A Comparative Analysis of Space- and Time-Sharing Techniques for Parallel Job Scheduling in Large Scale Parallel Systems

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. This paper discusses three techniques that can be used beyond simple space-sharing to greatly improve the performance figures of large parallel systems. The first technique we analyze is backfilling, the second is gang-scheduling, and the third is migration. The main contribution of this paper is an analysis of the effects of combining the above techniques. Using extensive simulations based on detailed models of realistic workloads, the benefits of combining the various techniques are shown 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 (SNL). 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 [3, 4, 7, 10, 13, 14, 20, 21, 24]. Early strategies 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. Submitted jobs are kept in a priority queue which is always traversed according to a priority policy in search of the next job to execute. 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 can be relatively high.

Among the several approaches used to alleviate these problems with space sharing scheduling, three have been recently studied. The first is a technique called backfilling [14], 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 approach requires the users to

provide an estimate of job execution times, in addition to the number of nodes required by each job. Jobs that exceed their execution time are killed. This encourages users to overestimate the execution time of their jobs.

The second approach is to add a time-sharing dimension to space sharing using a technique called gang-scheduling or coscheduling [17]. This technique virtualizes the physical machine by slicing the time axis into multiple 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 advantageous 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 LLNL (a large scale parallel system spanning thousands of nodes [16]).

The third approach is to dynamically migrate tasks of a parallel job. Migration delivers flexibility of adjusting your schedule to avoid fragmentation. Migration is particularly important when collocation in space and/or time of tasks is necessary. Collocation in space is important in some architectures to guarantee proper communication among tasks (*e.g.*, Cray T3D, CM-5, and Blue Gene). Collocation in time is important when tasks have to be running concurrently to make progress in communication (*e.g.*, gang-scheduling).

It is a logical next step to attempt to combine these approaches – gang-scheduling, backfilling, and migration – to deliver even better performance for large parallel systems. However, effectively implementing these combinations raises some challenges. For instance, straightforward combining backfilling and gang-scheduling requires obtaining precise estimates for job execution time under gang scheduling. This can be very difficult or even impossible. Similarly, migration incurs a cost and requires additional infrastructure. Migration costs make it more difficult to estimate execution times and decide if migration should be applied. In analyzing these combinations, we only consider systems like the IBM RS/6000 SP, that do not require spatial collocation. Therefore, we only address migration in the presence of gang-scheduling.

Progressing to combined approaches requires a careful examination of several issues related to backfilling, gang-scheduling, and migration. Using detailed simulations based on stochastic models derived from real workloads at LLNL, this paper analyzes (i) the impact of the priorty queueing mechanism in backfilling, (ii) the impact of overestimating job execution times on the effectiveness of backfilling, (iii) a strategy for combining gang-scheduling and backfilling, (iv) the impact of migration in a gang-scheduled system, and (v) the impact of combining gang-scheduling, migration, and backfilling in one scheduling system.

We find that a First Come First Serve (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. We also find that 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 [17] with jobs that are not necessarily in FCFS order. It is clearly demonstrated that, under certain conditions, this combined strategy is always better than the individual gang-scheduling or backfilling strategies for all the quality of service parameters that we consider. By combining gang-scheduling and migration we can further improve the system performance parameters. The improvement is larger when applied to plain gang-scheduling (without backfilling), although the absolute best performance was achieved by combining all three techniques: gang-scheduling, backfilling, and migration.

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 describes gang-scheduling, and the various phases involved in computing a time-sharing schedule. Section 6 demonstrates the significant improvements in performance that can be achieved with time-sharing techniques, particulary when enhanced with backfilling and migration. Finally, Section 7 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 of the workload 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.

We now describe these steps in more detail.

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, both job interarrival time distribution and job service time (execution time on a dedicated system) distribution have coefficients of variation that are 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 [12] 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 at LLNL.

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 [12]. Finally, we simulate the effects of these synthetic workloads and observe the results.

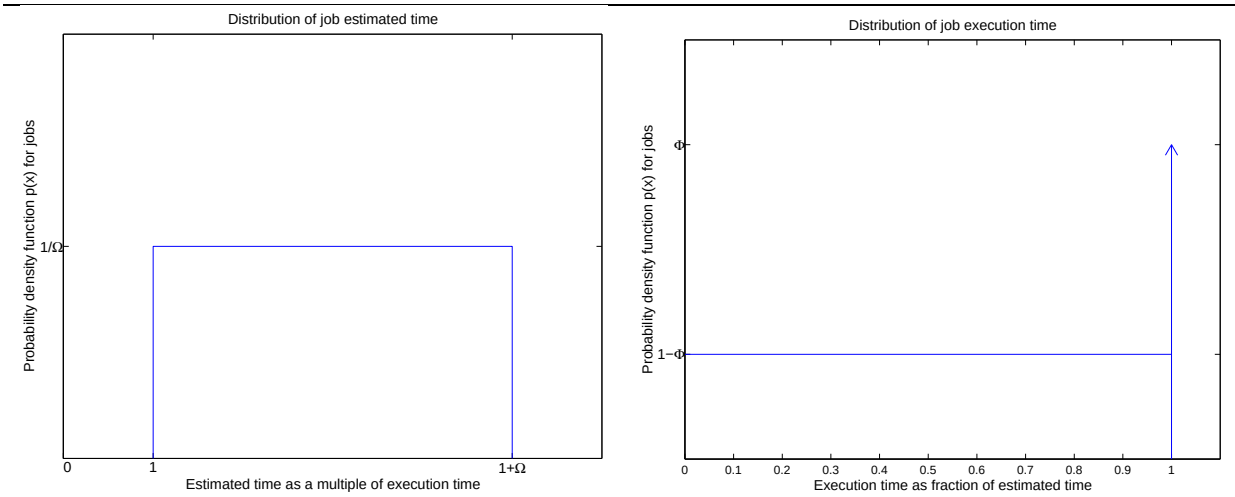
Within a workload trace, each job is described by its arrival time, the number of nodes it uses, its execution time on a dedicated system, and an overestimation factor. Backfilling strategies require an estimate of the job execution time. In a typical system, it is up to the each user to provide these estimates. This estimated execution time is always greater than or equal to the actual execution time, since jobs are terminated after reaching this limit. We capture this discrepancy between estimated and actual execution times for parallel jobs through an *overestimation factor*. The oversestimation factor for each job is the ratio between its estimated and actual execution times. 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. In this paper we consider two models for describing the distribution of estimated execution times as provided by the user.

In the first model, which we call the Ω model, we obtain the estimate by multiplying the dedicated execution time by the overestimation factor, which is a uniformly distributed random number between 1 and an upper limit $1 + \Omega$. This distribution is shown in Figure 1(a). In particular, $\Omega = 0$ indicates we have perfect knowledge of how long jobs are going to run.

We also make use of the Φ model. In the Φ model, Φ is the fraction of jobs that terminate at exactly the estimated time. This typically corresponds to jobs that are killed by the system because they reach the limit of their allocated time. The rest of the jobs $(1 - \Phi)$ are distributed such that the distribution of jobs that end at a certain fraction of their estimated time is uniform. This is shown in Figure 1(b). To obtain the desired distribution for execution times in the Φ model, in our simulations we compute the overestimation factor as follows: Let y be a uniformly distributed random number in the range $0 \leq y < 1$. If $y < \Phi$, then the overestimation factor is 1 (*i.e.*, estimated time = execution time). If $y \geq \Phi$, then the overestimation factor is $(1 - \Phi)/(1 - y)$.

2.2 Workload characteristics

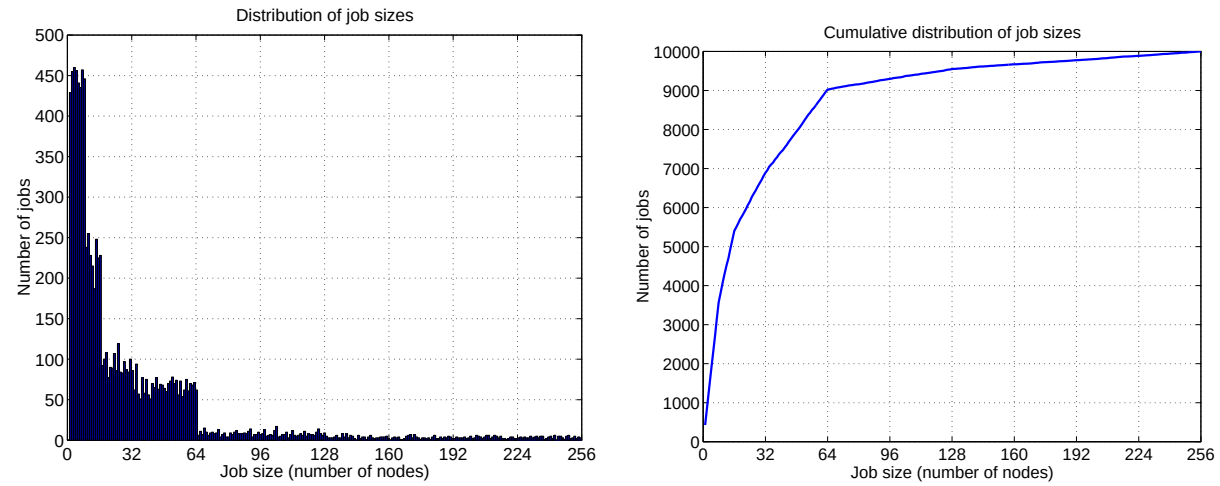
The baseline workload is the synthetic workload generated from the parameters directly extracted from the actual ASCI Blue-Pacific workload. It consists of 10000 jobs, varying in size from 1 to 256 nodes, in a system with a total of 320 nodes. Some characteristics of this workload are shown in Figures 2 and 3. Figure 2 reports the distribution of job sizes (number of nodes). For each job size, between 1 and 256, Figure 2(a) shows the number of jobs of that size, while Figure 2(b) plots the number of jobs with *at most* that size. (In other words, Figure 2(b) is the integral of Figure 2(a).) Figure 3 reports the distribution of



(a) Ω model (b) Φ model

Figure 1: The (a) Ω and (b) Φ models for overestimation.

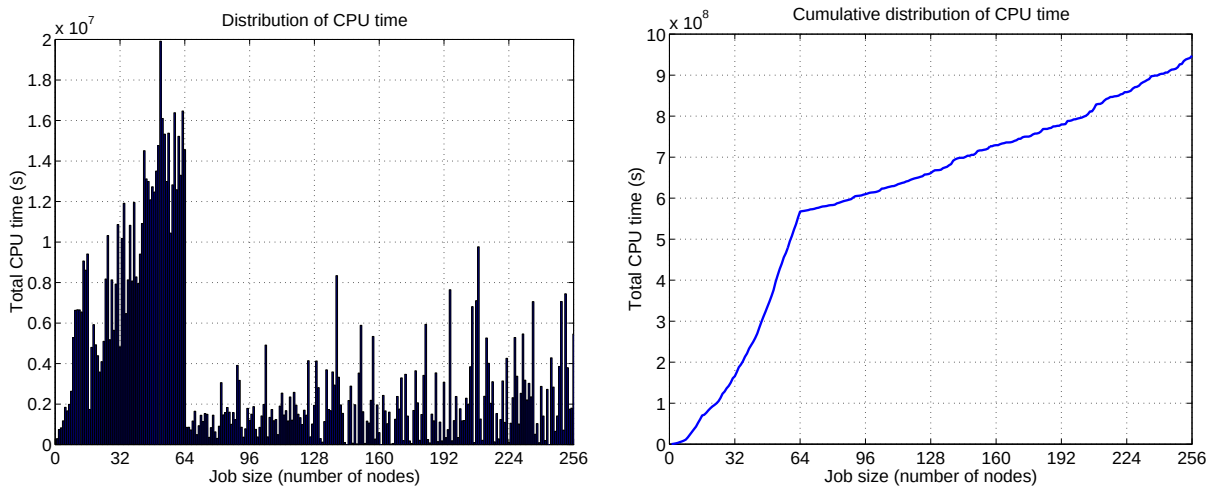
total CPU time, defined as job execution time on a dedicated setting times its number of nodes. For each job size, Figure 3(a) shows the sum of the CPU times for all jobs of that size, while Figure 3(b) is a plot of the sum of the CPU times for all jobs of *at most* that size. (In other words, Figure 3(b) is the integral of Figure 3(a).) From Figures 2 and 3 we observe that, although large jobs (defined as 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 2: Workload characteristics: distribution of job sizes.

