

Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling

Thu D. Nguyen, Raj Vaswani, and John Zahorjan
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA

Appears in *Job Scheduling Strategies for Parallel Processing*, Volume 1162 of *Lecture Notes in Computer Science*.
Springer-Verlag, 1996.

Abstract

We consider the use of runtime measured workload characteristics in parallel processor scheduling. Although many researchers have considered the use of application characteristics in this domain, most of this work has assumed that such information is available *a priori*. In contrast, we propose and evaluate experimentally dynamic processor allocation policies that rely on determining job characteristics at runtime; in particular, we focus on measuring and using job efficiency and speedup.

Our work is intended to be a first step towards the eventual development of production schedulers that use runtime measured workload characteristics in making their decisions. We consider two distinct scheduling scenarios: interactive systems, where minimizing response time is the goal, and batch systems, where maximizing useful instruction throughput is the goal. In both these environments, our experimental results validate the following observations:

- Despite the inherent inaccuracies of runtime measurements and the added overhead of more frequent reallocations, schedulers that use runtime measurements of workload characteristics can significantly outperform schedulers that are oblivious to these characteristics.
- Runtime measurements are sufficient for schedulers to achieve performance surprisingly close to that possible when efficiency and speedup information is available *a priori*.
- The primary performance loss, relative to the use of *a priori* information, is due to the transient decisions of the schedulers as they acquire information on the running applications, rather than to measurement and reallocation overheads.

Our experiments are performed using prototype implementations running on a 50-node KSR-2 shared memory multiprocessor.

1 Introduction

We consider the use of runtime measured workload characteristics in parallel processor scheduling. Although many researchers have considered the use of application characteristics in this domain, most of this work has assumed that such information is available *a priori*. While it is useful to understand how to best schedule a set of jobs given *a priori* information on their behaviors, in practice, it can be difficult to obtain and accurately specify such information. This difficulty arises because of factors such as the sensitivity of job performance to the input data set and to the relative locations of allocated processors on the machine’s interconnection network¹.

As an example, consider the speedup of MP3D, an application from the SPLASH [24] benchmark suite, when run on the KSR-2 multiprocessor. The KSR-2 has an interconnection network that is a hierarchy of rings. The basic communication time between two rings is roughly four times that for communication within any one [11]. Because of this, MP3D, which has poor locality, achieves optimal speedup at a number of processors that depends strongly on the location of allocated processors. In particular, if all allocated processors are located on the same ring, MP3D’s speedup peaks at 12 processors. If allocated processors are split across two rings, speedup peaks at 24 processors. Thus, in the case where the user requests 12 processors, but the ones allocated are spread across two rings, the achieved speedup is only 2/3 that of the actual optimum. Similarly, if the user requests 24 processors, but the ones allocated are all on one ring, the achieved speedup is also less than optimal.

In this paper, we focus on gathering information about the current workload at runtime and using this information to make scheduling decisions; in particular, we measure and use job efficiency and speedup. While at first glance, it would appear that runtime measurements of job behaviors are clearly useful, the actual situation is considerably more complicated. The value of runtime measurements to parallel processor allocation policies depends critically on the answers to the following questions:

- *How can speedup and efficiency be measured at runtime with acceptably high accuracy and low overhead?*
- *Do parallel applications have sufficiently stable characteristics that their recent past is a good indicator of the near future?*

¹Of course, supercomputer users running the same application repeatedly on similar data sets are accustomed to providing this information. However, at the very least, this is an inconvenience. At the worst, apparently insignificant changes in the data set may in fact have a substantial effect on the optimum allocation, although this could go undetected by the user.

- *How can the measures taken when an application is run on p processors be used to estimate its performance when run on q ?*
- *Do the costs of the potentially many reallocations (which are inherent in this approach) required in the search to find appropriate final allocations outweigh the benefits?*

Our goal is to help answer these questions. Taken in this context, this paper describes work intended to address the way in which future realizable schedulers might make use of information gathered at runtime, focusing particularly on job efficiency and speedup.

We begin by presenting a scheme that allows the runtime measurement of efficiency and speedup at low overhead. We then examine two distinct scheduling scenarios: interactive systems, where minimizing response time is the goal, and batch systems, where maximizing the rate at which useful work is completed is the goal. Both kinds of computing already have significant roles on existing large scale parallel platforms [7]. For the interactive environment, we propose a scheduler that uses measured speedups to adjust the processor allocation of each running job, attempting to maximize job speedup. For batch environments, we propose a scheduler that uses measured efficiencies to allocate processors in such a way as to *maximize system efficiency*.

We have implemented prototypes of both schedulers on a 50-node KSR-2. We evaluate the effectiveness of these prototypes using workload mixes comprised of hand-coded parallel applications from the SPLASH [24] benchmark suite and compiler-parallelized applications from the PERFECT Club [2] benchmark suite. Our central result is that the use of runtime measurements can improve system performance substantially, despite the inevitable noise in the gathered data and the associated overheads. Furthermore, schedulers that use runtime measurements can achieve performance surprisingly close to that possible when efficiency and speedup information is available *a priori*.

The remainder of the paper is organized as follows. In the next section, we discuss related work. Section 3 describes our technique for measuring job efficiency and speedup at runtime. Section 4 describes and evaluates a response-time oriented scheduler that makes use of these measurements. In Section 5, we turn to the problem of maximizing the completion rate of batch work, again proposing a policy that employs runtime measurements and evaluating its performance experimentally. Section 6 concludes our work.

2 Related Work

As previously mentioned, many researchers have studied the use of application characteristics by processor schedulers of multiprogrammed multiprocessor systems. Majumdar et al. [13], Chiang et al. [3], Leutenegger and Vernon [12], Sevcik [22, 23], Ghosal et al. [9], Rosti et al. [21] and others have proposed using application characteristics such as speedup, average parallelism, and processor working set to improve the performance of static processor schedulers. More recently, Guha [10] has proposed that application characteristics such as efficiency and execution time can also be used profitably by dynamic processor schedulers. All of these studies, however, assume that accurate historical performance data is provided to the scheduler at job submission time. In contrast, we concentrate on using recently measured characteristics of running jobs to optimize performance for the specific workload in execution.

