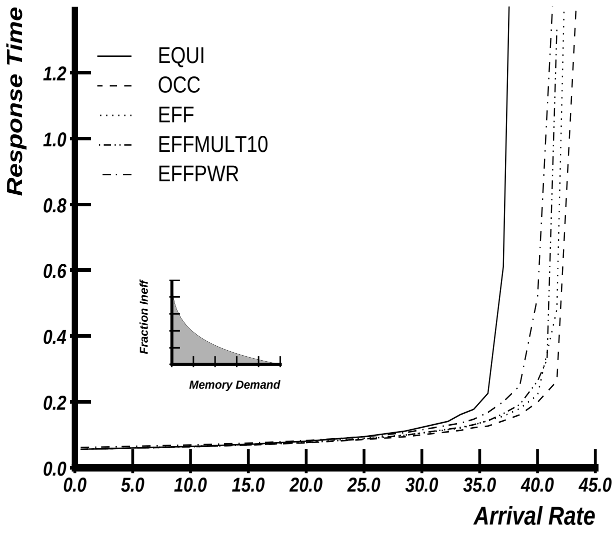
The performance of the various scheduling disciplines for our more general class of workloads is shown in Figure 9. The graphs inset in the performance graphs depict the correlation function $F$ used. In general, as the memory size increases, the probability of a job being inefficient decreases.

In all cases, the OCC discipline achieves the highest performance, closely followed by EFF. Also, the mean response times of OCC and EFF are very close to those of EQUI before EQUI leads to saturation; after this point, these disciplines continue to give good response times until they also cause saturation. To gain some sense of the performance of the disciplines relative to the maximum possible, we discretized the workloads into eight memory classes which we fed into the linear programming model. We found that OCC attained anywhere from 85% to 93% of the estimated maximum sustainable load.

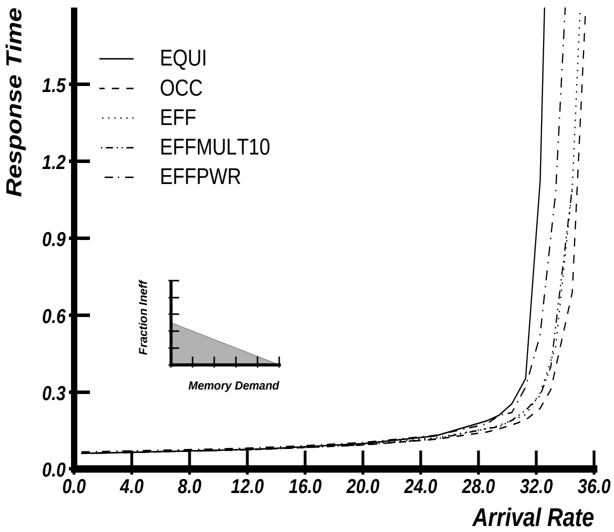
In examining a variety workloads, the following trends were observed:



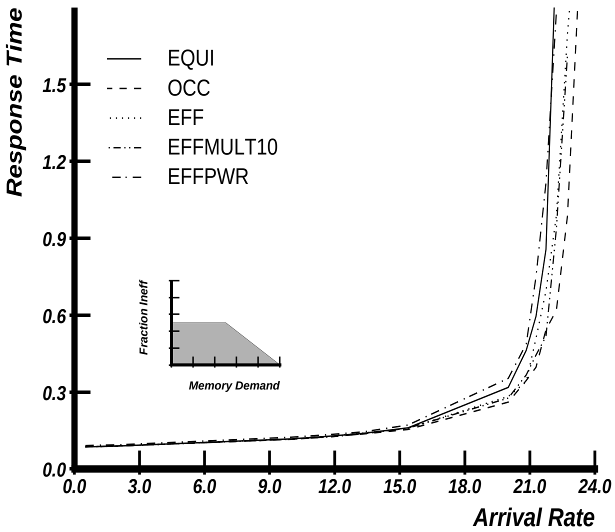

**Correlated Workload 4(a)**




**Correlated Workload 4(b)**




**Correlated Workload 4(c)**




**Correlated Workload 4(d)**

Figure 9: Performance of scheduling disciplines for general workloads having two speedup classes. The small inset graphs depict the correlation function $F$ used to define the workload.

- Performance gains from having speedup information are small unless more than 50% of the work is associated with jobs having good speedup, as there will not be enough efficient jobs to run with inefficient ones.

- As the expected memory-work for poor-speedup jobs (i.e., $\int xF(x)\,dx$) decreases, the improvement in throughput increases. The distributions in the graphs of Figure 9 have expected memory-work for poor-speedup jobs of 0.042, 0.061, 0.083, 0.1458, in that order. This mirrors the pattern of improvement over the four distributions.

The graphs and results shown here are for a representative sample of all the workloads that we studied. Some other workloads exhibited higher performance difference between EQUI and OCC and others less, but in no case studied did OCC or EFF perform worse than EQUI.

### 5.3 Constraining Processor Allocations

For many types of systems and applications, certain processor allocations are more natural than others. For example, in the shared-memory multiprocessor system being built at the University of Toronto [VBS+95], natural processor allocations are multiples of four or powers of two. Also, many applications are constrained to using specific allocations, such as an even number or power of two. For this reason, we investigated the effects of constraining processor allocations on the performance of the scheduling disciplines.