In addition to the baseline workload of Figures 2 and 3 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



(a) (b)

Figure 3: Workload characteristics: distribution of cpu time.

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. We simulate a system with 320 nodes, and 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).
- 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.
- $s_i = \frac{\max(t_i^r, \Gamma)}{\max(t_i^e, \Gamma)}$: the slowdown for job i. To reduce the statistical impact of very short jobs, it is common practice [5, 6] to adopt a minimum execution time of Γ seconds. This is the reason for the $\max(\cdot, \Gamma)$ terms in the definition of slowdown. According to the literature [6], we adopt $\Gamma = 10$ seconds.

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

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

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 in a purely space-shared system is then defined as

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

To compute the loss of capacity in a gang-scheduling system, we have o keep track of what happens in each time-slice. Let s_i be the number of time slices between scheduling event i and scheduling event $i + 1$. Also, let T_{ij} be the length of the j-th times slice betweens scheduling events i and $i + 1$. We can then define

$$\kappa = \frac{\sum_{i=1}^{2m-1} \left[\sum_{j=1}^{s_i} T_{ij}(e_{ij} + \sum_{k \in H_{ij}}(n_k \times w_k)) + T \times C \sum_{j \in J_{ij}} n_j\right]}{t_m^f \times N} \tag{3}$$

where

- e_{ij} is the number of empty nodes at the j-th time-slice between scheduling events i and $i + 1$;
- T is the base length of the time slice ($T_{ij} \leq T$);
- C is the context-switch overhead (as a fraction of time-slice);
- n_k is number of nodes of job k;
- w_k is fraction of time-slice not used by job k;
- J_{ij} is set of jobs that were context-switched into execution at the j-th time-slice between scheduling events i and $i + 1$;
- H_{ij} is set of jobs that terminate during the j-th time-slice between scheduling events i and $i + 1$;

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 4. Suppose we have to schedule five jobs, numbered from 1 to 5 in order of arrival. Figure 4(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 4(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.

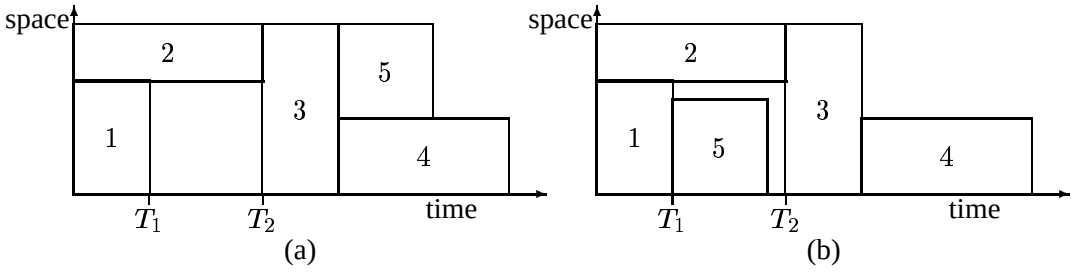
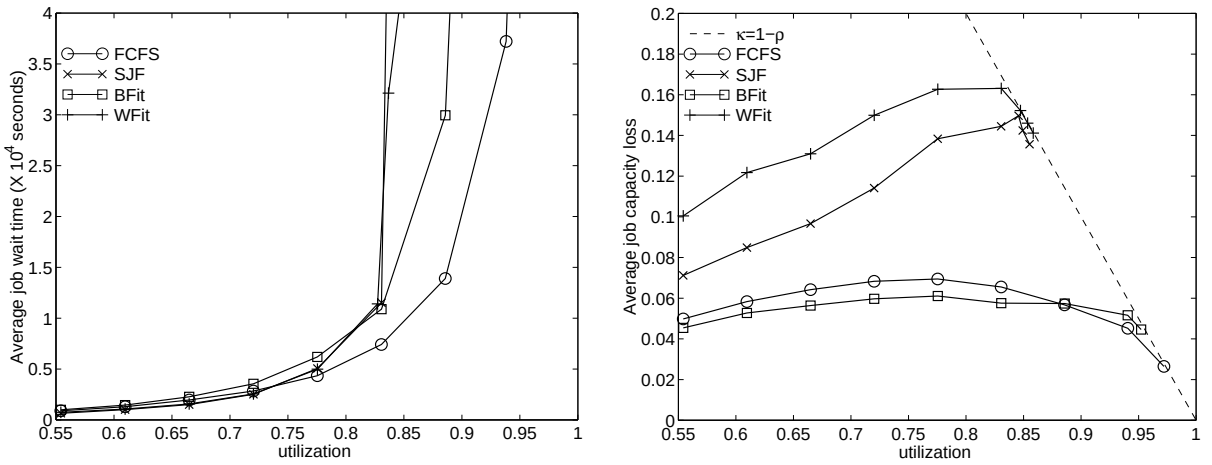


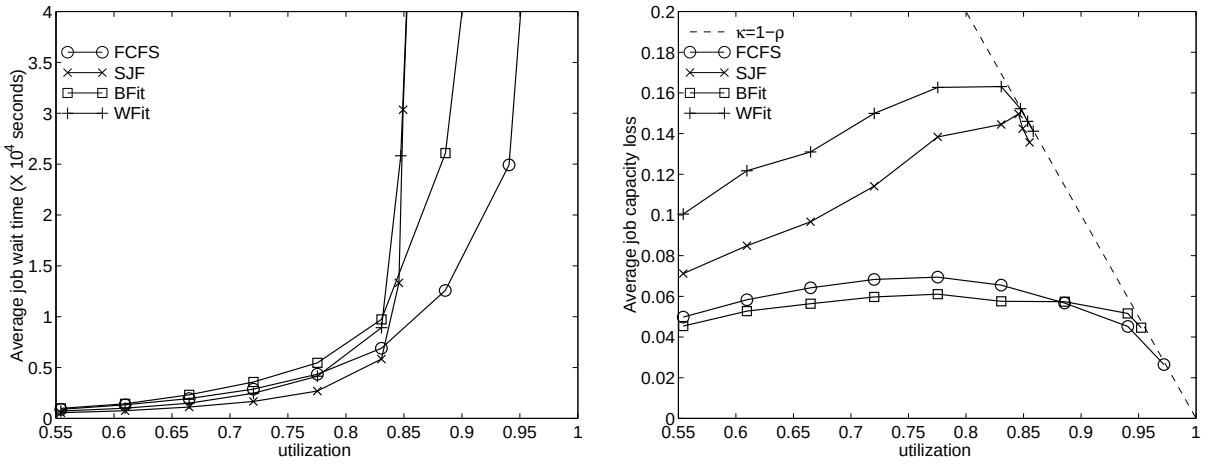
Figure 4: FCFS policy without (a) and with (b) backfilling. Job numbers correspond to their position in the priority queue.

Figure 5 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



(a) wait time for perfect estimation

(b) loss of capacity for perfect estimation



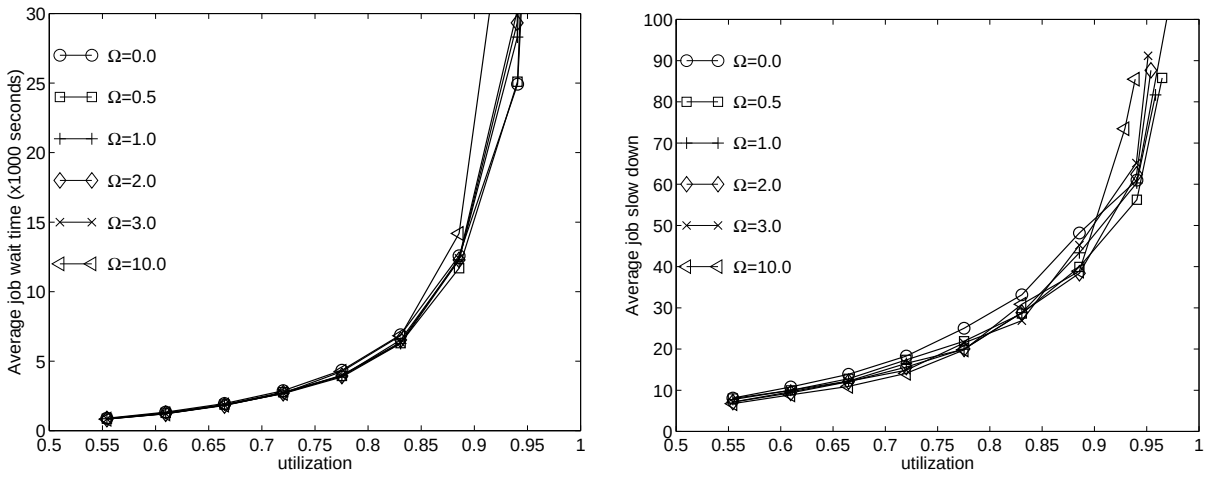
(c) wait time for $\Phi = 0.2$

(d) loss of capacity for $\Phi = 0.2$

Figure 5: Average job wait time and capacity loss for different queueing policies with backfilling.

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 ($\Omega = 0$, or $\Phi = 1$) when performing scheduling. We refer to this as *perfect estimation backfilling.* Results for these parameters are shown in Figures 5(a) and (b). The dashed line in Figure 5(b) is a plot of $\kappa = 1 - \rho$ and, as discussed in Section 2.3, represents the loci of maximum utilization (saturation) points. For completeness, we show in Figures 5(c) and (d) results for $\Phi = 0.2$.

From Figure 5(a), we observe that at lower utilization (up to 75%) all policies are comparable, with SJF displaying slightly better performance. However, at higher utilizations FCFS performs better than the other policies. From Figure 5(b), we observe that both SJF and WFit saturate at an utilization of 85%, while both BFit and FCFS can sustain utilizations of more than 90%. However, at this high loads, FCFS exhibits better average job wait time than BFit. On top of that, FCFS is straightforward to implement and 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.



(a) Wait time (b) Slow down

Figure 6: Average job slowdown and wait time for backfilling under Ω model of overestimation.

4 The impact of overestimation on backfilling

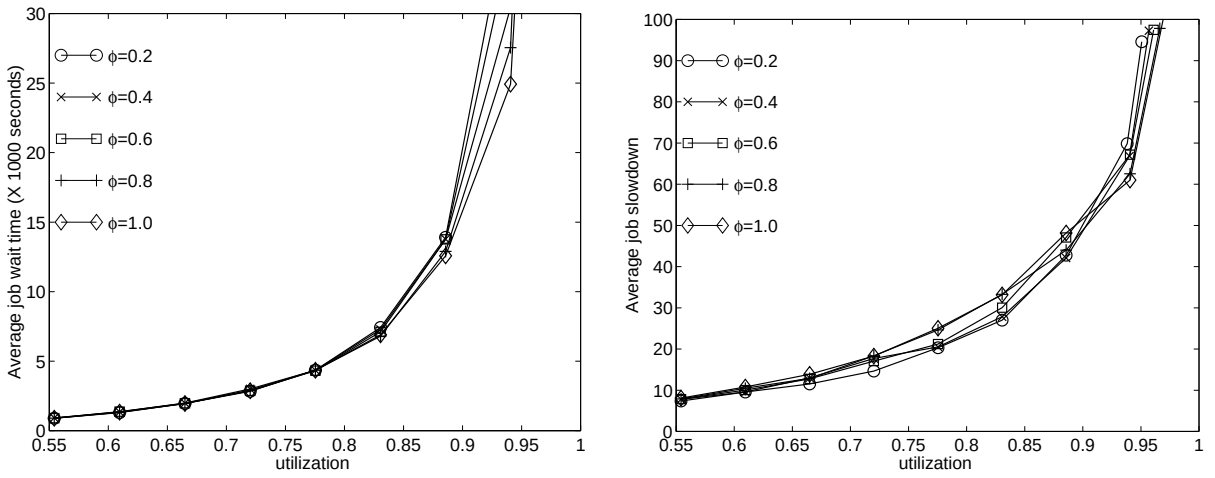
A common perception with backfilling is that one needs a fairly accurate estimation of job execution time to perform good backfilling scheduling. Users typically provide an estimate of job execution time when jobs are submitted. However, it has been shown in the literature [6] that there is little correlation 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 6 for the Ω model of overestimation and in Figure 7 for the Φ model. Figures 6(a) and 6(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 estimation 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. Figures 7(a) and 7(b) plot average job slow down and average job wait time, respectively, as a function of system utilization for different values of Φ. Again, we observe very little impact of overestimation. However, in contrast with the Ω model, we can see a little benefit in wait time from more accurate estimates.