McCann et al. [15] have proposed a dynamic scheduler that uses application-provided runtime idleness information to dynamically adjust processor allocations to improve processor utilization. This work differs from ours in two respects: (1) we consider all sources of inefficiency as opposed to just idleness, and (2) McCann et al.’s scheduler attempts to reallocate processors at a much finer grain than does ours. Thus, the effectiveness of their scheduler is dependent on the existence of application idle periods that are long relative to processor reallocation overheads.

Feitelson and Rudolph [8] take a similar approach to ours, proposing to dynamically gather information about communicating sets of processes in an attempt to relax the constraints of co-scheduling. Sobalvarro and Weihl [25] also propose several ways to use runtime identification of sets of communicating processes to relax the constraints of co-scheduling.

3 Measuring Job Efficiency and Speedup at Runtime

3.1 Measuring Efficiency and Speedup

The basic parallel job characteristics we wish to exploit are efficiency and speedup. While these measures are normally applied to the complete execution of an application (for example, speedup on P processors is the job completion time when run on P processors divided by the job completion time when run on a single processor), we take a more short-term view in our work. In particular, we wish to measure efficiency and speedup over the fairly short-term past, with the intention of relying on it as a predictor of the near-term future. We therefore use the terms efficiency and speedup

in this more instantaneous sense.

While efficiency and speedup are intimately related, in practice, efficiency is rather easily measured, whereas speedup is not. Thus, we only measure efficiency, or more precisely, we measure the inefficiencies due to overheads and subtract them from 1.0. Then, when necessary, we calculate speedup using:

$$Speedup(p) = Efficiency(p) * p \quad (1)$$

It is well known that loss of efficiency in shared memory systems arises from a combination of *idleness* (e.g, load imbalances, synchronization constraints, and sequential portions of execution), *communication*, *system overhead* (e.g., page faults and clock interrupts), and *parallelization overhead* (e.g., per-processor initialization, work partitioning, and synchronization). Of these four sources of loss of efficiency, it is particularly difficult to measure parallelization overhead for hand-coded applications because initialization, work partitioning, and synchronization code are typically embedded directly in normal application code. Fortunately, our experience with a wide variety of benchmark programs shows that parallelization overhead is typically small [18]. Thus, we require only estimates of the first three components (idleness, communication, and system overhead) to accurately assess efficiency.

On the KSR-2, we rely on a combination of hardware and software support to measure inefficiencies. Each node in the KSR-2 has a hardware monitoring unit that maintains three critical user-readable hardware counters: elapsed wall-clock time, elapsed user-mode execution time, and accumulated processor stall². Measuring communication and system overhead involves little more than periodically reading these counters. Measuring idleness is slightly more involved; we instrument all synchronization code in our runtime systems (KSR PRESTO [11] and CThreads [4]) to keep track of elapsed idle time using the wall-clock hardware counter. This idleness measurement scheme is relatively overhead free because idleness accounting is performed mostly when the processor would otherwise be idle. Of course, this approach assumes that all application synchronization takes place through calls to the PRESTO and CThreads libraries rather than through direct manipulation of shared variables. We did not, however, have to modify our applications to meet this assumption; none of our hand-coded programs violated this assumption while all synchronization in compiler-parallelized applications by definition takes place in the PRESTO runtime system.

²On shared memory systems such as the KSR-2, communication is required whenever data does not currently reside in the local cache, or is not in an appropriate state. Processors in many systems stall in this situation; that is, they execute no instructions until the remote data becomes available. Thus, processor stall corresponds to communication cost. On message passing machines, measuring performance loss due to communication would be even more straightforward, requiring only software support.

3.2 Measurement Interval

As will be seen below, in order for our runtime measurements to be useful, it is essential that comparisons of efficiency and speedup measurements made at different processor allocations be meaningful. Since *instantaneous* speedup reflects the characteristic of only a small section of the full application code, performing such comparisons can be problematic. This difficulty could be resolved by measuring efficiencies and speedups over relatively long intervals of time. Unfortunately, this approach has two disadvantages: (1) it would be difficult to determine what constitutes a sufficiently long period for an arbitrary application; and (2) long measurement intervals increase the latency of the scheduler in responding to changes.

Thus, we instead exploit a characteristic shared by a large variety of scientific parallel applications. In particular, we currently consider only *iterative* parallel applications. An iterative application is one in which the majority of the execution is driven by a sequential loop (whose bodies may be entirely general, involving the execution of many parallel phases, subroutine calls, etc.)³. Empirical evidence shows that successive iterations tend to behave similarly, so that measurements taken for a particular iteration are good predictors of near future behavior [18]. Thus, for such applications, we equate a measurement interval to an application iteration, providing a basis by which to reasonably compare a job's performance as its processor allocation is varied. Note, however, that in general, our approach does not require applications to be iterative. At a minimum, what we do require is that there be some identifiable point in the application's execution where it can indicate that a unit of work has been completed. For example, in a fairly coarse-grained application employing user-level threads as the basis of parallel execution, a work unit might be defined to be the work between the kernel thread's dequeuing and subsequent enqueueing (or termination) of a user-level thread.

4 Interactive Environments: Improving Response Time

In this section, we describe and evaluate a scheduling policy designed to improve response time in interactive environments through the use of runtime gathered job characteristics.

³In [18], we found that five of the ten SPLASH applications and all seven of the Perfect Club applications we could compile were iterative.

4.1 Policies

To evaluate whether runtime measurements can be used beneficially by a scheduler, we compare the multiprogramming performance of the following three policies:

EQUI: The basic scheduling policy on which we build is dynamic equipartition [26]. Under EQUI, each currently executing job is allocated an equal number of processors. Processor reallocations take place at job arrival and departure times. EQUI is representative of the space sharing approach to processor allocation that has been found to perform well for multiprogrammed shared-memory multiprocessors [26, 15].

ST-EQUI: At the highest level, the specific policy we propose to take advantage of runtime estimated speedup allocates an equal number of processors to each executing job, just as with EQUI. However, each time a reallocation takes place, each affected job engages in a *self-tuning* procedure [17] to estimate how many of its allocated processors should actually be used to maximize its current speedup. (We briefly describe self-tuning in Section 4.2.) It is well-known that many applications do not speed up monotonically with the number of allocated processors; instead, they slow down when executed on more processors than they can use efficiently. Thus, it is reasonable to expect such jobs to release excess processors because they have no incentive to keep them. Additionally, in any system that charges for resource use, there is a positive incentive to release excess resources. When one or more jobs release processors back to the system, the scheduler reallocates these processors as equally as possible among those jobs that can profitably make use of more than their fair share. A job that gives up processors can later ask for them back if its speedup changes.

