RC 21569 (97280) 01OCT99 Computer Science/Mathematics

Research Report

Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques

Yanyong Zhang
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802

Hubertus Franke, José E. Moreira
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Anand Sivasubramaniam
Department of Computer Science & Engineering
The Pennsylvania State University
University Park, PA 16802

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

Improving Parallel Job Scheduling by Combining Gang Scheduling and Backfilling Techniques

Y. Zhang† H. Franke‡ J. E. Moreira‡ A. Sivasubramaniam†

† Department of Computer Science & Engineering
The Pennsylvania State University
University Park PA 16802
{yyzhang, anand}@cse.psu.edu

‡ IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights NY 10598-0218
{frankeh, jmoreira}@us.ibm.com

Abstract

Effective scheduling strategies to improve response times, throughput, and utilization are an important consideration in large supercomputing environments. Such machines have traditionally used space-sharing strategies to accommodate multiple jobs at the same time. This approach, however, can result in low system utilization and large job wait times. There have been two recent approaches to address this problem: (a) augmenting space sharing with backfilling, which performs out of order job scheduling; and (b) augmenting space sharing with time sharing, using a technique called coscheduling or gang scheduling. Logically, it appears that one could combine these two approaches into a strategy that uses backfilling with gang scheduling. However, this has not been considered as a viable option until now, because of the difficulty in estimating job execution time, as required for backfilling. With three important experimental results – impact of priority queue order on backfilling, impact of overestimation of job execution times, and comparison of scheduling techniques – this paper presents an integrated strategy that combines backfilling with gang scheduling. Using extensive simulations based on detailed models of realistic workloads, the benefits of combining backfilling and gang scheduling are clearly demonstrated over a spectrum of performance criteria.

1 Introduction

Large scale parallel machines are essential to meet the needs of demanding applications at supercomputing environments such as Lawrence Livermore (LLNL), Los Alamos (LANL) and Sandia National Laboratories. With the increasing emphasis on computer simulation as an engineering and scientific tool, the load on such systems is expected to become quite high in the near future. As a result, it is imperative to provide effective scheduling strategies to meet the desired quality of service parameters from both user and system perspectives. Specifically, we would like to reduce response and wait times for a job, minimize the slowdown that a job experiences in a multiprogrammed setting compared to when it is run in isolation, maximize the throughput and utilization of the system, and be fair to all jobs regardless of their size or execution times.

Scheduling strategies can have a significant impact on the performance characteristics of a large parallel system [2, 3, 6, 9, 12, 13, 17, 18, 21]. Early strategies just used a space-sharing approach, wherein jobs can run side by side on different nodes of the machine at the same time, but each node is exclusively assigned to a job. Space sharing in isolation can result in poor utilization since there could be nodes that are unutilized despite a waiting queue of jobs. Furthermore, the wait and response times for jobs with an exclusively space-sharing strategy are relatively high.

Among the several approaches used to alleviate these problems with space sharing scheduling, two have been most commonly studied. The first is a technique called backfilling which attempts to assign unutilized nodes to jobs that are behind in the priority queue (of waiting jobs), rather than keep them idle. To prevent starvation for larger jobs, (conservative) backfilling requires that a job selected out of order completes before the jobs that are ahead of it in the priority queue are scheduled to start. This requires the user to provide an estimation of job execution times, in addition to the number of nodes that it requires. The second approach is to add a time-sharing dimension to space sharing using a technique called gang scheduling or coscheduling [16]. This technique virtualizes the physical machine by slicing the time axis into multiple space-shared virtual machines. Tasks of a parallel job are coscheduled to run in the same time-slices (same virtual machines). In some cases it may be advantages to schedule the same job to run on multiple virtual machines (multiple time-slices). The number of virtual machines created (equal to the number of time slices), is called the multiprogramming level (MPL) of the system. This multiprogramming level in general depends on how many jobs can be executed concurrently, but is typically limited by system resources. This approach opens more opportunities for the execution of parallel jobs, and is thus quite effective in reducing the wait time, at the expense of increasing the apparent job execution time. Gang scheduling does not depend on estimates for job execution time. Gang scheduling has been used in the prototype GangLL job scheduling system developed by IBM Research for the ASCI Blue-Pacific machine at Lawrence Livermore National Laboratory (a large scale parallel system spanning thousands of nodes [15]),

It is a logical next step to attempt to combine these two approaches – gang scheduling and backfilling. In principle, combining backfilling and gang scheduling is as simple as applying backfilling to each of the virtual machines created by gang scheduling. However, obtaining precise estimates for job execution time under gang scheduling can be very difficult or even impossible. The effective multiprogramming level of a parallel system can actually vary during the execution of a job. Therefore, even if exact information on the dedicated execution time were available, in general it would not be possible to estimate the execution time when the machine is time-shared.

Progressing to this combined approach requires a careful examination of several issues related to both backfilling and gang scheduling. This paper addresses the following four important issues which help us make this transition:

1. What is the impact of the priority queueing discipline on backfilling? Is one discipline clearly better than the others in terms of response times, slowdown and utilization? Can we have a discipline that is not only good in terms of these metrics, but is also fair to all jobs?

2. What is the impact of overestimating job execution times on the effectiveness of backfilling? Is it important to accurately estimate job execution times?

3. How do we estimate job completion times in a gang scheduled system?

4. What is the overall impact of combining backfilling and gang scheduling on quality of service and system performance parameters?

Using detailed simulations based on stochastic models derived from real workloads at LLNL, this paper provides answers to these questions. We find that FCFS queueing policy does as well as other priority policies and has the advantage of being fair to all jobs, regardless of their size or execution times. Overestimating job execution times does not really impact the quality of service parameters, regardless of the degree of overestimation. As a result, we can conservatively estimate the execution time of a job in a coscheduled system to be the multiprogramming level (MPL) times the estimated job execution time in a dedicated setting.

These results help us construct a backfilling gang scheduling system, called BGS, which fills in holes in the Ousterhout scheduling matrix [16] with jobs that are not necessarily in FCFS order. It is clearly demonstrated that this combined strategy is always better than the individual gang scheduling or backfilling strategies for all the quality of service parameters that we consider.

The rest of this paper is organized as follows. Section 2 describes our approach to modeling parallel job workloads and obtaining performance characteristics of scheduling systems. It also characterizes our base workload quantitatively. Section 3 is a study of the impact of backfilling on different job queuing policies. It shows that a FCFS priority policy together with backfilling is a sensible choice. Section 4 analyzes the impact of job execution time estimation on the overall performance from system and user perspectives. We show that relevant performance parameters are almost invariant to the accuracy of average job execution time estimation. Section 5 demonstrates the significant improvements in performance that can be achieved by combining gang scheduling and backfilling with a FCFS policy. Finally, Section 6 presents our conclusions and possible directions for future work.

2 Evaluation methodology

Before we present the results from our studies we first need to describe our methodology. In this section, we begin by describing how we generate synthetic workloads (drawn from realistic environments) that drive our simulator. We then present the particular characteristics of the workloads we use. Finally, we discuss the performance metrics we adopt to measure the quality of service in a parallel system.