We can explain 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



(a) Wait time (b) Slow down

Figure 7: Average job slowdown and wait time for backfilling under Φ model of overestimation.

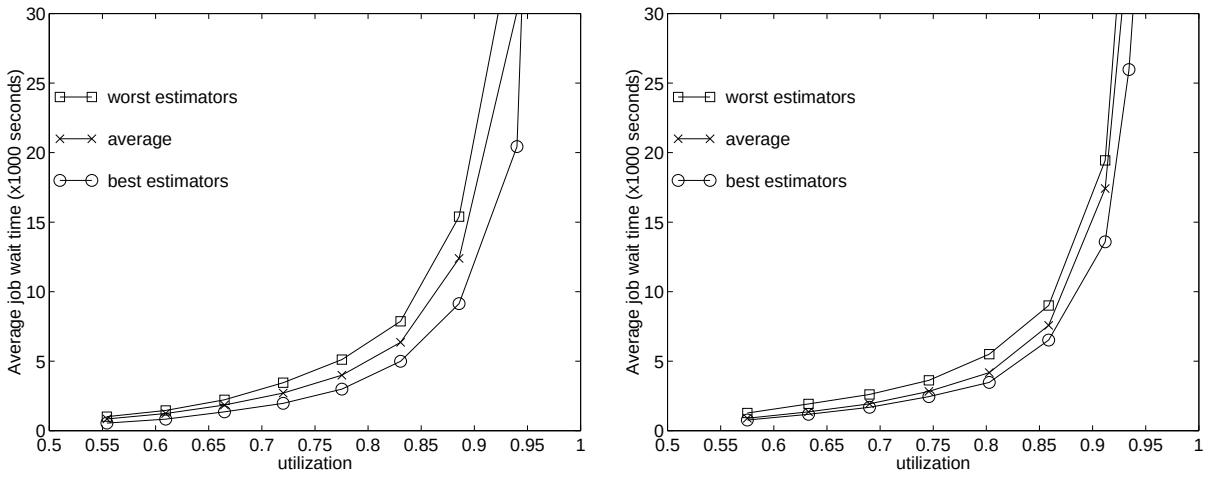
running jobs, creating larger holes in the schedule. The larger holes can then be used to accommodate waiting jobs that have overestimated execution times. The probability of finding a backfilling candidate effectively does not change with the overestimation.

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 close is their estimated time to their actual execution time. More precisely, for the Ω model, 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 8(a) 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 ($\Omega = 3$). For the Φ model, class i, $i = 0, \ldots, 9$ includes all those jobs for which their ratio of execution time to estimated time falls in the range $(i \times 10\%, (i+1) \times 10\%]$. Figure 8(b) shows the average job wait time for (i) all jobs, (ii) jobs in class 0 (worst estimators) and (iii) jobs in class 9 (best estimators) when $\Phi = 0.2$. 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.

Our findings are in agreement with the work described in [22]. 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 [2, 9].

5 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 [3, 4, 8, 11, 23]. As an example, time-sharing an 8-processor system with a multiprogramming level of four is shown in Figure 9. The figure shows the



(a) Ω model (b) Φ model

Figure 8: The impact of good estimation from a user perspective for the Ω and Φ models of overestimation.

scheduling matrix (also called the *Ousterhout 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. One cycle through all the rows of the matrix defines a *scheduling cycle.* 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* [3]. 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 9: The scheduling matrix defines spatial and time allocation.

5.1 Considerations in building a scheduling matrix

Creating one more virtual machine for the execution of a new 8-way job in the case of Figure 9 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 small.

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 we attempt to *expand* a job, by making it occupy multiple rows of the matrix. (See jobs J_4 and J_5 in Figure 9.) Our results demonstrate that this approach yields better performance than trying the fill the matrix with already running jobs.

4. For a particular instance of the Ousterhout matrix, each job has an assigned *home row*. Even if a job appears in multiple rows, one and only one of them is the home row. The home row of a job can change during its life time, when the matrix is recomputed. The purpose of the home row is described in Section 5.2.

Gang-scheduling is a time-sharing technique that can be applied together with any prioritization policy. In particular, we have shown in previous work [7, 15] that gang-scheduling is very effective in improving the performance of FCFS policies. This is in agreement with the results in [20]. 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 also delivers additional benefits beyond the scope of performance. In particular, the gang-scheduling infrastructure can be used to implement the important feature of *preemption.* With preemption, the execution of a less important job is suspended to allow the execution of a more important job. Preemption is accomplished in gang-scheduling by simply removing all tasks of a job from the scheduling matrix.

5.2 The phases of scheduling

Every job arrival or departure constitutes a *scheduling event* in the system. For each scheduling event, a new scheduling matrix is computed for the system. Even though we analyze various scheduling strategies in this paper, they all follow an overall organization for computing that matrix, which can be divided into the following steps:

1. **CleanMatrix:** The first phase of a scheduler removes every instance of a job in the Ousterhout matrix that is not at its assigned home row. Removing duplicates across rows effectively opens the opportunity of selecting other waiting jobs for execution.

2. **CompactMatrix:** This phase moves jobs from less populated rows to more populated rows. It further increases the availability of free slots within a single row to maximize the chances of scheduling a large job.

3. **Schedule:** This phase attempts to schedule new jobs. We traverse the queue of waiting jobs as dictated by the given priority policy until no further jobs can be fitted into the scheduling matrix.

4. **FillMatrix:** This phase tries to fill existing holes in the matrix by replicating jobs from their home rows into a set of replicated rows. This operation is essentially the opposite of **CleanMatrix**.

The exact procedure for each step is dependent on the exact scheduling strategy and the details will be presented as we discuss each strategy.

Orthogonal to the strategy, there is also the issue of the latency of the scheduler. One option is to always invoke the scheduler exactly at the time of a job arrival or departure. In that case a new schedule is computed and takes effect immediately. In other words, the current time slice is cut short. Alternatively, the schedule is only invoked at discrete times. That is, at the predetermined context switches time at the end of every time slice. In this case, every time slice runs to completion. The advantage of the first approach is that empty slots can be immediately utilized by an arriving job, or when a job departs, the remaining jobs can use the new free slots. These factors contribute to reduce capacity loss. The disadvantage of this approach is that when the job arrival and departure is too high the system can go into a thrashing mode, because context switches are more frequent.

6 Scheduling strategies

We now describe and analyze in detail the various time-shared scheduling strategies in our work. We start with plain gang-scheduling (GS), as described in Section 5. We augment it with backfilling capabilities to produce our backfilling gang-scheduling (BGS) strategy. We also analyze what happens when migration is added to gang-scheduling, thus creating the migration gang-scheduling (MGS) strategy. Finally, we combine both enhancing techniques (backfilling and migration) into the migration backfiling gang-scheduling (MBGS) strategy.

When analyzing the performance of the time-shared strategies we have to take into account the context-switch overhead. Context switch overhead is the time used by the system in suspending a currently running job and resuming the next job. During this time, the system is not doing useful work from a user perspective, and that is why we characterize it as overhead. In the RS/6000 SP, context switch overhead includes the protocol for detaching and attaching to the communication device. It also includes the operations to stop and continue user processes. When the working set of time-sharing jobs is larger than the physical memory of the machine, context switch should also include the time to page in the working set of the resuming job. For our analysis, we characterize context switch overhead as a percentage of time slice. Typical context switch overhead values are from 0 to 5% of time slice.

6.1 Gang-scheduling (GS)

The first scheduling strategy we analyze is plain gang-scheduling (GS). This strategy is described in Section 5. For gang-scheduling, we implement the four scheduling steps of Section 5.2 as follows.

CleanMatrix: The implementation of CleanMatrix is best illustrated with the following algorithm:

```
for i = first row to last row
    for all jobs in row i
        if row i is not home of job, remove it
```

It eliminates all occurrences of a job in the scheduling matrix other than the one in its home row.

CompactMatrix: We implement the CompactMatrix step in gang-scheduling according to the following algorithm:

```
for i = least populated row to most populated row
    for j = most populated row to least populated row
        for all jobs in row i
            if they can be moved to row j, then move
```

We traverse the scheduling matrix from the least populated row to the most populated row. We attempt to find new homes for the jobs in each row. The goal is to pick the most jobs in the least number of rows.

To move a job to a different row under gang-scheduling, the following conditions must be satisfied:

1. The destination columns, which are the same as the source columns for the job, must be empty.
2. The job must make progress. That is, we must ensure that moving the job will not prevent it from executing for at least one time-slice in one scheduling cycle. This must be enforced to prevent starvation of jobs.

To guarantee progress of jobs, we adopt the following straightforward algorithm for deciding where it is legal to move jobs. We call it the *clock algorithm*, which is illustrated in Figure 10. The algorithm works as follows: Each scheduling cycle corresponds to one turn of the clock. Each scheduling event corresponds to one particular time in the clock. The last time a job was run also corresponds to a particular time. A job can only be moved *ahead.* That is, to any time between now and the time corresponding to its last run. Once a job is moved to a different row, that becomes its new home row. (A job can appear in multiple rows of the matrix. Therefore, the time of last run could be later than the home row.)

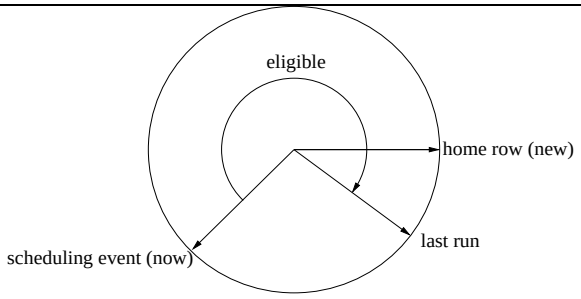


Figure 10: The clock algorithm.

Schedule: The Schedule phase for gang-scheduling traverses the waiting queue in FCFS order. For each job, it looks for the row with the least number of free slots in the scheduling matrix that has enough free columns to hold the job. This corresponds to a best fit algorithm. The row to which the job is assigned becomes its home row. We stop when the next job in the queue cannot be scheduled right away.

FillMatrix: After the schedule phase completes, we proceed to fill the holes in the matrix with the existing jobs. We use the following algorithm in executing the FillMatrix phase.

```
do {
   for each job in starting time order
       for all rows in matrix,
         if job can be replicated in same columns do it and break
} while matrix changes
```

The algorithm attempts ot replicate each job at least once, although some jobs can be replicated multiple times. We go through the job in starting time order, but other ordering policies can be applied.

Figure 11 illustrates the performance impact of the different steps in the gang-scheduling algorithm. Line (1) in each of the plots (slow down and wait time) is for a configuration in which we only perform the Schedule and FillMatrix steps in each scheduling event. With this approach, an expanded job does not loose rows in the scheduling matrix. As a net effect, it is more difficult to schedule new jobs. In contrast, for line (2) we perform the CleanMatrix step before the Scheduling step. This opens more opportunities for scheduling waiting jobs. Only after the Scheduling phase, we perform FillMatrix. This approach favors running as many jobs as possible, and we can see a beneficial impact on system performance with respect to slow down, wait time, and maximum achievable utilization. In line (3), we show the results for the configuration in which we perform all four phases of gang-scheduling. The additional CompactMatrix phase before Scheduling has the effect of opening up more space for large jobs. However, we do not see a significant impact in performance from adding this phase.