AP-EQUI: AP-EQUI is similar to ST-EQUI, except that it uses *a priori* information on job speedup rather than runtime estimates. Given this information, AP-EQUI needs to reallocate processors only at job arrival and departure times. When reallocating processors, it gives each job no more processors than the number that maximizes that job's speedup.

It is intuitively clear that AP-EQUI should outperform ST-EQUI. Distinctions in performance between AP-EQUI and ST-EQUI serve to illustrate the impact of errors in our runtime measurements, as well as the overhead of more frequent reallocations and dynamic self-tuning.

The case is less clear for EQUI and ST-EQUI. ST-EQUI can outperform EQUI when one or more jobs determine that they are better off using fewer than their fair share of processors and release excess processors back to the system. On the other hand, EQUI can outperform ST-EQUI because ST-EQUI can be expected to reallocate processors much

more frequently than EQUI, thereby incurring much greater reallocation overhead. Furthermore, under ST-EQUI, all jobs incur the cost of self-tuning, even when all jobs want their fair share of processors. To better understand this new source of overhead, we next present the self-tuning procedure in somewhat more detail.

4.2 Self-Tuning

In this section, we present a brief overview of self-tuning. Comprehensive details can be found in [17], which examines the use of this technique in a static (essentially uniprogramming) environment.

Self-tuning is an online search technique that allows a parallel job to: (a) dynamically measure its efficiencies at different allocations, (b) use these measurements to estimate speedups, and (c) automatically adjust its allocation to maximize its speedup. Our current implementation of self-tuning employs a heuristic-based optimization technique that is an adaptation of *the method of golden sections* (MGS) [16] to find the best allocation. MGS is a simple optimization procedure that finds the maximum of a unimodal function over a finite interval by iteratively computing and comparing function values and narrowing the interval in which the maximum may occur⁴. In our case, the function to be maximized is job speedup. A job that is self-tuning computes the function value at p by running a single iteration using p processors, measuring the resulting efficiency, and calculating speedup using equation 1 (Section 3.1).

There are two basic problems we must address in using MGS for self-tuning. The first is that speedup functions are not, in general, unimodal. We address this using a simple, greedy heuristic. When the results of speedup evaluations in the current interval of interest demonstrate that speedup is not unimodal, we reduce the interval of interest to the largest subinterval that contains the currently known maximum speedup and for which the known speedup values are conformal with a unimodal function. While this heuristic does not guarantee that self-tuning will always find the global maximum, the experiments in [17] show that this procedure works remarkably well, almost always converging to a near optimal value.

The second problem we face is one of efficiency. Given an initial allocation of P processors, MGS normally starts searching within the the interval $[1, P]$. While MGS converges relatively quickly ($O(\log(P))$ steps are required), the cost of individual probes can be quite large if the job has poor speedup at the probed number of processors. We address this by exploiting the fact that, in our system, speedup

⁴For our purposes, a function $f(x)$ is unimodal over an interval $[a, b]$ if there is some $x^* \in [a, b]$ such that $f(x)$ is monotone non-decreasing in $[a, x^*]$ and monotone non-increasing in $[x^*, b]$

at P processors cannot be greater than P^5 . In particular, we begin self-tuning by executing one application iteration using all P processors available to the application. This allows us to estimate $S(P)$, the job’s speedup with P processors. Since speedups at numbers of processors less than $S(P)$ must be less than $S(P)$, we know that the globally best number of processors must fall in $[S(P), P]$. Our search therefore starts in this interval instead of $[1, P]$. For applications with good speedup, the interval $[S(P), P]$ will typically be small, allowing self-tuning to be performed with little overhead. For applications with poor speedup but only modest slowdown, speedup will be similar at all points between $[1, P]$ and so, again, self-tuning can be carried out with little overhead. Only in the case where an application initially achieves good speedup but then slows down significantly as its allocation grows does self-tuning incur significant overhead. In this case, for large P , $S(P)$ is likely to be small, resulting in a large initial search interval $[S(P), P]$. Furthermore, measuring speedup can be expensive at large (close to P) and small (close to 1) allocations.

4.3 Workload

As explained in Section 1, we are interested in a diverse workload composed of both hand-coded parallel (SPLASH) and compiler-parallelized sequential (PERFECT Club) applications.

Our previous detailed study of these applications [18] suggests that they can be divided into three broad classes:

- *Good speedup.* Most of the hand-coded applications fall into this class, which is characterized by fairly good speedup that mostly rises monotonically as the job receives more processors. Some of these applications exhibit modest slowdown beyond a certain number of allocated processors.
- *Poor speedup.* Almost all of the compiler-parallelized applications fall into this class, which is characterized by nearly negligible speedup at most processor values. Most of these applications exhibit significant slowdown beyond a certain number of allocated processors.
- *Erratic speedup.* This class consists of applications whose speedup is irregular, e.g., it varies over time or exhibits multiple local maxima. Such behavior can be observed in both hand-coded and compiler-parallelized applications [18].

Because it is infeasible to run experiments with all possible combinations from our benchmark suites, we instead

⁵This property holds because we estimate efficiency by measuring inefficiencies and subtracting them from 1.0.

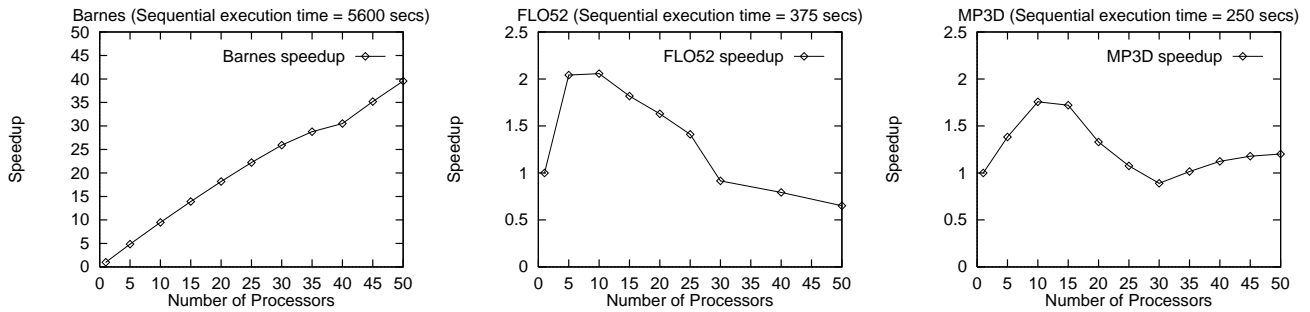


Figure 1: *Speedup Characteristics of the Representative Jobs*

use our taxonomy to reduce the number of jobs that must be considered, selecting a single representative application from each of the three classes. In particular, we chose the application exhibiting the best speedup from each class: Barnes from the SPLASH suite to exemplify good speedup, FLO52 from the PERFECT Club suite to exemplify poor speedup, and MP3D from the SPLASH suite to exemplify erratic speedup⁶. The measured speedup curves for these applications are given in Figure 1.