When selecting and developing job schedulers for use in large parallel system installations, it is important to understand their expected performance. The first stage is to have a characterization and a procedure to synthetically generate the expected workloads. Our methodology for generating these workloads, and from there obtaining performance parameters, involves the following steps:

1. Fit a typical workload with mathematical models.
2. Generate synthetic workloads based on the derived mathematical models.
3. Simulate the behavior of the different scheduling policies for those workloads.
4. Determine the parameters of interest for the different scheduling policies.

2.1 Workload modeling

When fitting a parallel load, it is very useful to be able to find a compact mathematical representation that is expressible by a few parameters, is reasonably easy to use for the generation of synthetic workloads, and is also suitable for theoretical queuing analysis of scheduling algorithms.

Parallel workloads often are over-dispersive. That is, job interarrival time distribution and job service time (execution time on a dedicated system) distribution each has a coefficient of variation that is greater than one. Distributions with coefficient of variation greater than one are also referred to as long-tailed distributions, and can be fitted adequately with Hyper Erlang Distributions of Common Order. In [11] such a model was developed, and its efficacy demonstrated by using it to fit a typical workload from the Cornell University Theory Center. Here we use this model to fit a typical workload from the ASCI Blue-Pacific System.

Our modeling procedure involves the following steps:

1. First we group the jobs into classes, based on the number of processors they require to execute on. Each class is a bin in which the upper boundary is a power of 2.

2. Then we model the interarrival time distribution for each class, and the service time distribution for each class as follows:

   (a) From the job traces, we compute the first three moments of the observed interarrival time and the first three moments of the observed service time.

   (b) Then we select the Hyper Erlang Distribution of Common Order that fits these 3 observed moments. We choose to fit the moments of the model against those of the actual data because the first 3 moments usually capture the generic features of the workload and are more robust to the effect of outliers. These three moments carry the information on the mean, variance, and skewness of the random variable respectively.

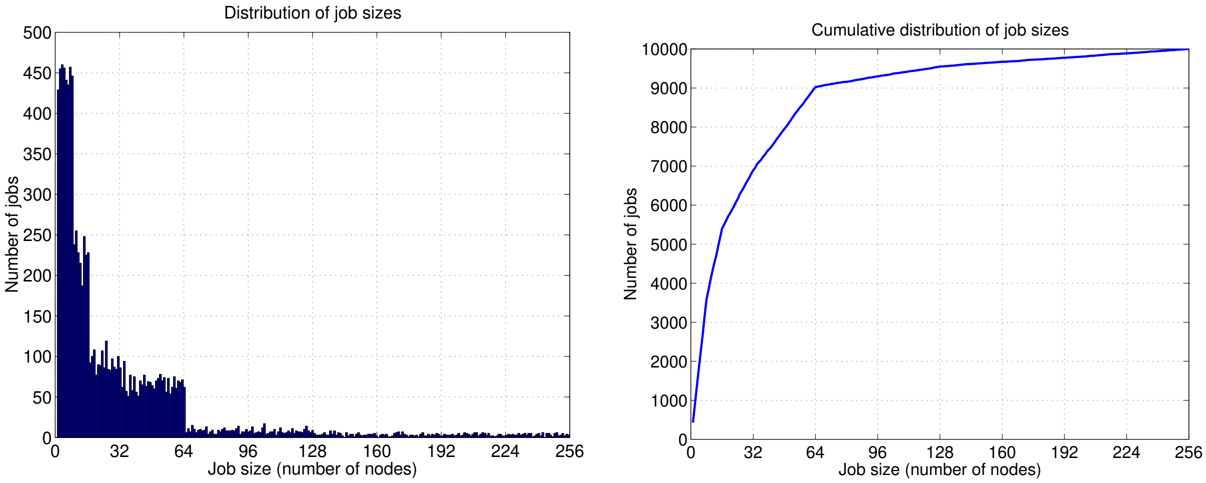
Next we generate various synthetic workloads from the observed workload by varying the interarrival rate and service time used. The Hyper Erlang parameters for these synthetic workloads are obtained by multiplying the interarrival rate and the service time each by a separate multiplicative factor, and by specifying the number of jobs to generate. From these model parameters synthetic job traces are obtained using the procedure described in [11]. Finally, we simulate the effects of these synthetic workloads and observe the results.

Within a workload trace, each job is described by its arrival time, the number of nodes it uses, its execution time on a dedicated system, and an overestimation factor. Backfilling strategies require an estimate of the job execution time. We obtain this estimate by multiplying the dedicated execution time by an overestimation factor, which is a uniformly distributed random number between 1 and an upper limit $1 + \Omega$. In particular, $\Omega = 0$ indicates we have perfect knowledge of how long jobs are going to run. In reality, users provide an estimated execution time for their job. This estimated execution time is always greater than or equal to the actual execution time, since jobs are terminated after reaching this limit. The overestimation factor is the mechanism we use to capture this discrepancy between estimated and actual execution times for parallel jobs. During simulation, the estimated execution time is used exclusively for performing job scheduling, while the actual execution time is only used to define the job finish event.

2.2 Workload characteristics

The baseline workload is the synthetic workload generated from the parameters directly extracted from the actual ASCI Blue-Pacific workload. It consists of 10000 jobs, varying in size from 1 to

256 nodes, in a system with a total of 320 nodes. Some characteristics of this workload are shown in Figures 1 and 2. Figure 1 reports the distribution of job sizes (number of nodes). For each job size, between 1 and 256, Figure 1(a) shows the number of jobs of that size, while Figure 1(b) plots the number of jobs with *at most* that size. (In other words, Figure 1(b) is the integral of Figure 1(a).) Figure 2 reports the distribution of total CPU time, defined as job execution time on a dedicated setting times its number of nodes. For each job size, Figure 2(a) shows the sum of the CPU times for all jobs of that size, while Figure 2(b) is a plot of the sum of the CPU times for all jobs of *at most* that size. (In other words, Figure 2(b) is the integral of Figure 2(a).) From Figures 1 and 2 we observe that, although large jobs (those with more than 32 nodes), represent only 30% of the number of jobs, they constitute more than 80% of the total work performed in the system. This baseline workload corresponds to a system utilization of $\rho = 0.55$. (System utilization is defined in Section 2.3.)



(a) (b)

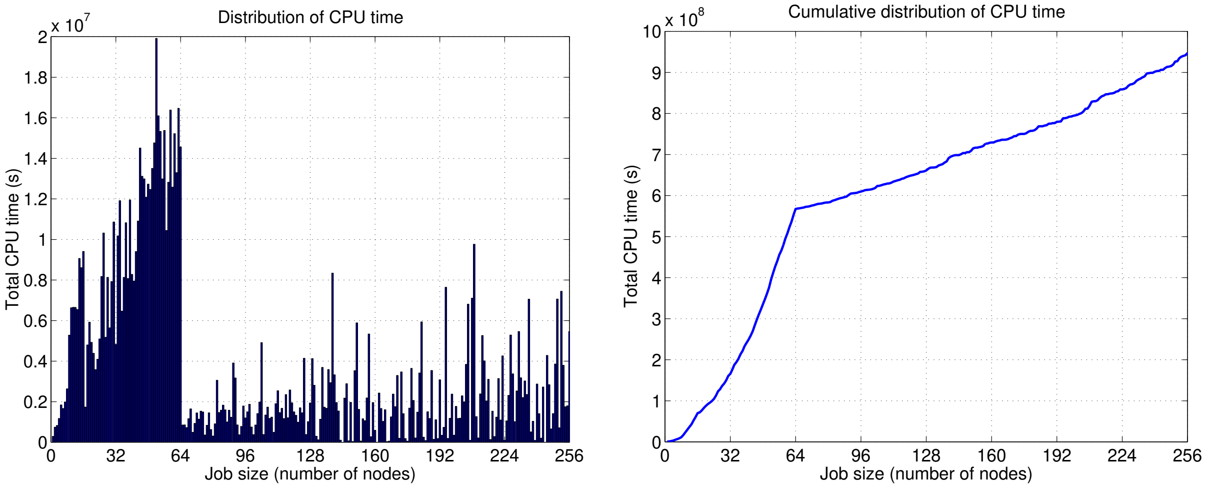
Figure 1: Workload characteristics: distribution of job sizes.