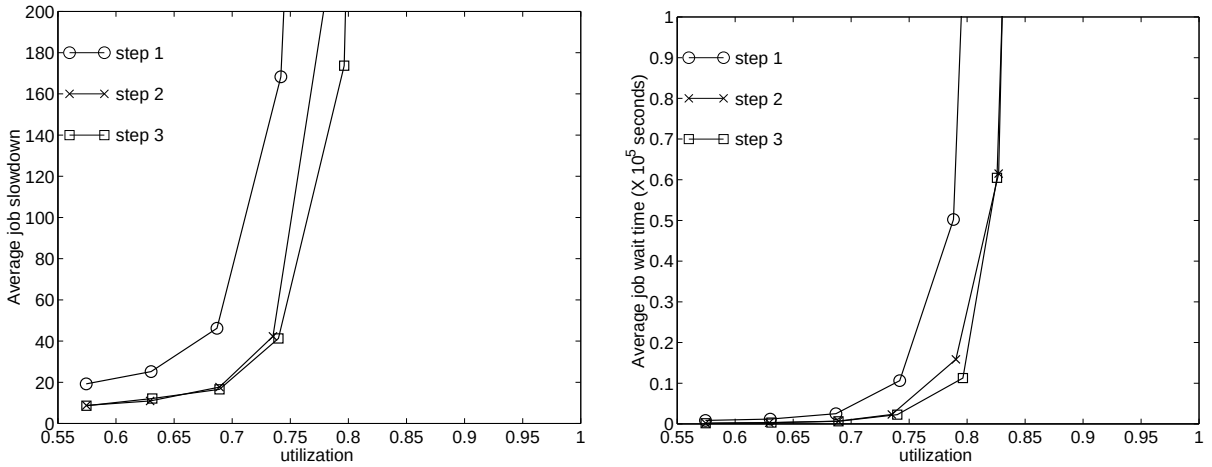


Figure 11: Average job wait time and slow down showing the performance of gang-scheduling (with MPL = 5 and time slice = 200 seconds) when we perform (1) Schedule+FillMatrix, (2) CleanMatrix+Schedule+FillMatrix, and (3) CleanMatrix+CompactMatrix+Schedule+FillMatrix.

6.2 Backfilling gang-scheduling (BGS)

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 that we call *backfilling gang-scheduling* (BGS). 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 9, 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.

In backfilling, each waiting job is assigned a maximum starting time based on the predicted execution times of the current jobs. That start time is a reservation of resources for waiting jobs. The reservation corresponds to a particular time in a particular row of the matrix. It is possible that a job will be run before its reserved time and in a row different than reserved. However, using a reservation guarantees that the start time of a job will not exceed a certain limit, thus preventing starvation.

The issue of reservations impact both the CompactMatrix and Schedule phases. When moving jobs in CompactMatrix we must make sure that the moved job does not conflict with any reservations in the destination row. In the Schedule phase, we first attempt to schedule each job in the waiting queue, making sure that its execution does not violate any reservations. If we cannot start a job, we compute the future start time for that job in each row of the matrix. We select the row with the lowest starting time, and make a reservation for that job in that row. This new reservation could be different from the previous reservation of the job. The reservations do not impact the FillMatrix phase, since the assignments in this phase are temporary and the matrix gets cleaned in the next scheduling event.

Figure 12 illustrates the performance impact of the different steps in the backfill gang-scheduling algorithm. As before, line (1) is for a configuration in which we only perform the Schedule and FillMatrix steps, for line (2) we perform the CleanMatrix step before the Schedule step, and in line (3) we show the results for the configuration in which we perform all four phases of gang-scheduling. We note that for BGS, the impact of the CleanMatrix and CompactMatrix phases is minimal. In BGS, backfilling does the job of filling the holes in the matrix and therefore leaves no opportunity for the other two phases to make a noticeable impact.

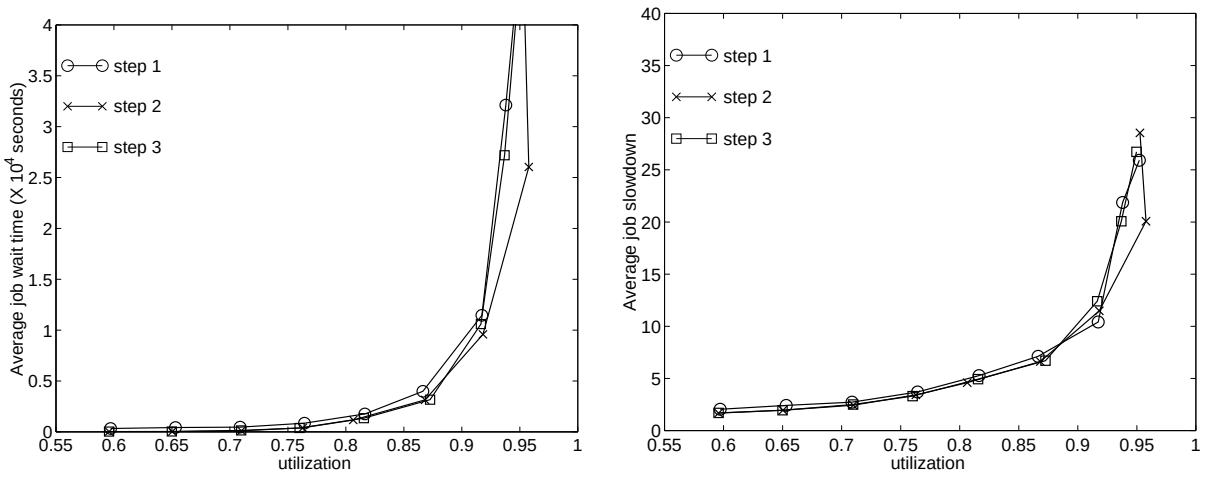


Figure 12: Average job wait time and slow down showing the performance of backfilling gang-scheduling (with MPL = 5 and time slice = 200 seconds) when we perform (1) Schedule+FillMatrix, (2) CleanMatrix+Schedule+FillMatrix, and (3) CleanMatrix+CompactMatrix+Schedule+FillMatrix.

To verify that indeed the assumption that overestimation of job execution times do not impact overall system performance, we experimented with various values of Ω and Φ. Results for the Ω and Φ models are shown in Figure 13 and Figure 14, respectively. For those plots, BGS with all four phases and MPL=5 was used. We observe that differences in wait time are insignificant across the entire range of utilization. For moderate utilizations of up to 75%, job slowdown differences are also insignificant. For utilizations of 85% and higher, job slowdown exhibits larger variation with respect to overestimation, but the variation is nonmonotonic and perfect estimation is not necessarily better.

Although the Ω model was widely used in the literature, it has been shown to not correspond well with actual user estimates. Therefore, from this point on we adopt the Φ model, with $\Phi = 0.2$, which matches

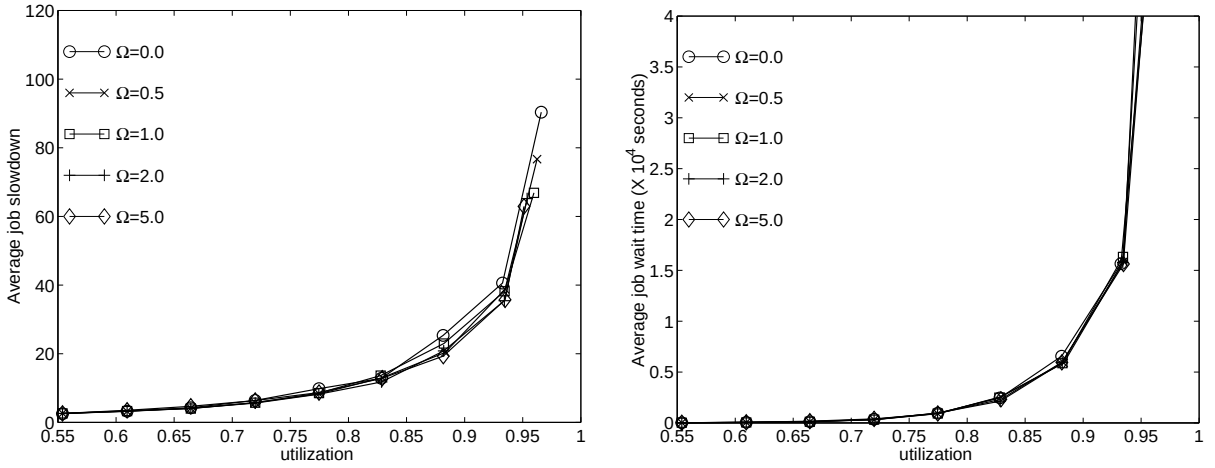


Figure 13: Average job wait time and slow down for BGS (best) with Ω model of overestimation.

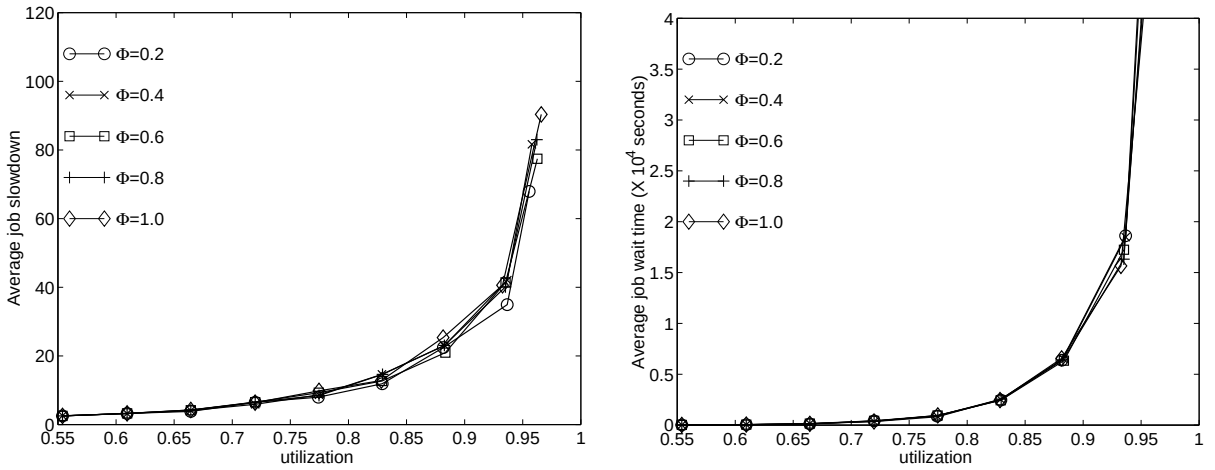


Figure 14: Average job wait time and slow down for BGS (best) with Φ model of overestimation.

more closely experimental measurements [6].

6.3 Comparing GS, BGS, and BF

We compare three different scheduling strategies, with a total of seven configurations. They all use FCFS as the prioritization policy. The first strategy is a space-sharing policy that uses backfilling to enhance the performance parameters. We identify this strategy as BF. We also use three variations of the gang-scheduling strategy, with multiprogramming levels 2, 3, and 5. These configurations are identified by GS-2, GS-3, GS-5, respectively. Finally, we consider three configurations of the backfilling gang-scheduling strategy. That is, backfilling is applied to each virtual machine created by gang-scheduling. These are referred to as BGS-2, BGS-3. and BGS-5, for MPL 2, 3, and 5. The results presented here are based on the Φ-model, with $\Phi = 0.2$.

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.

Figures 15 and 16 show the average and standard deviation of job slow down for all our seven configurations. Each plot ((a), (b), (c), and (d)) is for a different value of context switch overhead. 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. BF always performs better than GS-2 and GS-3. It also performs better than GS-5 when utilization is greater than 0.65. Equally, the standard deviation of slow down in Figure 16 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 $\rho = 0.67$ for GS-5, $\rho = 0.76$ 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. With BGS-5, we can drive utilization as high as $\rho = 0.87$.

Figures 17 and 18 show the average and standard deviation of job wait time for all our seven configurations. Again, each plot is for a different value of context-switch overhead. We observe that regular gang-scheduling (GS strategies) results in very high wait times, even at low or moderate (less than $\rho = 0.75$) utilizations. Even with 0% context switching overhead, saturation takes place at $\rho = 0.84$ for GS-5 and at $\rho = 0.79$ for GS-3. At 5% overhead, the saturations occur at $\rho = 0.73$ and $\rho = 0.75$ for GS-3 and GS-5 respectively. Backfilling performs better than gang-scheduling with respect to wait time for utilizations above $\rho = 0.72$. It saturates at $\rho = 0.95$. The standard deviation of wait times in Figure 18 reveals that BF provides better fairness to the users than GS for utilizations above $\rho = 0.68$. The combined approach (BGS) is always better than its individual components (BF and GS with corresponding multiprogramming level) for a zero context switch overhead. The improvement in average job wait time is monotonic with the multiprogramming level. This observation also applies most of the time for the standard deviation. With BGS and zero context switch overhead, the machine appears faster, more responsive and more fair.