Next, we chose to set a maximum multiprogramming level of four, reasoning that (a) given our 50-processor machine, higher multiprogramming levels would increase processor demand to an extent that would render allocation decisions trivial, and (b) such a limit is prevalent in practice since memory constraints dictate that only a relatively small number of jobs can be allowed to run concurrently. This decision is supported by the measurements in [7], which indicate that multiprogramming levels of 2, 3, and 4 are the three most common during daytime hours in a production environment.

Note that in the work presented here, we do not address the question of which jobs should be activated when there are more jobs than the desired level of multiprogramming. Rather, we assume that some other mechanism, such as the feedback scheduling employed in sequential systems, is used for this purpose. (Parsons and Sevcik [19] present the design and evaluation of two such schemes, for example.) We consider the workload mix we schedule to be the subset of a larger job mix chosen for current execution by such a mechanism.

4.4 Implementation

At the user-level, we implement *process control* to avoid loss of efficiency due to mismatches between threads and processors [26]. One limitation of our quick conversion of the benchmark programs, though, is that we have implemented application level dynamic scheduling only at itera-

⁶We show in [17] that self-tuning is effective for a much larger number of applications than the 3 representative applications used in this study.

tion boundaries; the applications examine and adjust to the number of available processors each time they begin an iteration, but do not do so while executing any one iteration. It is clearly possible to do much more dynamic scheduling, e.g., [20, 1, 6, 14]; we did not do so because of the very large incremental implementation cost relative to our more restrictive change, and because we expect that ST-EQUI would perform even better when jobs are more responsive to changes in their allocations. (Of the three policies, ST-EQUI reallocates processors most frequently, and is therefore most sensitive to the latency with which applications can respond to changing allocations.)

We have implemented the code required to perform self-tuning in the CThreads library. Thus, the self-tuning procedure is independent of the specific application to be run. Additionally, there is little code development overhead involved in using it; we merely depend on the application to call into our library at the beginning of each iteration.

4.5 Performance

Given the three representative applications and a maximum multiprogramming level of four, we constructed and evaluated all 31 of the possible static workload mixes containing more than a single job type. Figures 2–4 depict the performance of 16 representative samples of the 31 workload mixes under the EQUI and ST-EQUI policies for multiprogramming levels 2, 3, and 4 respectively. Response times under these two policies are shown normalized to those under AP-EQUI (the horizontal line on each graph). These results lead us to the following observations:

- *The policy using runtime measurement (ST-EQUI) outperforms the similar policy that does not (EQUI).*

This effect arises because all jobs can benefit by participating in cooperative processor allocation. In scenarios with high demand for processors (e.g., all jobs request their equipartition share), ST-EQUI behaves exactly as does EQUI, so its performance is no worse. However, in scenarios with more complex processor

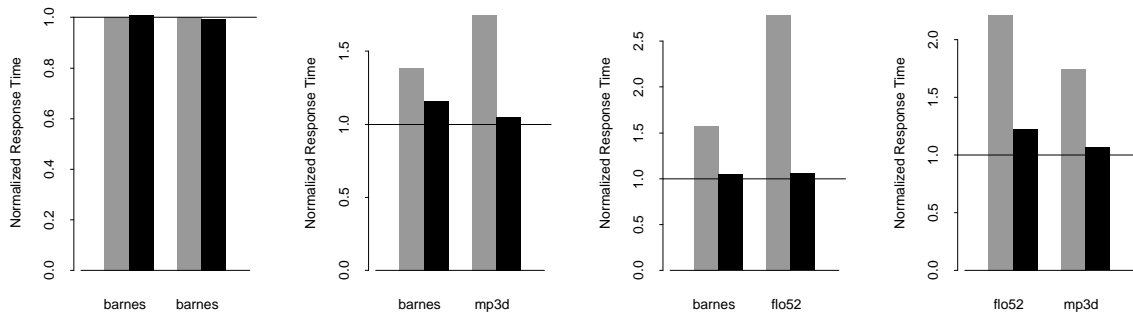


Figure 2: Response Time Results at Multiprogramming Level = 2. (Grey bars are results for EQU; black bars are results for ST-EQUI. Results are normalized with respect to AP-EQUI.)