In addition to the baseline workload of Figures 1 and 2 we generate 8 additional workloads, of 10000 jobs each, by varying the model parameters so as to increase average job execution time. More specifically, we generate the 9 different workloads by multiplying the average job execution time by a factor from 1.0 to 1.8 in steps of 0.1. For a fixed interarrival time, increasing job execution time typically increases utilization, until the system saturates.

2.3 Performance metrics

The synthetic workloads generated as described in Section 2.1 are used as input to our event-driven simulator of various scheduling strategies. In our simulation, we monitor the following parameters:

- t_i^a: arrival time for job i.
- t_i^s: start time for job i.
- t_i^e: execution time for job i (in a dedicated setting).



(a) (b)

Figure 2: Workload characteristics: distribution of cpu time.

- t_i^f: finish time for job i.
- n_i: number of nodes used by job i.

From these we compute:

- $t_i^r = t_i^f - t_i^a$: response time for job i.
- $t_i^w = t_i^s - t_i^a$: wait time for job i.
- $s_i = \frac{\max(t_i^r, 10)}{\max(t_i^e, 10)}$: the slowdown for job i. To reduce the statistical impact of very short jobs, it is common practice [4, 5] to adopt a minimum execution time of 10 seconds. This is the reason for the $\max(\cdot, 10)$ terms in the definition of slowdown.

To report quality of service figures from a user's perspective we use the average job slowdown and average job wait time. Job slowdown measures how much slower than a dedicated machine the system appears to the users, which is relevant to both interactive and batch jobs. Job wait time measures how long a job takes to start execution and therefore it is an important measure for interactive jobs. In addition to objective measures of quality of service, we also use these averages to characterize the fairness of a scheduling strategy. We evaluate fairness by comparing average and standard deviation of slowdown and wait time for small jobs, large jobs, and all jobs combined. As discussed in Section 2.2 large jobs are those that use more than 32 nodes, while small jobs use 32 or fewer nodes.

We measure quality of service from the system's perspective with two parameters: utilization and capacity loss. Utilization is the fraction of total system resources that are actually used during the execution of a workload. Let the system have N nodes and execute m jobs, where job m is the last job to finish execution. Also, let the first job arrive at time $t = 0$. Utilization is then defined as

$$\rho = \frac{\sum_{i=1}^{m} n_i t_i^e}{N \times t_m^f \times \text{MPL}} \tag{1}$$

A system incurs loss of capacity when (i) it has jobs waiting in the queue to execute, and (ii) it has empty nodes (either physical or virtual) but, because of fragmentation, it still cannot execute those waiting jobs. Before we can define loss of capacity, we need to introduce some more concepts. A *scheduling event* takes place whenever a new job arrives or an executing job terminates. By definition, there are $2m$ scheduling events, occurring at times ψ_i, for $i = 1, \ldots, 2m$. Let e_i be the number of nodes left empty between scheduling events i and $i+1$. Finally, let δ_i be 1 if there are any jobs waiting in the queue after scheduling event i, and 0 otherwise. Loss of capacity is then defined as

$$\kappa = \frac{\sum_{i=1}^{2m-1} e_i(\psi_{i+1} - \psi_i)\delta_i}{\mathrm{MPL} \times t_m^f \times N} \tag{2}$$

A system is in a saturated state when increasing the load does not result in an increase in utilization. At this point, the loss of capacity is equal to one minus the maximum achievable utilization. More specifically, $\kappa = 1 - \rho$.

3 Queuing policies with backfilling

In this section we analyze the behavior of different well known scheduling policies when backfilling is used. As previously mentioned, a scheduling policy is a set of rules that prioritizes the order with which jobs are selected for execution. We consider four different scheduling policies:

1. First come first serve (FCFS): Jobs are ordered according to their arrival time. Jobs that arrived earlier have higher priority over jobs that arrived later.

2. Shortest job first (SJF): Jobs are ordered according to their estimated execution time. Shorter running jobs have higher priority over longer running jobs. Note that this policy can lead to starvation of long running jobs.

3. Best fit (BFit): Jobs are ordered according to their size (number of nodes). The scheduler looks for the job that best matches the number of empty nodes. That is, the goal is to minimize the size of the fragment left after each scheduling event. Note that the queue of jobs typically has to be reordered after each scheduling event.

4. Worst fit (WFit): Jobs are ordered according to their size, and scheduling proceeds from the smallest to the largest job. The goal is to fill the fragments with small jobs. Note that this policy can lead to starvation of large jobs.

Backfilling is a space-sharing optimization technique that can be used with any of the above policies. Using one of the policies above, the scheduler can build a schedule for all jobs in the waiting queue. This schedule will determine a specific start time t_i^s for each job i. With backfilling, we can bypass the priority order imposed by the policy. This allows a lower priority job j to be scheduled before a higher priority job i as long as this reschedule does not incur a delay on the start time of job i for that particular schedule. This requirement of not delaying higher priority jobs is exactly what imposes the need for an estimate of job execution times. The effect of backfilling on a particular schedule can be visualized in Figure 3. Suppose we have to schedule five jobs, numbered from 1 to 5 in order of arrival. Figure 3(a) shows the schedule that would be produced by a FCFS policy without backfilling. Note the empty space between times T_1 and T_2, while job 3 waits for job 2 to finish. Figure 3(b) shows the schedule that would be produced by a FCFS policy with backfilling. The empty space was filled with job 5, which can be executed before job 3 without delaying it.

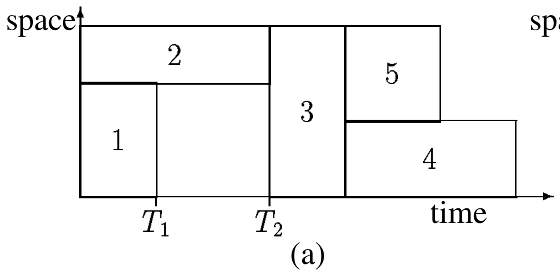
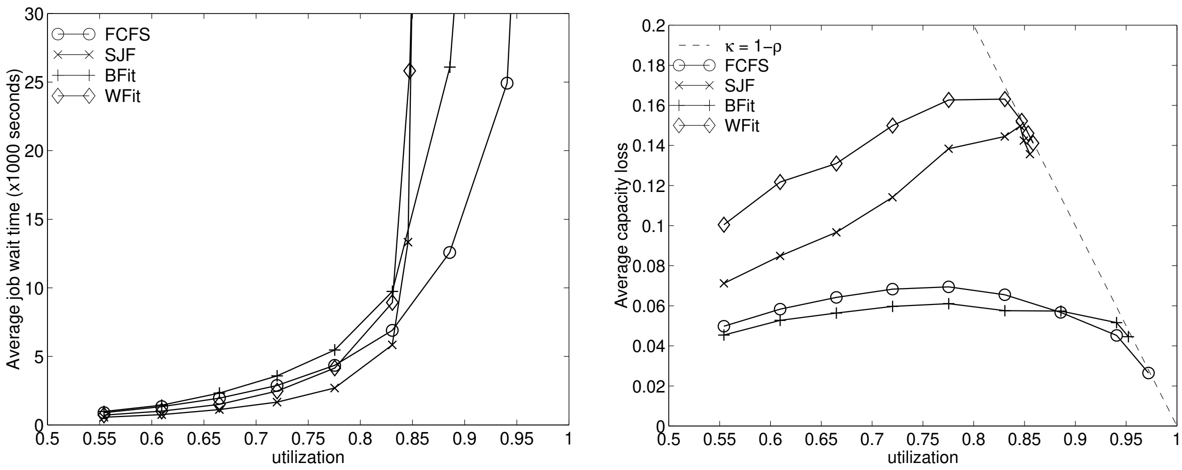