At all combinations of context switch overhead and utilization, BGS outperforms GS with the same MPL. BGS also outperforms BF at low context switch overheads 0% or 1%. Even at context switch overhead of 2% or 5%, BGS has significantly better slowdown than BF in an important operating range. For 2%, BGS-5 saturates at $\rho = 0.93$ whereas BF saturates at $\rho = 0.95$. Still, BGS-5 is significantly better than BF for utilization up to $\rho = 0.92$. For context switch overhead of 5%, BGS-5 is superior to BF only up to $\rho = 0.83$. Therefore, we have two options in designing the scheduler system: we either keep the context switch overhead low enough that BGS is always better than BF or we use an adaptive scheduler that switches between BF and BGS depending on the utilization of the system. Let $\rho_{critical}$ be the utilization at which BF starts performing better than BGS. For utilization smaller than $\rho_{critical}$, we use BGS. When utilization goes beyond $\rho_{critical}$, we use BF. Further investigation of adaptive scheduling is beyond the scope of this paper.

We further analyze the scheduling strategies by comparing the behavior of the system for large and small jobs. (As defined in Section 2.2, small job uses 32 or fewer nodes, while a large job uses more than 32 nodes.) The results for slowdown and wait times are shown in Figure 19, when a 0% context switch overhead is used. With respect to slowdown, we observe that, BGS-5 always performs better than BF for either large or small jobs. For any utilization, BGS-5 provides better slowdown for small jobs, while BF provides better slowdown for large jobs. This behavior is explained by the fact that larger jobs in general tend to execute longer than smaller jobs. Although BGS reduces the average wait time for all jobs, it increases the execution time by the effective multiprogramming level. Therefore, longer running jobs will

benefit less from gang scheduling. With respect to wait time, we observe that the improvement generated by BGS is actually larger for large jobs. In other words, for any given utilization, the difference in wait time between large and small jobs is less in BGS-5 than in BF. 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.

At first, the BF 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.

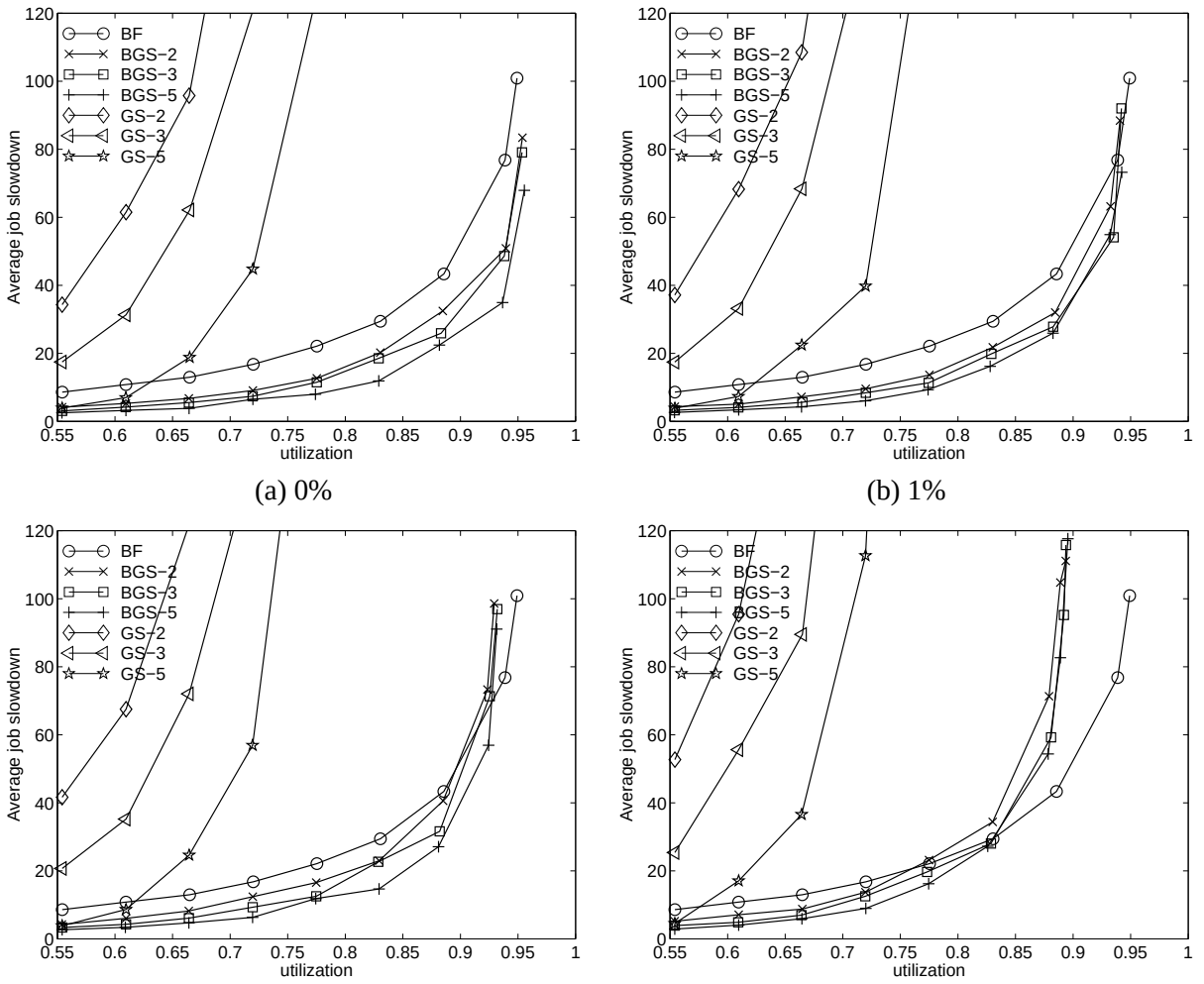
Whereas Figures 15 through 19 report performance from a user's perspective, we now turn our attention to the system's perspective. Figure 20 is a plot of the average capacity loss as a function of utilization for all our seven strategies. By definition, all strategies saturate at the line $\kappa = 1 - \rho$, which is indicated by the dashed line in Figure 20. 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 $\rho = 0.83$. 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 $\rho = 0.95$ utilization.

To demonstrate the importance of continuous scheduling, Figure 21 shows slowdown as a function of utilization for all scheduling strategies where (a) continuous scheduling is used and (b) discrete scheduling is used. We observe that, with discrete scheduling, BF is superior to all other strategies except BGS-5 for low to medium utilization ($\rho < 0.83$). BF does not have a concept of time-slice, so it always uses continuous scheduling. For reference, the curve BGS-1 indicates the performance of BF with discrete scheduling. If we look through the operating range $0.55 \leq \rho \leq 0.80$, the slowdown for BGS-1 is roughly double that of BF. Using the usual maximum acceptable slowdown of 20, discrete scheduling can drive the system to up to $\rho = 0.79$, whereas continuous scheduling can go as high as $\rho = 0.88$. In both cases, the maximum utilization is reached with BGS-5.

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. For BGS to realize this advantage, context switch cost must be kept low. We have shown BGS to be superior to BF over the entire spectrum of workloads when the context switch overhead is 1% or less of the time slice.

6.4 Migration gang-scheduling (MGS)

We now analyze how gang-scheduling can be improved through the addition of migration capabilities. The process of migration embodies moving a job to any row in which there are enough free processors to execute that job. There are basically two options each time we attempt to migrate a job A from a source row r to a target row p (in either case, row p must have enough nodes free):



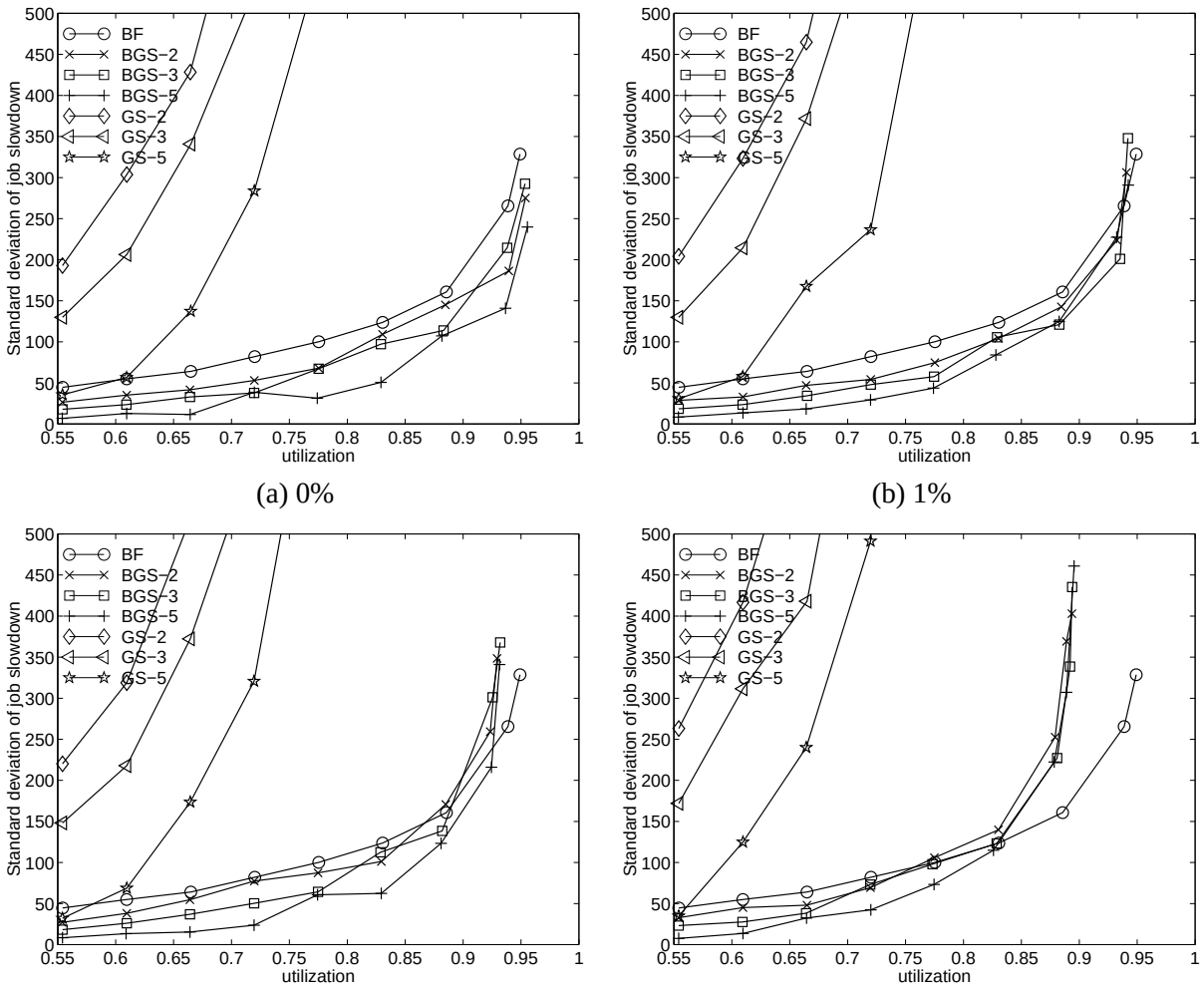
(a) 0% (b) 1%

(c) 2% (d) 5%

Figure 15: Average job slowdown for four different values of context switch overhead: 0% (a), 1% (b), 2% (c), 5%(d) of time slice.

- *Option 1*: We migrate the jobs which occupy the nodes of job A at row p, and then we simply replicate job A, in its same set of nodes, in row p.
- *Option 2*: We migrate job A to the set of nodes in row p that are free. The other jobs at row p remain undisturbed.

We can quantify the cost of each of these two options based on the following model. For the distributed system we target, namely the IBM RS/6000 SP, migration can be accomplished with a checkpoint/restart operation. (Although it is possible to take a more efficient approach of directly migrating processes across nodes [1, 18, 19], we choose not to take this route.) Let $S(A)$ be the set of jobs in target row p that overlap with the nodes of job A in source row r. Let C be the total cost of migrating one job, including the checkpoint and restart operations. We consider the case in which (i) checkpoint and restart have the same cost $C/2$, (ii) the cost C is independent of the job size, and (iii) checkpoint and restart are dependent operations (*i.e.*, you have to finish checkpoint before you can restart). During the migration process, nodes participating in the migration cannot make progress in executing a job. The total amount of resources (processor $\times$ time)



(a) 0% (b) 1%

(c) 2% (d) 5%

Figure 16: Standard deviation of job slowdown for four different values of context switch overhead: 0% (a), 1% (b), 2% (c), 5%(d) of time slice.

wasted during this process is the overhead for the migration opreation.