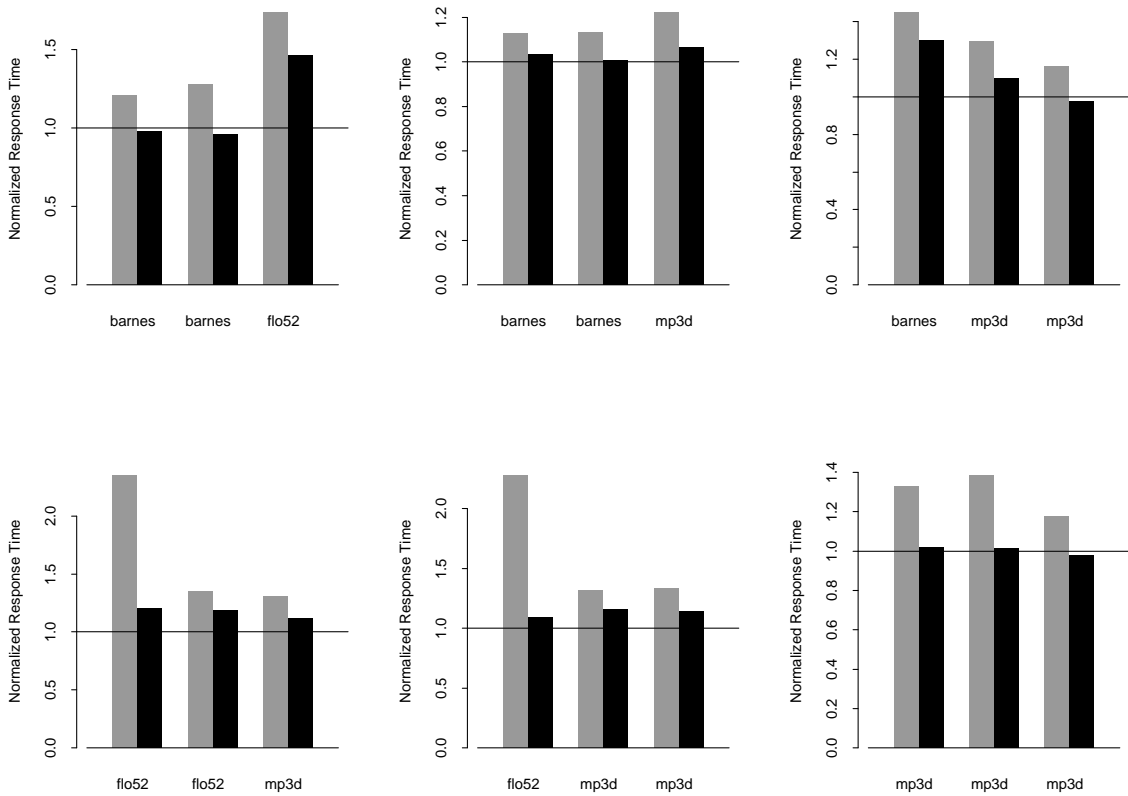


Figure 3: Response Time Results at Multiprogramming Level = 3. (Grey bars are results for EQU; black bars are results for ST-EQUI. Results are normalized with respect to AP-EQUI.)

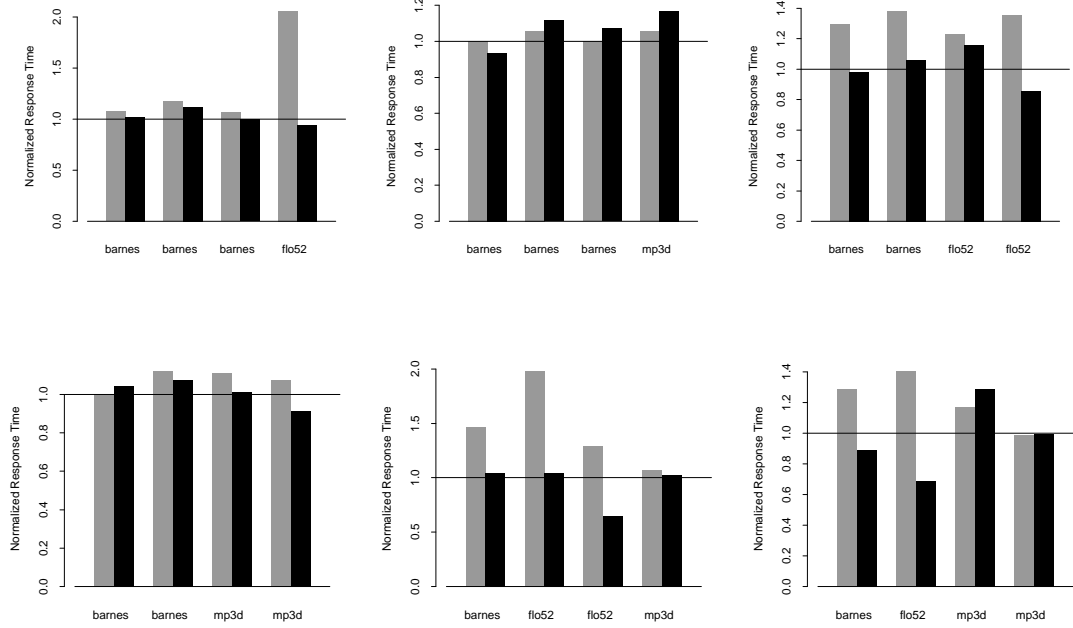


Figure 4: *Response Time Results at Multiprogramming Level = 4. (Grey bars are results for EQUI; black bars are results for ST-EQUI. Results are normalized with respect to AP-EQUI.)*

demands, ST-EQUI performs much better than does EQUI; jobs that start to slow down at allocations smaller than their EQUI-allocations run faster by shedding excess processors while jobs exhibiting good speedup gain these extra processors to run faster as well.

- *The policy using runtime measurements (ST-EQUI) performs nearly as well as the policy that uses a priori speedup information (AP-EQUI) in most cases.*

While there are some noticeable distinctions in the performance obtained by individual applications under ST-EQUI and AP-EQUI, in general, the two achieve performance that is roughly equivalent. This is especially true when one considers the workload mix as a whole: while the two policies discriminate among the individual jobs somewhat differently, frequently worse performance for one application is offset by better performance for another.

- *The performance distinctions between the policy that uses runtime measurements (ST-EQUI) and the policy that uses a priori information (AP-EQUI) result from costs associated with transient allocations.*

Detailed examination of the decisions made by the two policies shows that ST-EQUI converges to a set of allocations with no or insignificant performance

distinctions from those under AP-EQUI. This shows that, at least for the applications we examined, neither inaccuracies in the runtime measurements nor the inability of the search procedure to find appropriate allocations is a serious problem.

The primary cost of using runtime measurements in ST-EQUI relative to relying on perfect *a priori* information, as in AP-EQUI, is the penalty imposed by the sometimes poor allocations made during ST-EQUI's search. If the search procedure allocates too many processors to an application that exhibits significant slowdown, that application is affected directly by the resulting long self-tuning time. Additionally, other applications may suffer an opportunity cost, since they do not have access to the excess processors until the application determines that it was allocated too many.

4.6 Summary

We have shown that ST-EQUI, a policy that gathers and uses runtime information on application performance, clearly outperforms EQUI, a similar policy that does not make use of runtime information. ST-EQUI is fair in the sense that it does not discriminate among job classes; it simply responds to jobs' requests to release/acquire processors, giving each job equal weight in its attempt to minimize response time.