Figure 3: FCFS policy without (a) and with (b) backfilling. Job numbers correspond to their position in the priority queue.

Figure 4 summarizes results of average job wait time and loss of capacity for each of the four policies we discussed above in the presence of backfilling. For these policies, wait time is a particularly good indication of the quality of service from a user's perspective. Once a job is done waiting and starts executing, the execution proceeds as in a dedicated machine. Therefore, wait time captures the essential performance characteristics. For the purpose of providing a performance reference, we assume perfect knowledge of job execution times ($\Omega = 0$) when performing scheduling. We refer to this as *perfect backfilling*. The dashed line in Figure 4(b) is a plot of $\kappa = 1 - \rho$ and, as discussed in Section 2.3, represents the loci of maximum utilization (saturation) points.



(a) average wait time (b) average capacity loss

Figure 4: Comparing different policies in BF.

From Figure 4(a), we observe that at lower utilization (up to 80%) all policies are comparable, with SJF leading the pack. However, at higher utilizations FCFS performs better than the other policies. From Figure 4(b), we observe that both SJF and WFit saturate at an utilization of 87%, while both BFit and FCFS can sustain utilizations of 95%. However, at this high loads, FCFS exhibits better average job wait time than BFit. On top of that, FCFS is straightforward to implement and provides better fairness, since it has no implied starvation problems. In face of these results, and in order to limit the length of this paper, we restrict our discussion to FCFS policies for the remaining of the paper.

4 The impact of overestimation on backfilling

A common perception with backfilling is that one needs a fairly accurate estimation of job execution time to perform good backfilling scheduling. Users typically provide an estimate of job execution time when jobs are submitted. However, it has been shown in the literature [5] that there is little correlation between estimated and actual execution times. Since jobs are killed when the estimated time is reached, users have an incentive to overestimate the execution time. This is indeed a major impediment to applying backfilling to gang scheduling. The effective rate at which a job executes under gang scheduling depends on many factors, including: (i) what is the effective multiprogramming level of the system, (ii) what other jobs are present, and (iii) how many time slices are occupied by the particular job. This makes it even more difficult to estimate the correct execution time for a job under gang scheduling.

We conducted a study of the effect of overestimation on the performance of backfilling schedulers using a FCFS prioritization policy. The results are summarized in Figure 5. Figures 5(a) and 5(b) plot average job slow down and average job wait time, respectively, as a function of system utilization for different values of Ω. We can see that the impact of overestimation is minimal with respect to the average behavior of user jobs. We observe that for utilizations of up to $\rho = 0.90$ overestimation actually helps in reducing average slow down in approximately 20% with respect to perfect backfilling. The variation in average wait time for utilizations up to $\rho = 0.85$ is negligible. Only at very high utilizations we start to see some impact of overestimation.

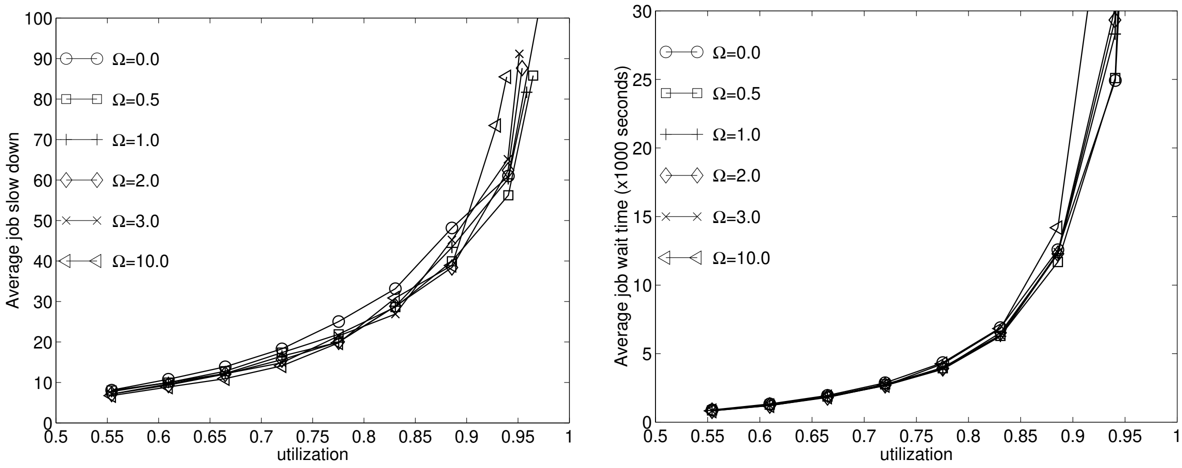
We can understand why backfilling is not that sensitive to the estimated execution time by the following reasoning:

1. When the load is low, the estimation does not really matter, since backfilling is not really performed that often. There are not that many jobs waiting, as indicated by the low waiting time.

2. Backfilling has more effect when the load is higher. On average, overestimation impacts both the jobs that are running and the jobs that are waiting. The scheduler computes a later finish time for the running jobs, creating larger holes in the schedule. The larger holes can then be used to accommodate waiting jobs that have overestimated execution times. The probability of finding a backfilling candidate effectively does not change with the overestimation.

An example that demonstrates how overestimation can actually help in improving overall performance is shown in Appendix A.

Even though the average job behavior is insensitive to the average degree of overestimation, individual jobs can be affected. To verify that, we group the jobs into 10 classes based on how



(a) average slowdown (b) average wait time

Figure 5: The impact of overestimation on BF.

close is their estimated time to their actual execution time. More precisely, class i, $i = 0, \ldots, 9$ includes all those jobs for which their ratio of estimated to actual execution time falls in the range $[1 + \frac{\Omega}{10}i, 1 + \frac{\Omega}{10}(i + 1))$. Figure 6 shows the average job wait time for (i) all jobs, (ii) jobs in class 0 (best estimators) and (iii) jobs in class 9 (worst estimators) when the average overestimation factor is 3 ($\Omega = 3$). We observe that those users that provide good estimates are rewarded with a lower average wait time. The conclusion is that the "quality" of an estimation is not really defined by how close it is to the actual execution time, but by how much better it is compared to the average estimation. Users do get a benefit, and therefore an encouragement, from providing good estimates.