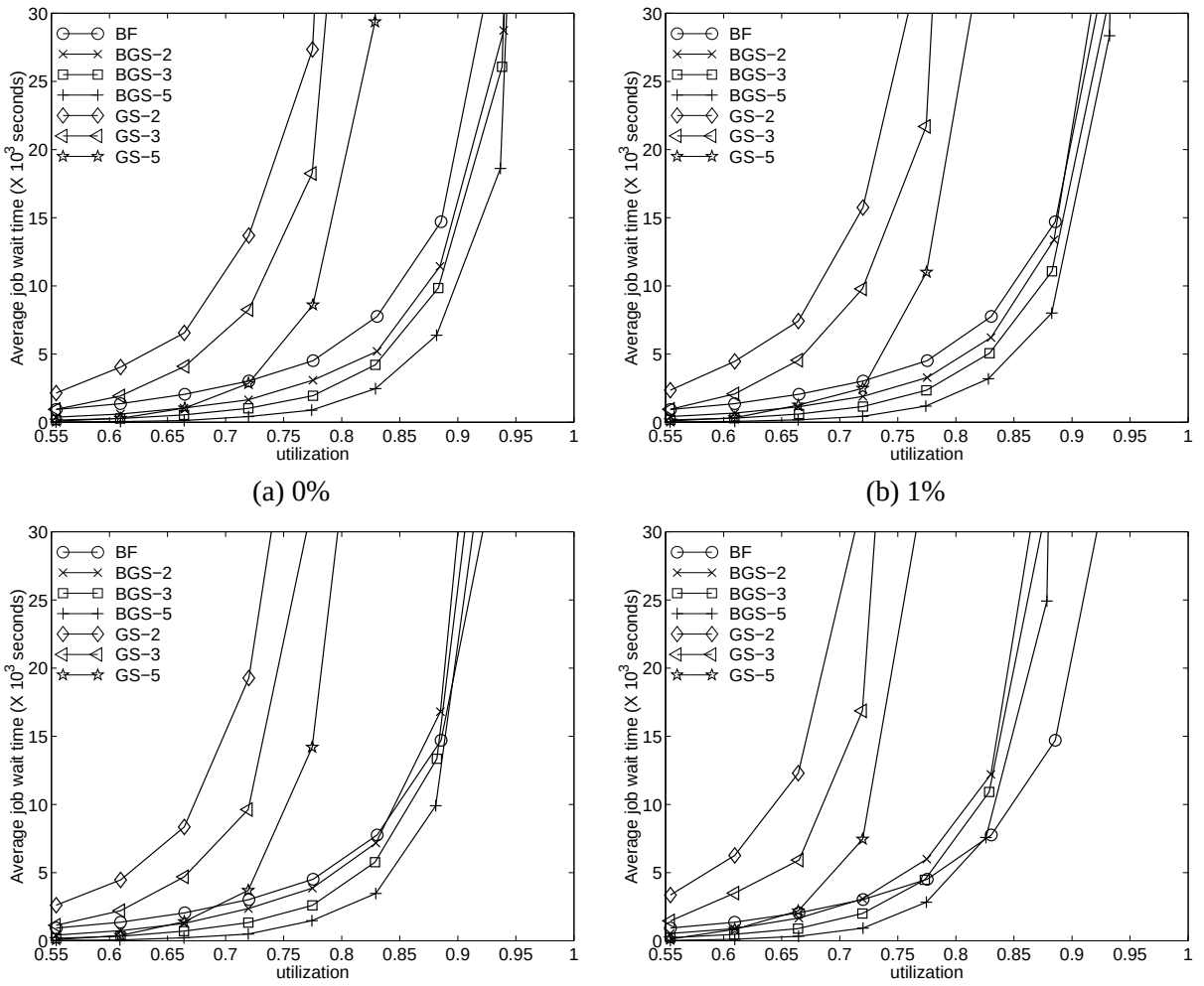
The overhead for option 1 is

$$(\frac{C}{2} \times |A| + C \times \sum_{J \in S(A)} |J|), \tag{4}$$

where $|A|$ and $|J|$ denote the number of tasks in jobs A and J, respectively. The operations for option 1 are illustrated in Figure 22(a), with a single job J in set $S(A)$. The first step is to checkpoint job J in its current set of nodes. This checkpointing operation takes time $C/2$. As soon as the checkpointing is complete we can resume execution of job A. Therefore, job A incurs an overhead $\frac{C}{2} \times |A|$. To resume job J in its new set of nodes requires a restart step of time $\frac{C}{2}$. Therefore, the total overhead for job J is $C \times |J|$.

The overhead for option 2 is estimated by

$$(C \times |A| + \frac{C}{2} \times \sum_{J \in S(A)} |J|). \tag{5}$$



(a) 0% (b) 1%

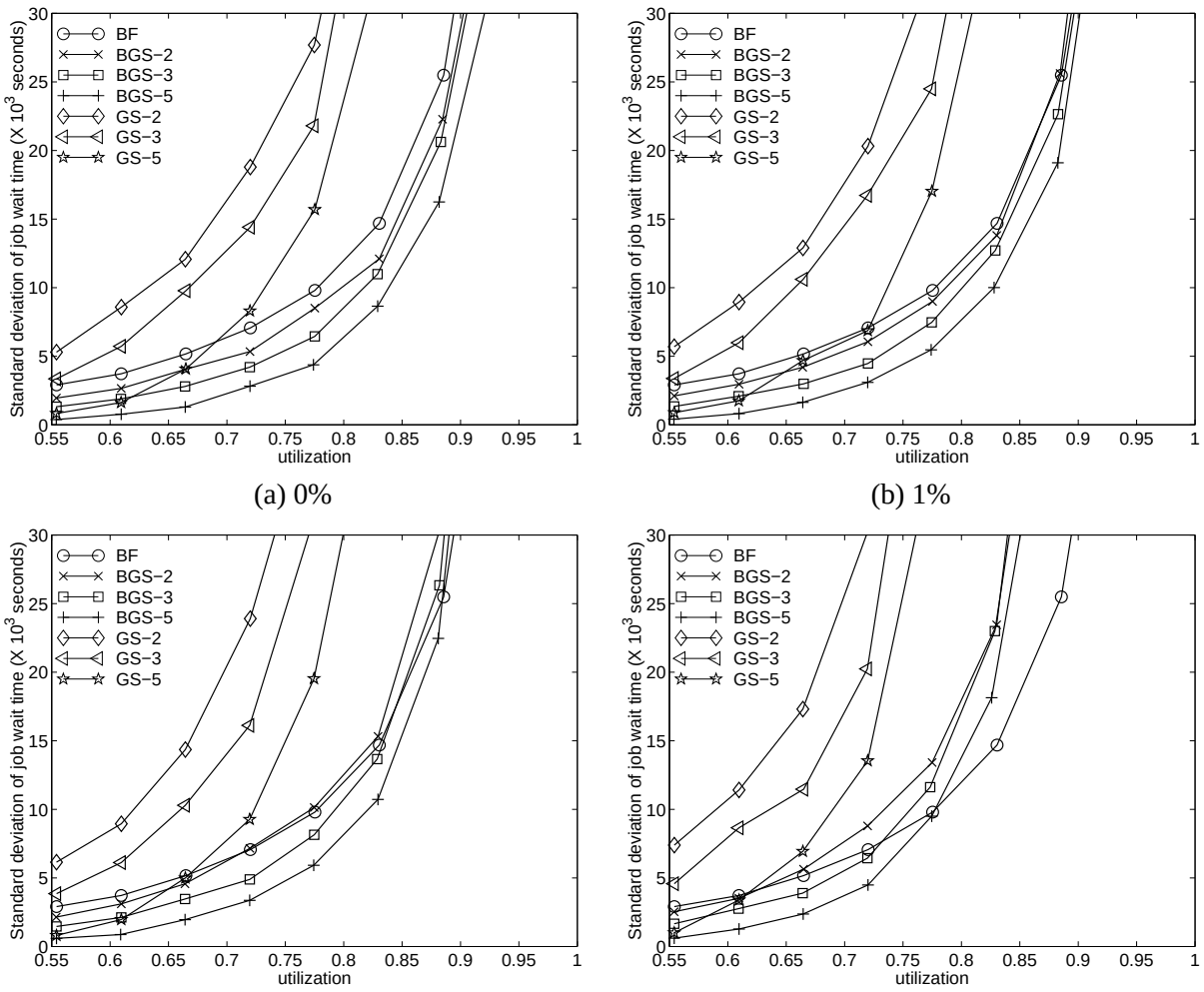
(c) 2% (d) 5%

Figure 17: Average job wait times for four different values of context switch overhead: 0% (a), 1% (b), 2% (c), 5%(d) of time slice.

The migration for option 2 is illustrated in Figure 22(b), with a single job J in set $S(A)$. The first step is to checkpoint job A. This checkpoint operation takes time $\frac{C}{2}$. After job A is checkpointed we can resume execution of job J. Therefore, the overhead for job J is $\frac{C}{2} \times |J|$. To resume job A we need to restart it in its new set of processors, which again takes time $\frac{C}{2}$. The overhead for job A is then $C \times |A|$.

The first use of migration is during the compact phase, in which we consider migrating a job when moving it to a different row. The goal is to maximize the number of empty slots in some rows, thus facilitating the scheduling of large jobs. The order of traversal of jobs during the CollapseMatrix phase is from the least populated row to the most populated row, wherein each row the traversal continues from the smallest job (least number of processors) to the largest job. During the compact phase, both migration options discussed above are considered, and we choose the one with smaller cost.

We also apply migration during the expansion phase. If we cannot replicate a job in a different row because its set of processors are busy with another job, we attempt to move the blocking job to a different set of processors. A job can appear in multiple rows of the matrix, but it must occupy the same set of processors in all the rows. This rule prevents the ping-pong of jobs. For the expansion phase, jobs are



(a) 0% (b) 1%

(c) 2% (d) 5%

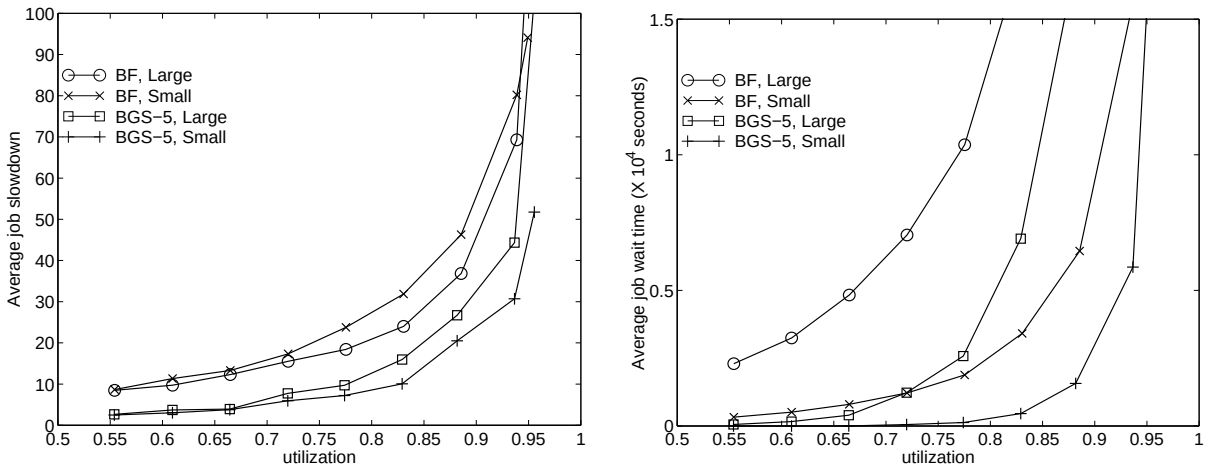
Figure 18: Standard deviation of wait times for four different values of context switch overhead: 0% (a), 1% (b), 2% (c), 5%(d) of timeslice.

traversed in first-come first-serve order. During expansion phase, only migration option 1 is considered.