Because of this, ST-EQUI easily accommodates jobs that do not self-tune, since they receive allocations equivalent to those provided under EQUI.

In the next section, we explore schedulers that relax fairness in the interest of maximizing overall system efficiency for batch environments.

5 Batch Environments: Improving System Efficiency

In batch environments, such as are common for overnight runs of large parallel applications, the critical performance measure is not response time, but rather the rate at which useful work can be completed; the higher this rate, the larger the workload that can be processed in a fixed amount of time. In these environments, the goal of the scheduler is to maximize *system efficiency*, the sum of the efficiencies of all processors. In this section, we describe and evaluate a scheduling policy, EQUAL-EFF, that relaxes fairness in the interest of maximizing throughput.

5.1 The EQUAL-EFF Policy

The scheme we present below is motivated by consideration of how a scheduler designed to maximize system efficiency could be built in practice. It seems clear that any such allocation policy must reward applications exhibiting good efficiencies by allocating them many processors, and penalize those with bad efficiencies by allocating them only a few. An appealing structure for accomplishing this is to run all jobs for some quantum, measure their efficiencies, transfer processors from low efficiency to high efficiency applications, and then repeat the process. One advantage of this approach is its simplicity: it relies on only the recent efficiency measurements, which are easily and reliably obtainable, and so does not require knowledge of the full efficiency curves. Furthermore, we expect such a scheme to stabilize quickly to a set of reasonable allocations, and then to perform only a very modest rate of reallocations in the steady state.

Our early experiments with schedulers of this sort presented us with a set of problems that did not seem fundamentally difficult in the abstract, but which frustrated our attempts to build such a scheduler in practice. In particular, we needed a reliable way to decide whether or not the transfer of one or more processors from a job running with low efficiency to one running with high efficiency was merited. Furthermore, we needed a way to deal with the local irregularities found in real efficiency curves. While both these problems could be addressed by performing a thorough search of allocation choices, the transient poor allocations involved in

such searches can make them quite expensive.

Instead of performing thorough searches of the allocation space, we choose another approach, which we present as a first step in understanding how to solve these problems. First, to address the local irregularities in the efficiency curve, we use an artificial curve extrapolated from the most recently measured efficiency. In particular, having just measured the efficiency of an application on P processors to be ϵ , we use the function $Efficiency(p) = (1 + \beta)/(p + \beta)$ [5], choosing β so that the function evaluates to ϵ at $p = P$.

Next, we determine allocations by following an *equal efficiency* rule; that is, we allocate processors in a way that causes all applications to have about equal efficiencies according to our extrapolated curves⁷. In particular, we compute allocations in a simple, greedy way: we initially assign a single processor to each application, and then assign remaining processors one by one to the application with the currently highest (extrapolated) efficiency. The number of processors actually allocated to the jobs are then adjusted to match the newly computed allocation. We call this policy EQUAL-EFF.

In what follows, we present two sets of results. The first set is based on simulations, and is intended to show whether allocations based on equal-efficiency according to extrapolated efficiency curves come close to the goal of maximizing system efficiency. The second set of results is based on measurements of a prototype implementation, and so takes into account the inaccuracies inherent in the measurement process and the overheads involved in reallocating processors.

5.2 Evaluating the Inherent Effectiveness of EQUAL-EFF

Recall that our goal is to maximize system efficiency, but that we propose to do so by using extrapolated efficiency curves to determine allocations that result in nearly equal application efficiencies. In this subsection, we use simulations to determine whether such a procedure can come close to achieving our goal under the optimistic assumptions that measured efficiency information is perfectly reliable and that processor reallocation cost is negligible.

The inputs to the simulations are the measured application efficiency curves; the outputs are total system efficiencies. Given a set of jobs, our simulator begins by allocating available processors (nearly) equally. It then iteratively uses measured job efficiencies for the current allocation to derive extrapolated efficiency curves and follows EQUAL-EFF's

⁷Setting equal efficiency as a goal is a heuristic that is compatible with the scheduling structure described at the beginning of this subsection, a structure to which we plan to return. We address how well this heuristic does in achieving maximum system efficiency in the next subsection.

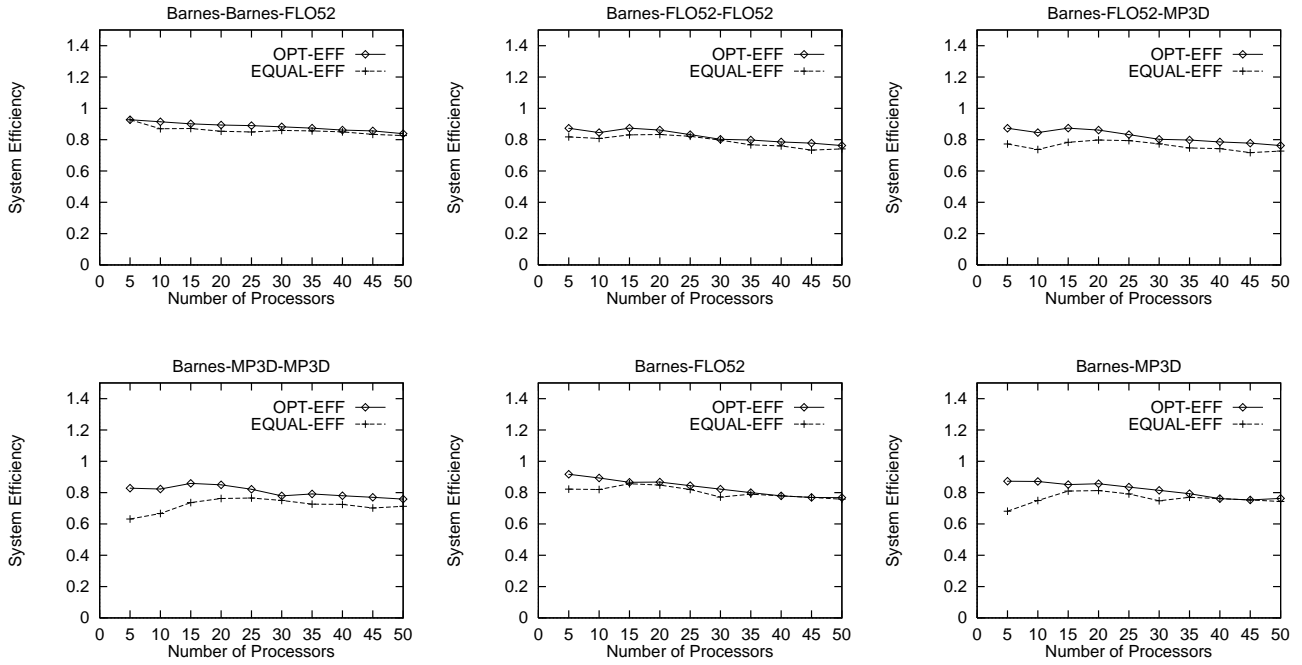


Figure 5: Modeled Results for EQUAL-EFF and OPT-EFF

allocation scheme to compute the next allocation. This process continues until successive allocations are identical. (It is not at all clear that this iterative process must always converge; however, in practice we did not encounter convergence problems.) This final allocation is used to compute system efficiency.