Given that EFF performed almost as well as OCC in most cases and it is simple to adapt, we chose it to be the basis for the constrained-allocation disciplines. The first constraint we studied was to limit allocations to be either a single processor, or a multiple of four and ten processors.[8] But as the performance for the multiple-of-four case was nearly the same as the no-constraint case, we only show the performance of the multiple-of-ten case. The performance of this discipline, labeled EFFMULT10, is shown in Figures 8 and 9. As can be seen, this discipline performs nearly as well as EFF, implying that we can constrain allocations in the system in this manner and still obtain excellent performance.

The second constraint was to limit allocations to be powers of two. Note that due to the way in which the algorithm is implemented, only 96 of the 100 processors were ever utilized. As can be seen, this discipline performs noticeably worse than its unconstrained counterpart. The reason is that the discipline suffers from not being able to allocate as many processors to efficient jobs as EFF can if one efficient job is scheduled with one or more inefficient ones. Thus, this discipline may benefit from better job selection to avoid situations in which it cannot make good processor allocations.

## 6 Conclusion

In this paper, we investigate the benefit of having speedup knowledge of individual jobs in multiprocessor scheduling. Our approach was to first model the performance of two classes of disciplines, those that have no knowledge of the speedup characteristics of individual jobs and those that have full knowledge. By using these models, we were able to determine how much benefit existed under various workload situations. We then proposed and evaluated two scheduling disciplines that make

[8] Our system is actually expected to have a power of two processors, in which case we would choose multiples of four and eight.

use of speedup knowledge, and showed that they perform well in the type of workloads likely to occur in real environments.

If memory size and speedup are uncorrelated, there is a moderate benefit from using speedup information in increasing the sustainable throughput. Obtaining these improvements in practice might be difficult, however, given the nature of the optimal solution (any deviation can lead to large performance degradations). When memory size and speedup are correlated, it becomes much easier to improve the performance in practice. In the cases we considered, we obtained anywhere from 85% to 100% of the maximum sustainable load. Although our primary interest was to minimize mean response times, we found that matching or surpassing the performance of our baseline scheduler, Naive EQUI, at all load levels to be easy even if our processor allocations are chosen to maximize throughput.

This paper focuses on jobs that had a very different speedup characteristics. We have found that such differences are crucial for one to obtain the level of performance improvement described in this paper. If most jobs share approximately the same speedup characteristics, then the benefits of using efficiency information naturally decreases. There is a need for better understanding of the types of workloads found in real environments, including the correlation between memory requirements, speedup characteristics, and service time.

## References


[AKK+95] Gail Alverson, Simon Kahan, Richard Korry, Cathy McCann, and Burton Smith. Scheduling on the Tera MTA. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 19–44. Springer-Verlag, 1995.

[Ast93] Greg Astfalk. Fundamentals and practicalities of MPP. *The Leading Edge*, 12:839–843, 907–911, 992–998, 1993.

[BHMW94] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proceedings Supercomputing '94*, November 1994.

[CMV94] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems*, pages 33–44, 1994.

[Dow88] L. W. Dowdy. On the partitioning of multiprocessor systems. Technical Report 88-06, Dept. of Computer Science, Vanderbuilt University, 1988.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[FN95] Dror G. Feitelson and Bill Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 337–360. Springer-Verlag, 1995.

[FR92] D. G. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[GGK93] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[GST91] Dipak Ghosal, Guiseppe Serazzi, and Satish K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[GTU91] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, 1991.

[MT90] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley & Sons, 1990.

[MVZ93] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[MZ94] Cathy McCann and John Zahorjan. Processor allocation policies for message-passing parallel computers. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 19–32, 1994.

[MZ95] Cathy McCann and John Zahorjan. Scheduling memory constrained jobs on distributed memory parallel computers. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modelling of Computer Systems*, pages 208–219, 1995.

[NVZ95] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. Technical Report UW-CSE-95-10-01, Department of Computer Science and Engineering, University of Washington, 1995.

[PS95] Eric W. Parsons and Kenneth C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 127–145. Springer-Verlag, 1995.

[PS96] Eric W. Parsons and Kenneth C. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems (to appear)*, 1996.

[PSN94] Vinod G. J. Peris, Mark S. Squillante, and Vijay K. Naik. Analysis of the impact of memory in distributed parallel processing systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 5–18, 1994.

[Set95] Sanjeev Setia. The interaction between memory allocations and adaptive partitioning in message-passing multiprocessors. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, Lecture Notes in Computer Science Vol. 949, pages 146–164. Springer-Verlag, 1995.

[Sev94] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.

[SN93] Xian-He Sun and L. Ni. Scalable problems and memory-bounded speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, Sept 1993.

[ST93] Sanjeev Setia and Satish Tripathi. A comparative analysis of static processor partitioning policies for parallel computers. In *Proceedings of the International Workshop on Modeling and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 283–286, January 1993.

[TG89] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[VBS+95] Z. Vranesic, S. Brown, M. Stumm, S. Caranci, A. Grbic, R. Grindley, M. Gusat, O. Krieger, G. Lemieux, K. Loveless, N. Manjikian, Z. Zilic, T. Abdelrahman, B. Gamsa, P. Pereira, K. Sevcik, A. Elkateeb, and S. Srbljic. The NUMAchine multiprocessor. Technical Report 324, Computer Science Research Institute, University of Toronto, April 1995.