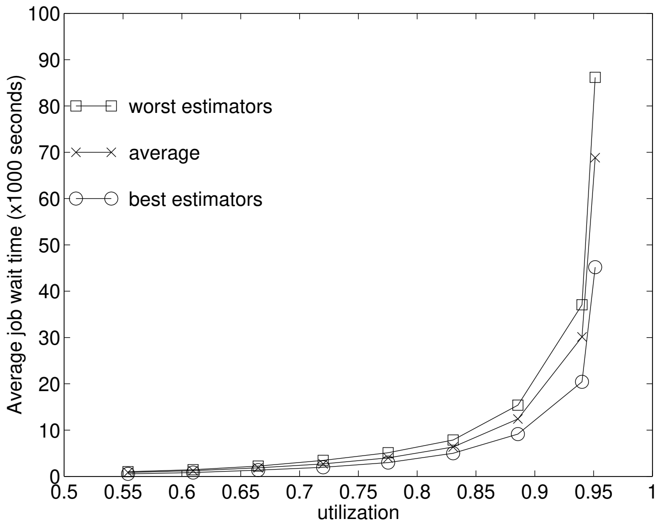


Figure 6: The impact of good estimation from a user perspective.

Our findings are in agreement with the work described in [19]. In that paper, the authors describe mechanisms to more accurately predict job execution times, based on historical data. They find that more accurate estimates of job execution time leads to more accurate estimates of wait time. However, the accuracy of execution time prediction has minimal effect on system parameters, such as utilization. The authors do observe an improvement in average job wait time, for a particular Argonne National Laboratory workload, when using their predictors instead of previously published work [1, 8].

5 Backfilling gang scheduling

In the previous sections we only considered space-sharing scheduling strategies. An extra degree of flexibility in scheduling parallel jobs is to share the machine resources not only spatially but also temporally by partitioning the time axis into multiple time slices [2, 3, 7, 10, 20]. As an example, time-sharing an 8-processor system with a multiprogramming level of four is shown in Figure 7. The figure shows the scheduling matrix that defines the tasks executing on each processor and each time-slice. J_i^j represents the j-th task of job J_i. The matrix is cyclic in that time-slice 3 is followed by time-slice 0. Each row of the matrix defines an 8-processor virtual machine, which runs at $1/4$th of the speed of the physical machine. We use these four virtual machines to run two 8-way parallel jobs and several smaller jobs. All tasks of a parallel job are always coscheduled to run concurrently. This approach gives each job the impression that it is still running on a dedicated, albeit slower, machine. This type of scheduling is commonly called *gang scheduling* [2]. Note that some tasks can utilize multiple processors (such as J_3^0 and J_3^1) and that some jobs can appear in multiple rows (such as jobs J_4 and J_5).

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
time-slice 0	J_1^0	J_1^1	J_1^2	J_1^3	J_1^4	J_1^5	J_1^6	J_1^7
time-slice 1	J_2^0	J_2^1	J_2^2	J_2^3	J_2^4	J_2^5	J_2^6	J_2^7
time-slice 2	J_3^0	J_3^0	J_3^1	J_3^1	J_4^0	J_4^1	J_5^0	J_5^1
time-slice 3	J_6^0	J_6^1	J_6^2	J_6^3	J_4^0	J_4^1	J_5^0	J_5^1

Figure 7: The scheduling matrix defines spatial and time allocation.

Creating one more virtual machine for the execution of a new 8-way job in the case of Figure 7 requires, in principle, only adding one more row to the Ousterhout matrix. Obviously, things are not so simple. There is a cost associated with time-sharing, due mostly to: (i) the cost of the context-switches themselves, (ii) additional memory pressure created by multiple jobs sharing nodes, and (iii) additional swap space pressure caused by more jobs executing concurrently. For that reason, the degree of time-sharing is usually limited by a parameter that we call, in analogy to uniprocessor systems, the multiprogramming level (MPL). A gang scheduling system with multiprogramming level of 1 reverts back to a space-sharing system.

In our particular implementation of gang scheduling, we operate under the following conditions:

1. Multiprogramming levels are kept at modest levels, in order to guarantee that the images of all tasks in a node remain in core. This eliminates paging and significantly reduces the cost of context switching. Furthermore, the time slices are sized so that the cost of the resulting context switches are negligible.

2. Assignments of tasks to processors are static. That is, once spatial scheduling is performed for the tasks of a parallel job, they cannot migrate to other nodes.

3. When building the scheduling matrix, we first attempt to schedule as many jobs for execution as possible, constrained by the physical number of processors and the multiprogramming level. Only after that do we attempt to *expand* a job, by making it occupy multiple rows of the matrix. (See jobs J_4 and J_5 in Figure 7.)

The results presented in this section are for a system operating under those conditions.

Gang scheduling is a time-sharing technique that can be applied together with any prioritization policy. In particular, we have shown in previous work [6, 14] that gang scheduling is very effective in improving the performance of FCFS policies. This is in agreement with the results in [17]. We have also shown that gang scheduling is particularly effective in improving system responsiveness, as measured by average job wait time. However, gang scheduling alone is not as effective as backfilling in improving average job response time, unless very high multiprogramming levels are allowed. These may not be achievable in practice by the reasons mentioned in the previous paragraphs.

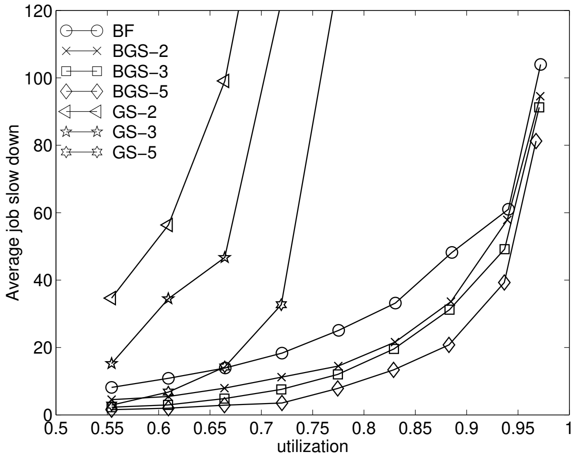
Gang scheduling and backfilling are two optimization techniques that operate on orthogonal axes, space for backfilling and time for gang scheduling. It is tempting to combine both techniques in one scheduling system. In principle this can be done by treating each of the virtual machines created by gang scheduling as a target for backfilling. The difficulty arises in estimating the execution time for parallel jobs. In the example of Figure 7, jobs J_4 and J_5 execute at a rate twice as fast as the other jobs, since they appear in two rows of the matrix. This, however, can change during the execution of the jobs, as new jobs arrive and executing jobs terminate.

Fortunately, as we have shown in Section 4, even significant average overestimation of job execution time has little impact on average performance. Therefore, it is reasonable to attempt to use a worst case scenario when estimating the execution time of parallel jobs under gang scheduling. We take the simple approach of computing the estimated time under gang scheduling as the product of the estimated time on a dedicated machine and the multiprogramming level.