We compare the system efficiencies resulting from this process with the maximum possible system efficiencies. We compute the latter using a simple dynamic programming procedure that takes the full, measured efficiency curves as its input. We call the policy that makes these allocations OPT-EFF.

Figure 5 plots the system efficiencies obtained by these simulations against the total number of processors in the system for a representative set of workload mixes, as well as the optimal system efficiencies computed for OPT-EFF. In all cases, the greedy scheme employed by EQUAL-EFF comes very close to the optimum, with the largest differences occurring at very small numbers of processors. Furthermore, much of this difference is caused by local irregularities in actual efficiency curves. Results from simulations that use smooth theoretical speedup curves show even smaller performance differences.

Based on the results of this simple model, we were motivated to continue to the prototype implementation of EQUAL-EFF. The results obtained from experiments with that prototype are presented next.

5.3 Experimental Performance Results

We evaluate the use of runtime measured job characteristics in improving scheduling in a batch environment by considering four related policies:

- **EQUI.** The basic dynamic equipartition policy (Section 4.1).
- **EQUAL-EFF.** The equal efficiency heuristic policy (Section 5.1).
- **ST-EQUI.** The self-tuned equipartition algorithm, which was designed to reduce response times (Section 4.1).
- **ST-EQUAL-EFF.** The EQUAL-EFF policy with the addition that each job also engages in self-tuning, releasing processors when it determines that it has been assigned more than it can profitably use.

The performance results under these policies are compared against those predicted by OPT-EFF. Because OPT-EFF is computed off-line, it does not capture any of the overheads that are inevitable in scheduling in practice, exaggerating its optimism.

We assessed performance with workloads composed of the same representative jobs as were used in Section 4. However, we used a single multiprogramming level of 3 in all experiments (a reduction from the maximum of 4 considered

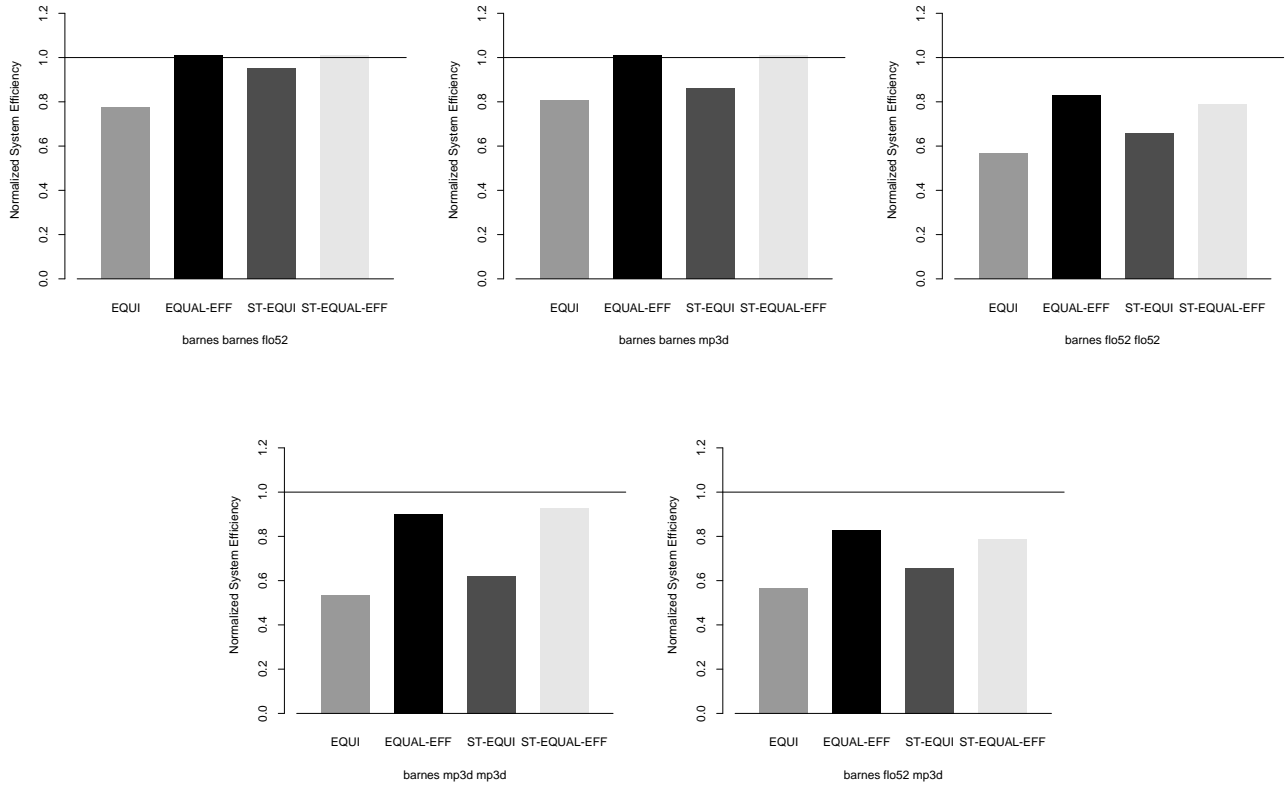


Figure 6: System Efficiency Results (Multiprogramming level = 3; results are normalized with respect to OPT-EFF).