As discussed, migration in the IBM RS/6000 SP requires a checkpoint/restart operation. Although all tasks can perform a checkpoint in parallel, resulting in a C that is independent of job size, there is a limit to the capacity and bandwidth that the file system can accept. Therefore we introduce a parameter Q that controls the maximum number of tasks that can be migrated in any time-slice.

When migration is used, the scheduling proceeds along the following steps:

step	reason
ClearMatrix	Maximize holes
CollapseMatrix-1	Compaction without migration
Schedule-1	Accomodate new jobs after compaction
CollapseMatrix-2	Compaction with migration
Schedule-2	Accomodate new jobs in holes created after migration
FillMatrix-1	Replicate jobs in different holes without migration
FillMatrix-2	Replicate jobs after migrating destination



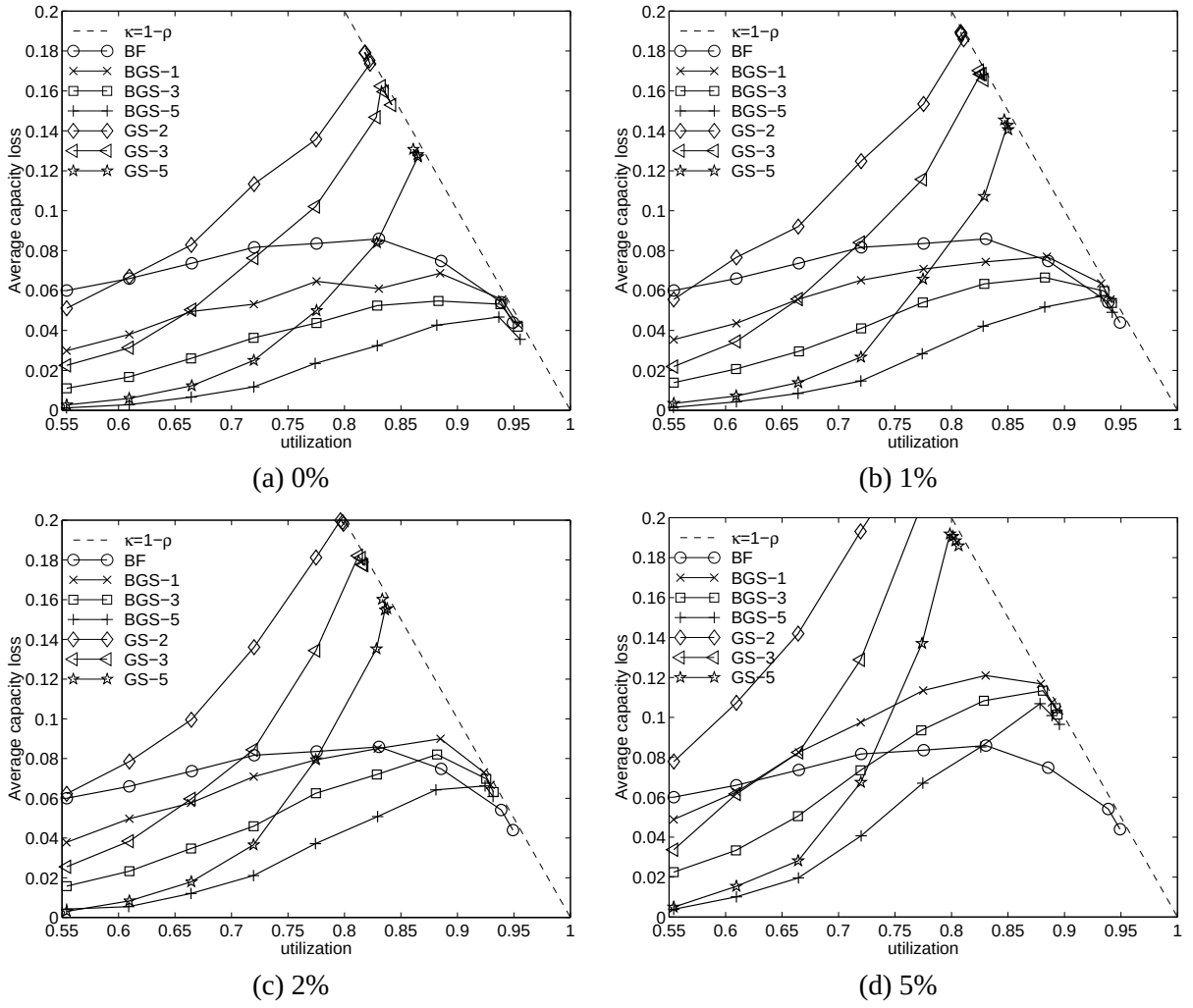
(a) slowdown (b) wait time

Figure 19: Slowdown and wait time for large and small jobs.

The ordering results in applying optimizations without incurring unnecessary costs. We first attempt to optimize without migration (CollapseMatrix-1,Schedule-1). After Schedule-1, we then attempt to collapse with migration (CollapseMatrix-2) and repeat scheduling (Schedule-2) to accomodate new jobs. After we are done accomodating new jobs, we do FillMatrix-1 first because it does not incur in a migration cost. Then we try FillMatrix-2 with migration.

The algorithm for CollapseMatrix-2 is the same as for CollapseMatrix-1 in GS. The only difference are the conditions for moving a job. With migration, a job can be moved to any row and any set of columns, provided that (i) enough empty columns are available in the destination row, (ii) number of migrated tasks does not violate the Q parameter, and (iii) a job must make progress, that is, it must execute in at least one row for every cycle of scheduling. The last requirement is identical as for gang-scheduling (GS). If migration is required to move a job to a new target row, we consider the two options described above (option 1 and option 2) and choose the one with the least estimated cost. FillMatrix-2 uses the same algorithm as FillMatrix-1, with the following constraints when deciding to replicate a job in a new row. First, the job must not already be replicated in that row. Second, the row must have sufficient empty columns to execute the job and the total number of migrated tasks must not exceed parameter Q. Only option 1 (move jobs in target row) is considered for FillMatrix-2, and therefore those jobs must not be present in any other row of the schedule. Given these algorithms, we ensure that migration never incurs recurring cost. That is, a job will not ping-pong between different columns within the same scheduling matrix.

Figure 23 shows the impact of the different steps in the the MGS strategy. Line (1) is the result for GS, and serves as a reference. Line (2) is the result for MGS when we add to GS just the FillMatrix-2 step: After completing GS by replicating jobs on the same set of nodes, we further attempt to fill the matrix by replicating the job on rows that have enough nodes free, employing migration to move the jobs that are preventing replication. Line (3) shows the result when we further add CollapseMatrix-2 and Schedule-2. CollapseMatrix-2 moves jobs from one row to another in order to create emptier rows in the matrix. It employs migration to move either the source job (the one that is changing rows) or the jobs in the destination row that occupy the set of nodes of the source job. After CollapseMatrix-2 frees space in the rows of the matrix a new round of scheduling (Schedule-2) attempts to run more jobs. From Figure 23 we observe that most of the improvements in MGS are from adding the FillMatrix-2 step. The improvements from more aggressive migration in line (3) are offset by the additional migration cost. The net effect is that there is little



(a) 0% (b) 1%

(c) 2% (d) 5%

Figure 20: Loss of capacity for BGS, GS, and BF, with different context-switch overheads.

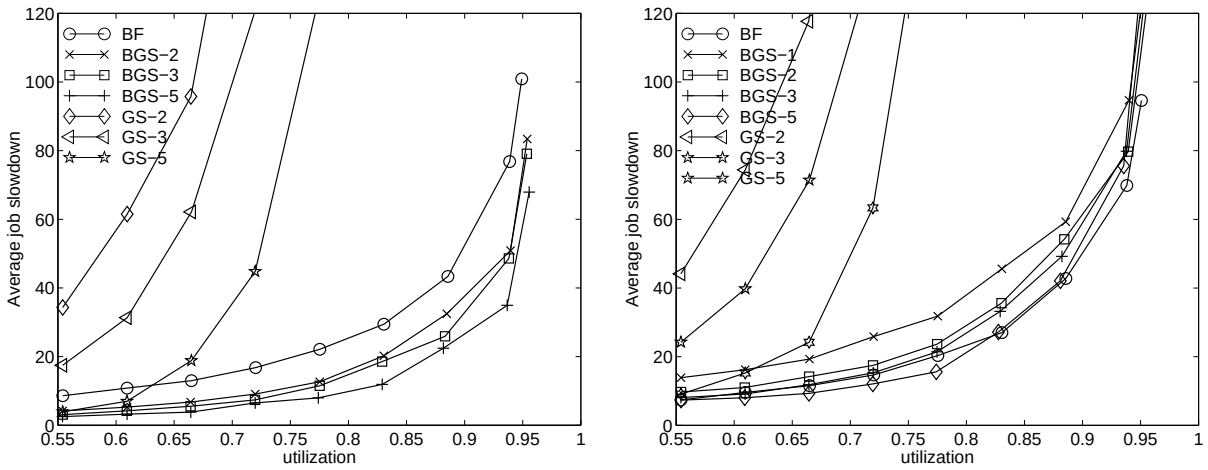
benefit from the extra Collapse and Schedule steps.

6.5 Migration backfilling gang-scheduling (MBGS)

Just as we augmented plain gang-scheduling (GS) with migration, the same can be done with backfilling gang-scheduling (BGS). This creates the migration backfilling gang-scheduling (MBGS). The differences between MGS and MBGS are in the CollapseMatrix and Schedule steps. MBGS use the same scheduling as BGS, that is, backfilling is performed in each row of the matrix, and reservations are created for jobs that cannot be immediately scheduled. When compacting the matrix, MBGS must make sure that reservations are not violated.

6.6 Comparing GS, BGS, MGS, and MBGS

Table 1 summarizes some of the results from migration applied to gang-scheduling and backfilling gang-scheduling. For each of the nine workloads (numbered from 0 to 8) we present achieved utilization (ρ) and average job slowdown (s) for four different scheduling policies: (i) backfilling gang-scheduling without



(a) continuous (b) discrete

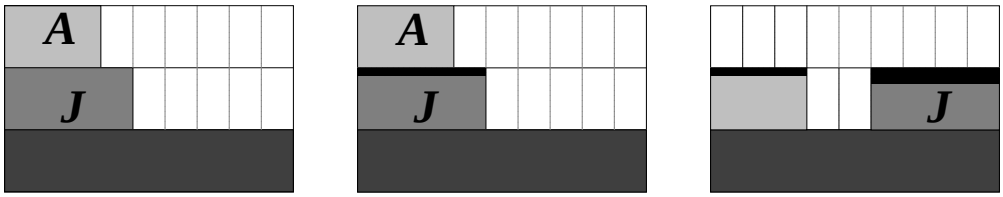
Figure 21: Comparing continuous and discrete scheduling.

migration (BGS), (ii) backfilling gang-scheduling with migration (MBGS), (iii) gang-scheduling without migration (GS), and (iv) gang-scheduling with migration (MGS). We also show the percentage improvement in job slowdown from applying migration to gang-scheduling and backfilling gang-scheduling. Those results are from the best case for each policy: 0 cost and unrestricted number of migrated tasks, with an MPL of 5.

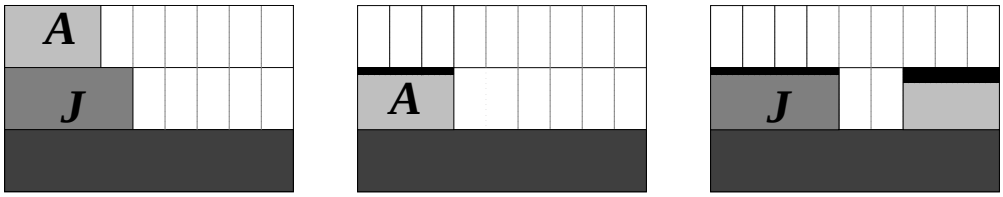
We can see an improvement from the use of migration throughout the range of workloads, for both gang-scheduling and backfilling gang-scheduling. We also note that the improvement is larger for mid-to-high utilizations between 70 and 90%. Improvements for low utilization are less because the system is not fully stressed, and the matrix is relatively empty. Therefore, there are not enough jobs to fill all the time-slices, and expanding without migration is easy. At very high loads, the matrix is already very full and migration accomplishes less than at mid-range utilizations. Improvements for backfilling gang-scheduling are not as impressive as for gang-scheduling. Backfilling gang-scheduling already does a better job of filling holes in the matrix, and therefore the potential benefit from migration is less. With backfilling gang-scheduling the best improvement is 45% at a utilization of 94%, whereas with gang-scheduling we observe benefits as high as 90%, at utilization of 88%.