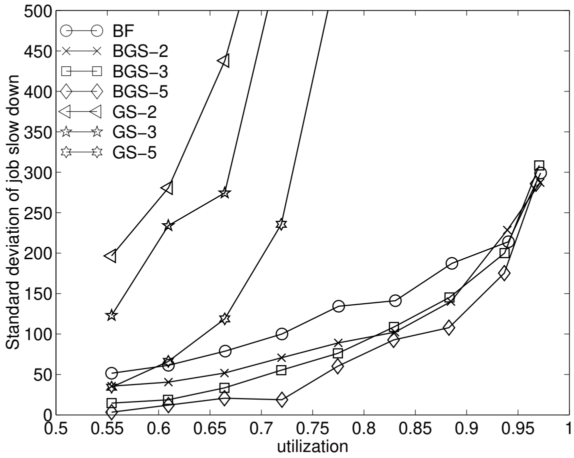
We compare seven different scheduling strategies. They all use FCFS as the prioritization policy. The first strategy is a space-sharing policy that uses backfilling to optimize the performance parameters. We identify this strategy as BF. We also use three variations of gang scheduling, with multiprogramming levels 2, 3, and 5. These strategies are identified by GS-2, GS-3, GS-5, respectively. Finally, we consider three strategies that combine backfilling and gang scheduling, with the same multiprogramming level as above. That is, backfilling is applied to each virtual machine created by gang scheduling. These are referred to as BGS-2, BGS-3, BGS-5. (It is important to note that the combined backfilling and gang scheduling with a multiprogramming level of 1 reverts back to our BF strategy.) The results presented here are based on a perfect estimation of the jobs execution time for a dedicated system. Later, we show that overestimation does not change the conclusions.

We use the performance parameters described in Section 2.3, namely (i) average slow down, (ii) average wait time, and (iii) average loss of capacity, to compare the strategies. For slow down and wait time we additionally compare the standard deviation for these parameters. The standard deviation serves as a measure of fairness: smaller standard deviations indicate that more jobs operate closer to the average and therefore closer to each other.

Figure 8 shows the slow down results for all our seven strategies. We observe that regular gang scheduling (GS strategies) results in very high slow downs, even at low or moderate (less than $\rho = 0.75$) utilizations. Backfilling basically performs much better than gang scheduling with respect to slow down. Equally, the standard deviation of slow down reveals that BF provides better fairness to the users. The combined approach (BGS) is always better than its individual components (BF and GS with corresponding multiprogramming level). The improvement in average slow down is monotonic with the multiprogramming level. This observation also applies most of the time for the standard deviation. Given a highest tolerable slow down, BGS allows the system to be driven to much higher utilizations, while preserving good fairness characteristics. We want to emphasize that significant improvements can be achieved even with the low multiprogramming level of 2. For instance, if we choose a maximum acceptable slow down of 20, the resulting maximum utilization is $\rho = 0.68$ for GS-5, $\rho = 0.73$ for BF and $\rho = 0.82$ for BGS-2. That last result represents an improvement of 20% over GS-5 with a much smaller multiprogramming level.



(a) average slowdown

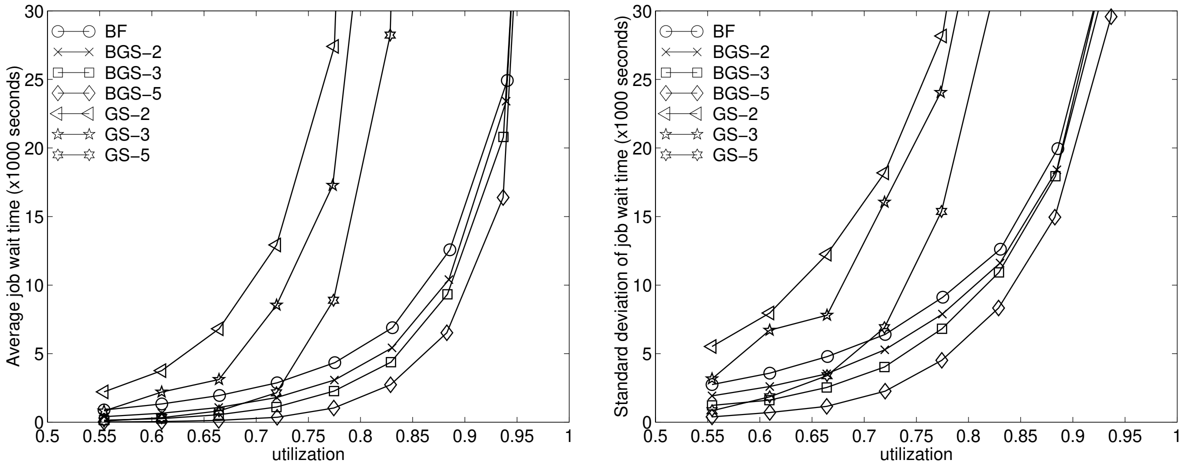


(b) standard deviation of slowdown

Figure 8: Comparing slow down for BGS with BF and GS.

Figure 9 shows the wait time results for all our seven strategies. We observe the same kind of behavior as for slow down. The combined strategies (BGS) are always superior to pure backfilling or gang scheduling, both with respect to average and standard deviation. Again we observe a very consistent monotonic improvement with the multiprogramming level. With BGS the machine appears faster, more responsive and more fair.

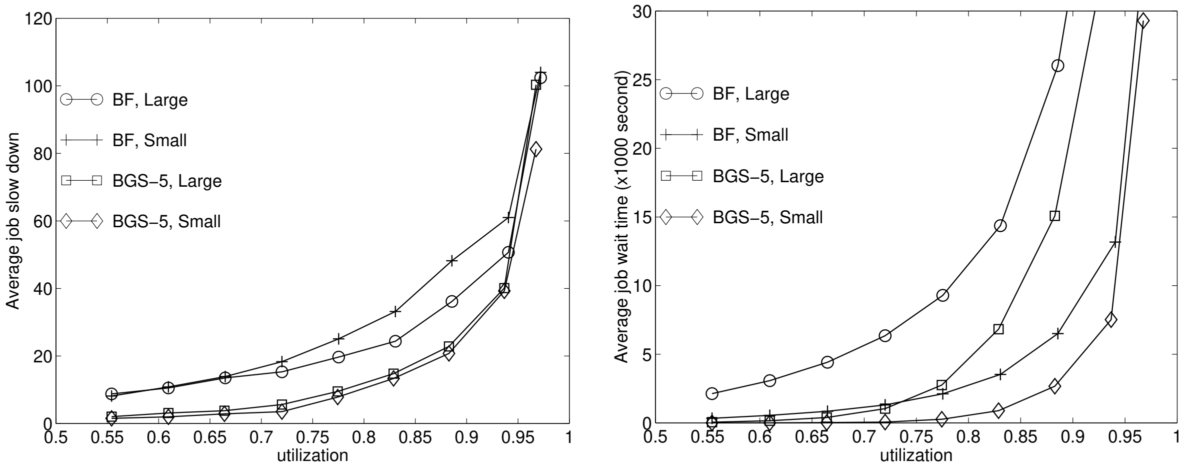
We further analyze the fairness of the scheduling strategies by comparing the behavior of the system for large and small jobs. (We remind the reader that a small job uses 32 or fewer nodes, while a large job uses more than 32 nodes.) Those results are shown in Figure 10. We observe that,



(a) average wait time (b) standard deviation of wait time

Figure 9: Comparing job wait time for BGS with BF and GS.

for a given utilization, the machine appears almost equally as slow for both large and small jobs, when the BGS strategy is used. In contrast, for BF the difference increases with higher utilizations. At $\rho = 0.90$ utilization, the machine appears 35% slower to small jobs than to large jobs. The differences between large and small jobs are more significant for the wait time parameter. Both for BF and BGS, the machine appears less responsive to large jobs than to small jobs as utilization increases. However, the difference is larger for BF.