Load	Job	EQUI	EQUAL-EFF	ST-EQUI	ST-EQUAL-EFF
Barnes-Barnes-FLO52	Barnes	0.32	0.44	0.43	0.44
	FLO52	0.05	0.22	0.31	0.23
Barnes-Barnes-MP3D	Barnes	0.35	0.44	0.37	0.44
	MP3D	0.30	0.22	0.32	0.21
Barnes-FLO52-FLO52	Barnes	0.21	0.31	0.24	0.29
	FLO52	0.30	0.41	0.53	0.46
Barnes-MP3D-MP3D	Barnes	0.18	0.28	0.23	0.36
	MP3D	0.55	0.44	0.63	0.36
Barnes-FLO52-MP3D	Barnes	0.23	0.31	0.23	0.29
	FLO52	0.14	0.20	0.27	0.23
	MP3D	0.31	0.23	0.33	0.26

Table 1: Job Throughput Rates (jobs/ minute).

for the interactive environment) to reflect the likely larger size of jobs submitted for batch execution. (This change is supported by the measurements in [7].) Additionally, we present here results only for those workloads that include a Barnes job, the representative from the class of jobs having good speedup. Workloads without Barnes are relatively uninteresting, as there are 50/3 processors available to each job under simple equipartition, and this number exceeds the number that can be used profitably by the other two representative jobs in our mixes. For this reason, as well as space limitations, we omit these results in what follows.

Figure 6 presents the experimental results we obtained. From them, we draw conclusions similar to those in Section 4.5:

- *Policies using runtime measurements can greatly outperform those without access to such information.*
This is supported by comparing the results for EQUAL-EFF, ST-EQUI, and ST-EQUAL-EFF scheduling to those for EQUI.
- *Equal-efficiency-based policies outperform Equipartition-based policies.*
This is supported by comparing the results for EQUAL-EFF and ST-EQUAL-EFF to those for ST-EQUI.
- *Policies using runtime measurements can approach the performance of policies with access to a priori efficiency information.*
For all workloads, the performance of the equal efficiency policies is within 20% of those for the overly optimistic OPT-EFF.
- *The primary policy-induced loss of efficiency is the cost associated with the allocation search procedure.*
In the cases where EQUAL-EFF and ST-EQUAL-EFF fail to approach closely the (theoretical) optimal performance of OPT-EFF, further examination revealed that it was because of the overhead associated with our search procedure, rather than because the search was settling on poor final allocation choices.

Because the EQUAL-EFF policy attempts to maximize throughput without regard to fairness, it is natural to wonder if jobs with poor speedup characteristics are starved under this discipline. Table 1 shows the job throughput rates (in jobs/minute) for each job class under our test workload mixes. If starvation were a problem in practice, we would expect to see sharp drops in throughput for FLO52 and MP3D when comparing an equal efficiency policy to an equipartition policy. The fact that this does not happen is a reflection of the equal efficiency policies' guarantee that every job be given at least one processor.

6 Conclusions

Our goal in this paper was to determine if parallel processor allocation policies could beneficially exploit runtime measurements of application performance. If so, such runtime characterization could replace a reliance on *a priori* specification of job characteristics, a time-consuming and possibly error-prone task, or could supplement the use of *a priori* information when available.

For a number of reasons, it was not obvious whether runtime measurements would be useful to parallel schedulers. To explore these issues, we have formulated policies that make use of runtime measurements for both interactive and batch oriented environments. Our experimental results show that schedulers that use runtime measurements of workload characteristics can significantly outperform schedulers that are oblivious to these characteristics. Furthermore, these schedulers can achieve performance surprisingly close to that possible when efficiency and speedup information is available *a priori*. Given the convenience of these policies for the users of the system, their resilience to changes in program behavior due to phase changes within a single run or to changes in datasets between runs, their good performance, and the evidence from our prototypes that practical implementations are possible, we believe that the availability of runtime measurements is an important factor to be considered in designing parallel processor allocation policies.

Acknowledgments

Mary Vernon provided insightful comments that helped with both the content and presentation of this work.

References

- [1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, Feb. 1992.
- [2] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Scharzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.
- [3] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies.

- In *Proceedings of the ACM SIGMETRICS Conference*, pages 33–44, May 1994.
- [4] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.
 - [5] L. Dowdy. On the Partitioning of Multiprocessor Systems. Technical report, Vanderbilt University, June 1988.
 - [6] D. L. Eager and J. Zahorjan. Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing. *ACM Transactions on Computer Systems*, 11(1):1–32, Feb. 1993.
 - [7] D. G. Feitelson and B. Nitzberg. Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860. In *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 337–360, Apr. 1995.
 - [8] D. G. Feitelson and L. Rudolph. Coscheduling Based on Runtime Identification of Activity Working Sets. *International Journal of Parallel Programming*, 23(2):135–160, Apr. 1995.
 - [9] D. Ghosal, G. Serazzi, and S. Tripathi. The Processor Working Set and Its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.
 - [10] K. Guha. Using Parallel Program Characteristics in Dynamic Processor Allocation Policies. Technical Report CS-95-03, Department of Computer Science, York University, May 1995.
 - [11] Kendall Square Research Inc., 170 Tracer Lane, Waltham, MA 02154. *KSR/Series Principles of Operation*, 1994.
 - [12] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 226–236, May 1990.
 - [13] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of the ACM SIGMETRICS Conference*, pages 104–113, May 1988.
 - [14] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, Apr. 1994.
 - [15] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.
 - [16] G. P. McCormick. *Nonlinear Programming*. John Wiley & Sons, Inc., 1983.
 - [17] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup Through Self-Tuning of Processor Allocation. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 463–468, Apr. 1996.
 - [18] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Parallel Application Characterization for Multiprocessor Scheduling Policy Design. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
 - [19] E. W. Parsons and K. C. Sevcik. Multiprocessor Scheduling for High-Variability Service Time Distribution. In *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 127–145, Apr. 1995.
 - [20] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, Dec. 1987.
 - [21] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust Partitioning Policies of Multiprocessor Systems. *Performance Evaluation*, 19:141–165, 1994.
 - [22] K. C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the ACM SIGMETRICS Conference*, pages 171–180, May 1989.
 - [23] K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. *Performance Evaluation*, 19(2/3):107–140, Mar. 1994.
 - [24] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.
 - [25] P. B. Sobalvarro and W. E. Weihl. Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors. In *Proceedings of the IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 106–126, Apr. 1995.
 - [26] A. Tucker and A. 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, Dec. 1989.