(a) slowdown of large and small jobs (b) wait time of large and small jobs

Figure 10: Comparing the behavior of large and small jobs for BGS with BF.

At first, the results for slow down and wait time for large and small jobs may seem contradictory: small jobs have smaller wait times but larger slow down. Slow down is a relative measure, of the response time normalized by the execution time. Since smaller jobs tend to have shorter execution time, the relative cost of waiting in the queue can be larger. We note that BGS is very effective in affecting the wait time for large and small jobs in a way that ends up making the system feel equal to all kinds of jobs.

Whereas Figures 8 through 10 report performance from a user's perspective, we now turn our attention to the system's perspective. Figure 11 is a plot of the average capacity loss as a function of utilization for all our seven strategies. By definition, all strategies saturate at the line $\kappa = 1 - \rho$, which is indicated by the dashed line in Figure 11. Again, the combined policies deliver consistently better results than the pure backfilling and gang scheduling (of equal MPL) policies. The improvement is also monotonic with the multiprogramming level. However, all backfilling based policies (pure or combined) saturate at essentially the same point. Loss of capacity comes from holes in the scheduling matrix. The ability to fill those holes actually improves when the load is very high. We observe that the capacity loss for BF actually starts to decrease once utilization goes beyond $\rho = 0.75$. At very high loads ($\rho > 0.95$) there are almost always small jobs to backfill arising holes in the schedule. Looking purely from a system's perspective, we note that pure gang scheduling can only be driven to utilization between $\rho = 0.82$ and $\rho = 0.87$, for multiprogramming levels 2 through 5. On the other hand, the backfilling strategies can be driven to up to $\rho = 0.97$ utilization.

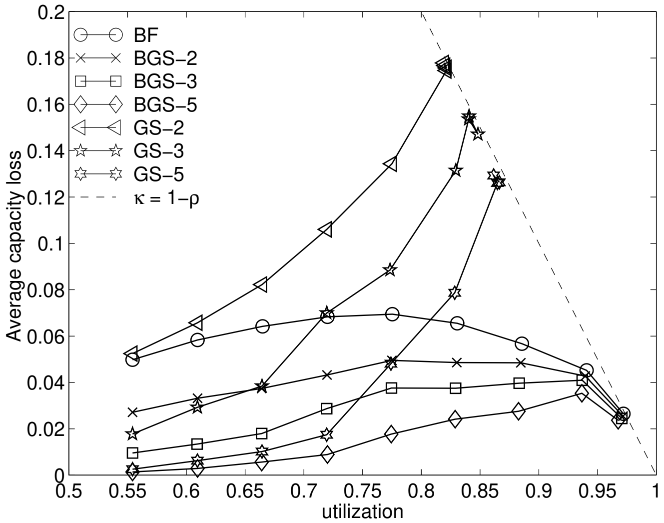
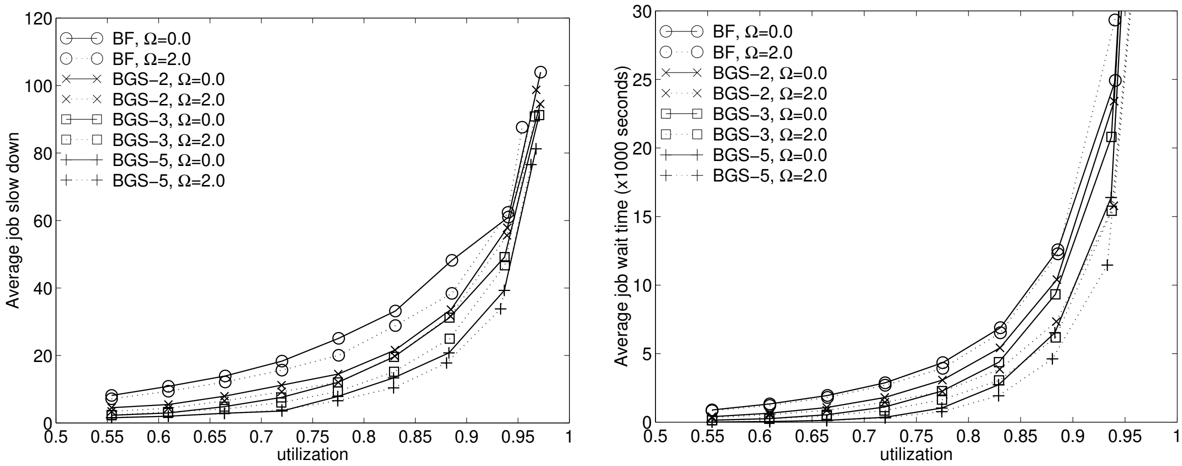


Figure 11: Comparing capacity loss for BGS with BF and GS.

One could argue that the perfect estimate of completion time ($\Omega = 0.0$) could be a reason for unduly biasing the results in favor of BGS here. To point out that this is not the case, we have also run experiments with $\Omega = 2.0$, and the performance from user's (slowdown and wait time) and system's (capacity loss) perspectives are presented in Figure 12 and Figure 13. As can be seen, a larger overestimation factor even improves performance from a user's perspective, despite a slightly higher capacity loss. Nevertheless, the overall trends and conclusions drawn comparing BGS with BF and GS for $\Omega = 0$ still hold for $\Omega = 2.0$.



(a) slow down

(b) wait time

Figure 12: Comparing performance parameters for BGS with BF and GS.

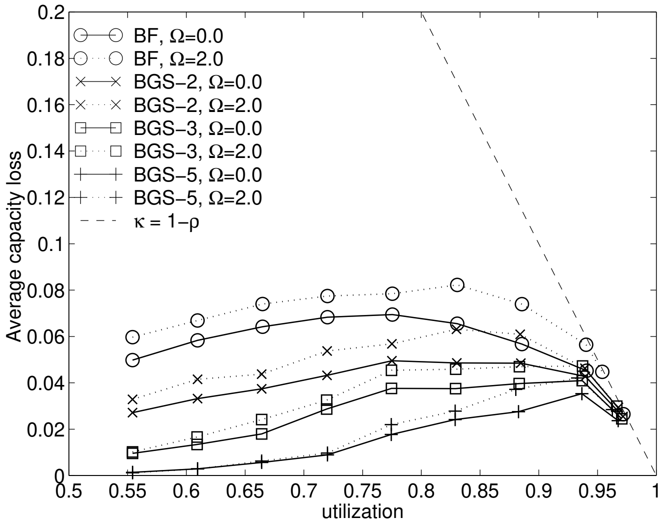


Figure 13: Average capacity loss for different scheduling strategies.

To summarize our observations, we have shown that the combined strategy of backfilling with gang scheduling (BGS) consistently outperforms the other strategies (backfilling and gang scheduling separately) from the perspectives of responsiveness, slow down, fairness, and utilization.

6 Concluding remarks and future work

With the increasing use of large scale parallel machines to tackle grand challenge problems in science and engineering, it is critical to design effective scheduling strategies to manage the high load that is placed on such machines. On one hand, it is important to maximize system utilization and throughput. On the other hand, it is important to reduce job response and wait times. All this has to be accomplished while providing fairness across jobs. Keeping these user and system administrator's performance criteria in mind, this paper has made three valuable contributions towards implementing effective job scheduling strategies for large scale parallel machines:

1. We have shown that FCFS policy, in conjunction with backfilling, does just as well as other fancier policies such as shortest job first, best fir, and worst fit. FCFS is not only straightforward to implement, but is also fair to all jobs (avoids starvation).

2. A common perception going against backfilling has been the argument that one needs an accurate estimate of job execution times for good efficiency. Contrary to this perception, we have shown that overestimation of execution times has minimal impact on the resulting system behavior. While the accuracy of estimation of an individual job compared to others is important to get relatively better performance, the overall system performance is more or less independent of the average degree of overestimation.

3. Even though it is only logical to extend backfilling techniques to time-sharing techniques such as gang scheduling, the issue of accurately estimating the job execution times has made it difficult to combine these approaches. Our results showing that overestimation has little effect on system performance make this logical transition possible. We can conservatively estimate the gang scheduling execution time to be the multiprogramming level times the estimated execution time of the job on a dedicated machine.

These three results help us take an important step towards developing an efficient execution environment for parallel applications on large scale machines. In particular, we can develop an integrated strategy called Backfilling Gang Scheduling (BGS), which combines backfilling (on a FCFS job arrival queue) with gang scheduling. We have shown how this integrated strategy outperforms a system which is just backfilled or just gang scheduled, over a spectrum of performance criteria. This exercise has involved detailed simulations of the different alternatives using workloads that have been synthesized from realistic loads at LLNL.

There are several topics for future research that are closely related to what has been studied in this paper. We would like to consider the impact of context switching costs in our BGS strategy. In particular, we want to measure the effectiveness of combined gang scheduling and backfilling in actual production use, as implemented by IBM Research in the prototype GangLL job scheduling system for ASCI Blue-Pacific. We would also like to examine issues related to migration in BGS with respect to different performance criteria. Finally, we are planning to conduct a detailed study comparing the pros and cons of space sharing, space and time sharing via coscheduling, and the recently evolving strategies falling in the class of space and time sharing via dynamic coscheduling.

References

[1] A. B. Downey. **Using Queue Time Predictions for Processor Allocation**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 35–57. Springer-Verlag, April 1997.

[2] D. G. Feitelson. **A Survey of Scheduling in Multiprogrammed Parallel Systems**. Technical Report RC 19790 (87657), IBM T. J. Watson Research Center, October 1994.

[3] D. G. Feitelson and M. A. Jette. **Improved Utilization and Responsiveness with Gang Scheduling**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 238–261. Springer-Verlag, April 1997.

[4] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. **Theory and Practice in Parallel Job Scheduling**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer-Verlag, April 1997.

[5] D. G. Feitelson and A. M. Weil. **Utilization and predictability in scheduling the IBM SP2 with backfilling**. In *12th International Parallel Processing Symposium*, pages 542–546, April 1998.

[6] H. Franke, J. Jann, J. E. Moreira, and P. Pattnaik. **An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific**. In *Proceedings of SC99, Portland, OR*, November 1999. IBM Research Report RC21559.

[7] H. Franke, P. Pattnaik, and L. Rudolph. **Gang Scheduling for Highly Efficient Multiprocessors**. In *Sixth Symposium on the Frontiers of Massively Parallel Computation, Annapolis, Maryland*, 1996.

[8] R. Gibbons. **A Historical Application Profiler for Use by Parallel Schedulers**. In *IPPS'97 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 58–77. Springer-Verlag, April 1997.

[9] B. Gorda and R. Wolski. **Time Sharing Massively Parallel Machines**. In *International Conference on Parallel Processing*, volume II, pages 214–217, August 1995.

[10] N. Islam, A. L. Prodromidis, M. S. Squillante, L. L. Fong, and A. S. Gopal. **Extensible Resource Management for Cluster Computing**. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 561–568, 1997.

[11] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan. **Modeling of Workload in MPPs**. In *Proceedings of the 3rd Annual Workshop on Job Scheduling Strategies for Parallel Processing*, pages 95–116, April 1997. In Conjuction with IPPS'97, Geneva, Switzerland.

[12] H. D. Karatza. **A Simulation-Based Performance Analysis of Gang Scheduling in a Distributed System**. In *Proceedings 32nd Annual Simulation Symposium*, pages 26–33, San Diego, CA, April 11-15 1999.

[13] D. Lifka. **The ANL/IBM SP scheduling system**. In *IPPS'95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer-Verlag, April 1995.

[14] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette. **An Infrastructure for Efficient Parallel Job Execution in Terascale Computing Environments**. In *Proceedings of SC98, Orlando, FL*, November 1998.

[15] J. E. Moreira, H. Franke, W. Chan, L. L. Fong, M. A. Jette, and A. Yoo. **A Gang-Scheduling System for ASCI Blue-Pacific**. In *High-Performance Computing and Networking, 7th International Conference*, volume 1593 of *Lecture Notes in Computer Science*, pages 831–840. Springer-Verlag, April 1999.

[16] J. K. Ousterhout. **Scheduling Techniques for Concurrent Systems**. In *Third International Conference on Distributed Computing Systems*, pages 22–30, 1982.

[17] U. Schwiegelshohn and R. Yahyapour. **Improving First-Come-First-Serve Job Scheduling by Gang Scheduling**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[18] J. Skovira, W. Chan, H. Zhou, and D. Lifka. **The EASY-LoadLeveler API project**. In *IPPS'96 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 41–47. Springer-Verlag, April 1996.

[19] W. Smith, V. Taylor, and I. Foster. **Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance**. In *Proceedings of the 5th Annual Workshop on Job Scheduling Strategies for Parallel Processing*, April 1999. In conjunction with IPPS/SPDP'99, Condado Plaza Hotel & Casino, San Juan, Puerto Rico.

[20] K. Suzaki and D. Walsh. **Implementation of the Combination of Time Sharing and Space Sharing on AP/Linux**. In *IPPS'98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[21] K. K. Yue and D. J. Lilja. **Comparing Processor Allocation Strategies in Multiprogrammed Shared-Memory Multiprocessors**. *Journal of Parallel and Distributed Computing*, 49(2):245–258, March 1998.

A Illustrating the impact of overestimating job execution time

The following example shows a situation where better performance is achieved with overestimation than with exact execution times. Let there be 2 processors, P_0 and P_1, in our system. We estimate the job running time by multiplying its actual execution time by 1.5. We consider the execution of three jobs, with the following parameters:

job	arrival time	size (# of processors)	actual execution time	estimated execution time
A	0	1	4	6
B	1	2	4	6
C	3	1	2	3

If we have perfect knowledge of each job exact execution time, we use that for scheduling. In that case, the schedule looks as follows:

time	0	1	2	3	4	5	6	7	8	9
arrival	A	B		C						
P_0	A	A	A	A	B	B	B	B	C	C
P_1					B	B	B	B		

If instead we use the estimated job execution time for scheduling, the schedule looks as follows:

time	0	1	2	3	4	5	6	7	8
arrival	A	B		C					
P_0	A	A	A	A		B	B	B	B
P_1				C	C	B	B	B	B

Comparing the two schedules, we can see that overestimation of job execution time may give us better average job response time and may result in better system utilization than when using an exact estimate.