We note that the maximum utilization with gang-scheduling increases from 85% without migration to 94% with migration. Maximum utilization for backfilling gang-scheduling increases from 95% to 97% with migration. Migration is a mechanism that significantly improves the performance of gang-scheduling without the need for job execution time estimates. However, it is not as effective as backfilling in improving plain gang-scheduling. The combination of backfilling and migration results in the best overall gang-scheduling system.

Figure 24 shows average job slowdown and average job wait time as a function of the parameter Q, the maximum number of task that can be migrated in any time slice. We consider two representative workloads, 2 and 5, since they define the bounds of the operating range of interest. Beyond workload 5, the system reaches unacceptable slowdowns for gang-scheduling, and below workload 2 there is little benefit from migration. We note that migration can significantly improve the performance of gang-scheduling even with as little as 64 tasks migrated. (Note that the case without migration is represented by the parameter $Q = 0$ for number of migrated tasks.) We also observe a monotonic improvement in slowdown and wait time with the number of migrated tasks, for both gang-scheduling and backfilling gang-scheduling. Even with migration costs as high as 30 seconds, or 15% of the time slice, we still observe benefit from migration.



(a) Migration option 1: move blocking jobs in destination row



(b) Migration option 2: move job to different set of nodes in destination row

Figure 22: The two migration options.

work load	backfilling gang-scheduling					gang-scheduling				
	BGS		MBGS		% s better	GS		MGS		% s better
	ρ	s	ρ	s		ρ	s	ρ	s	
0	0.55	2.5	0.55	2.1	19.2%	0.55	3.9	0.55	2.6	33.7%
1	0.61	3.2	0.61	2.5	23.9%	0.61	7.0	0.61	4.0	42.5%
2	0.66	3.8	0.66	2.9	24.8%	0.66	18.8	0.66	6.9	63.4%
3	0.72	6.5	0.72	3.7	43.1%	0.72	44.8	0.72	13.5	69.9%
4	0.77	8.0	0.77	5.1	36.6%	0.78	125.6	0.77	29.4	76.6%
5	0.83	11.9	0.83	7.6	36.2%	0.83	405.6	0.83	54.4	86.6%
6	0.89	22.4	0.88	11.0	50.8%	0.86	1738.0	0.88	134.2	92.3%
7	0.94	34.9	0.94	20.9	40.2%	0.86	4147.7	0.94	399.3	90.4%
8	0.96	67.9	0.98	56.8	16.4%	0.86	5941.5	0.97	1609.9	72.9%

Table 1: Percentage improvements from migration.

Most of the benefit of migration is accomplished at $Q = 64$ migrated tasks, and we choose that value for further comparisons. Finally, we note that the behaviors of wait time and slowdown follow approximately the same trends. Thus, for the next analysis we focus on slowdown.

Figure 25 compares loss of capacity, slowdown, and wait time for all four time-sharing strategies: GS, BGS, MGS and MBGS. Results shown are for MPL of 5, $\Phi = 0.2$, and (for MGS and MBGS) a migration cost of 10 seconds (5% of the time-slice). We observe that MBGS is always better than the other strategies, for all three performance parameters and across the spectrum of utilization. Correspondingly, GS is always worse than the other strategies. The relative behavior of BGS and MGS deserves a more detailed discussion.

With respect to loss of capacity, MGS is consistently better than BGS. MGS can drive utilization up to 98% while BGS saturates at 96%. With respect to wait time, BGS is consistently better than MGS. Quantitatively, the wait time with MGS is 50-100% larger than with BGS throughout the range of utilizations. With respect to slowdown, we observe that BGS is always better than MGS and that the difference increases with utilization. For workload 5, the difference is as high as a factor of 5. At first, it is not intuitive that BGS can be so much better than MGS in the light of the loss of capacity and wait time results. The explanation

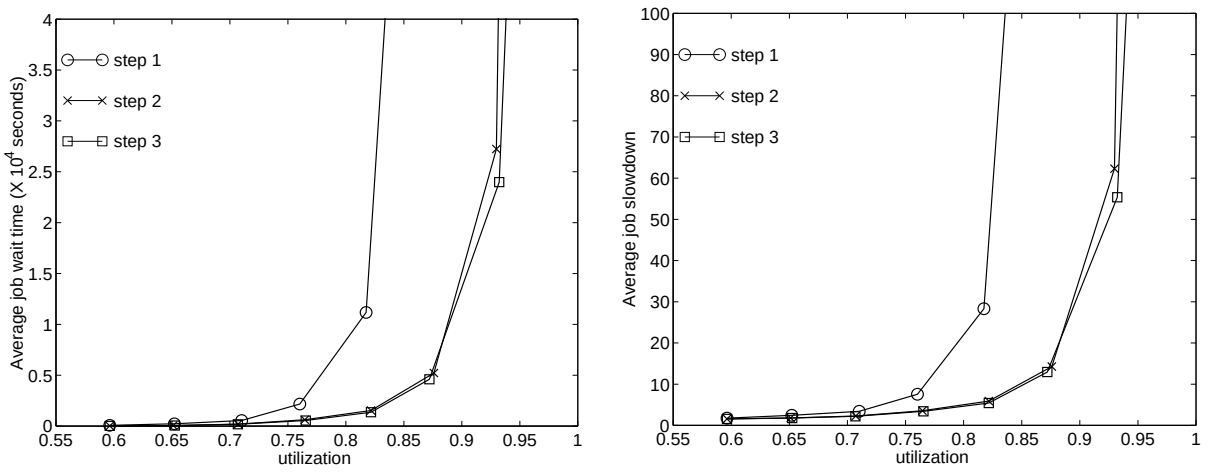


Figure 23: Comparision of Configurations for MGS: GS, GS + FillMatrix-2, GS+CollapseMatrix-2+Schedule-2+FillMatrix-2

is that BGS favors short-running jobs when backfilling, thus reducing the average job slowdown. To verify that, we further investigated the behavior of MGS and BGS in two different classes of jobs: one class is comprised of the jobs with running time shorter than the median (680 seconds) and the other class of jobs with running time longer than or equal to the median. For the shorter jobs, slowdown with BGS and MGS are 18.9 and 104.8, respectively. On the other hand, for the longer jobs, slowdown with BGS and MGS are 4.8 and 4.1, respectively. These results confirm that BGS favors short running jobs. We note that the penalty for longer jobs in BGS (as compared to MGS) is very small, whereas the benefit for shorter jobs is quite significant.

We emphasize that the strategy that combines all techniques (gang-scheduling, backfilling, and migration), that is, MBGS provides the best results. In particular, it can drive utilization higher than MGS, and achieves better slow down and wait times than BGS. Quantitatively, wait times with MBGS are 2 to 3 times shorter than with BGS, and slowdown is 1.5 to 2 times smaller.

7 Conclusions

This paper has reviewed several techniques we developed to enhance job scheduling for large parallel systems. We started with an analysis of two commonly used strategies: backfilling and gang-scheduling. We showed how the two could be combined into a backfilling gang-scheduling (BGS) strategy that is always superior to its two components when the context switch overhead is kept low. With BGS, we observe a monotonic improvement in job slowdown, job wait time, and maximum system utilization with the multiprogramming level. We have also demonstrated the importance of continuous scheduling when time-sharing techniques are used.

Further improvement in scheduling efficacy can be accomplished with the introduction of migration. We have demonstrated that both plain gang-scheduling and backfilling gang-scheduling benefit from migration. The scheduling strategy that incorporates all our techniques: gang-scheduling, backfilling, and migration consistently outperforms the others for average job slow down, job wait time, and loss of capacity. It also achieves the highest system utilization, allowing the system to achieve up to 98% utilization. When a maximum acceptable slowdown of 20 is adopted, the system can achieve 94% utilization.

We have shown that combining techniques such as backfilling and migration with well established gang-

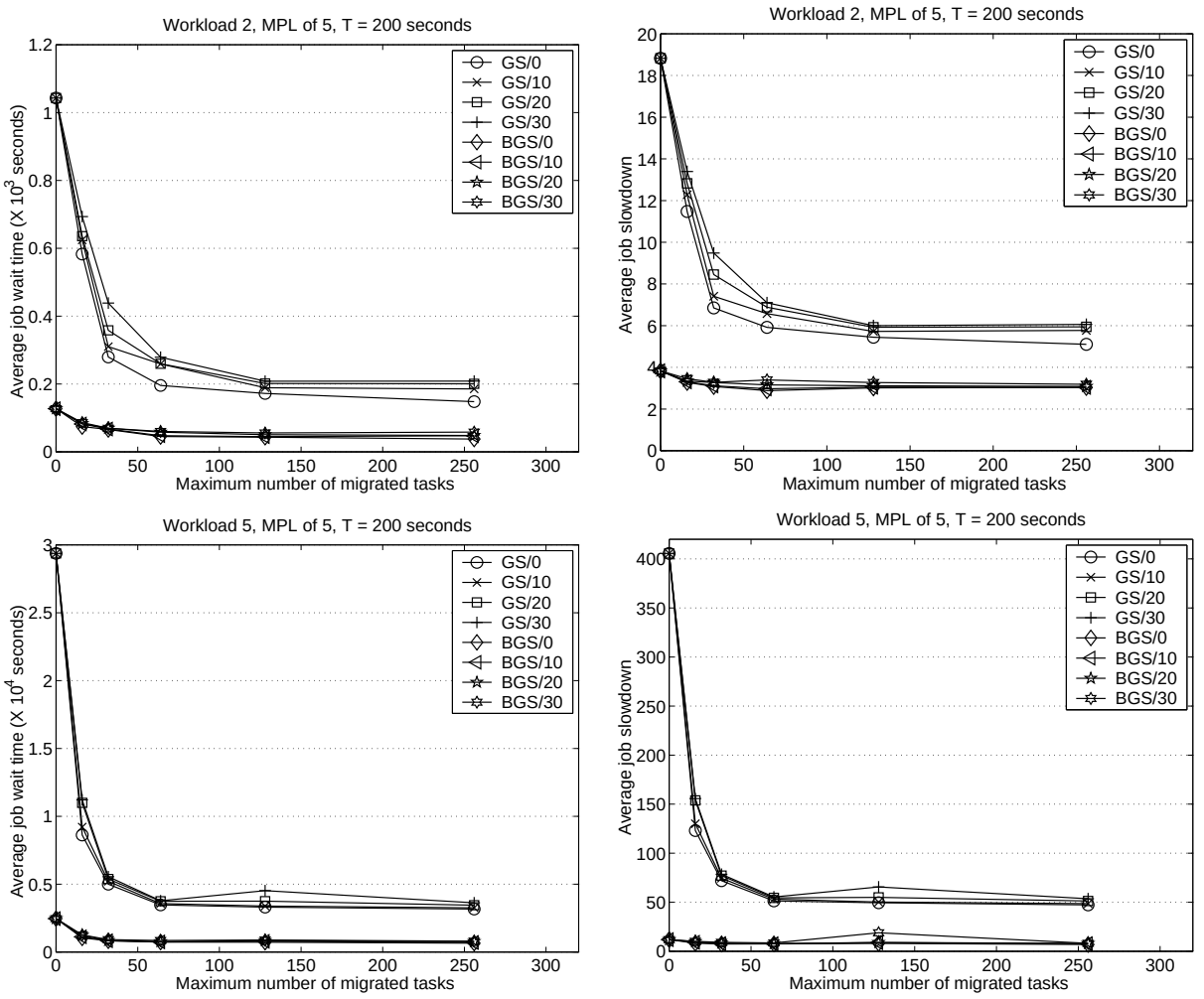


Figure 24: Slowdown and wait time as a function of number of migrated tasks. Each line is for different migration cost

scheduling strategies can improve system performance significantly. A backfilling gang-scheduling system has been successfully deployed in the multi-Teraflop ASCI Blue and White machines. The new scheduling system in those machines is expected to improve utlization, reduce job wait times, and overall enhance the execution of large jobs.

References

[1] J. Casas, D. L. Clark, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole. **MPVM: A Migration Transparent Version of PVM**. *Usenix Computing Systems*, 8(2):171–216, 1995.

[2] 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.

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

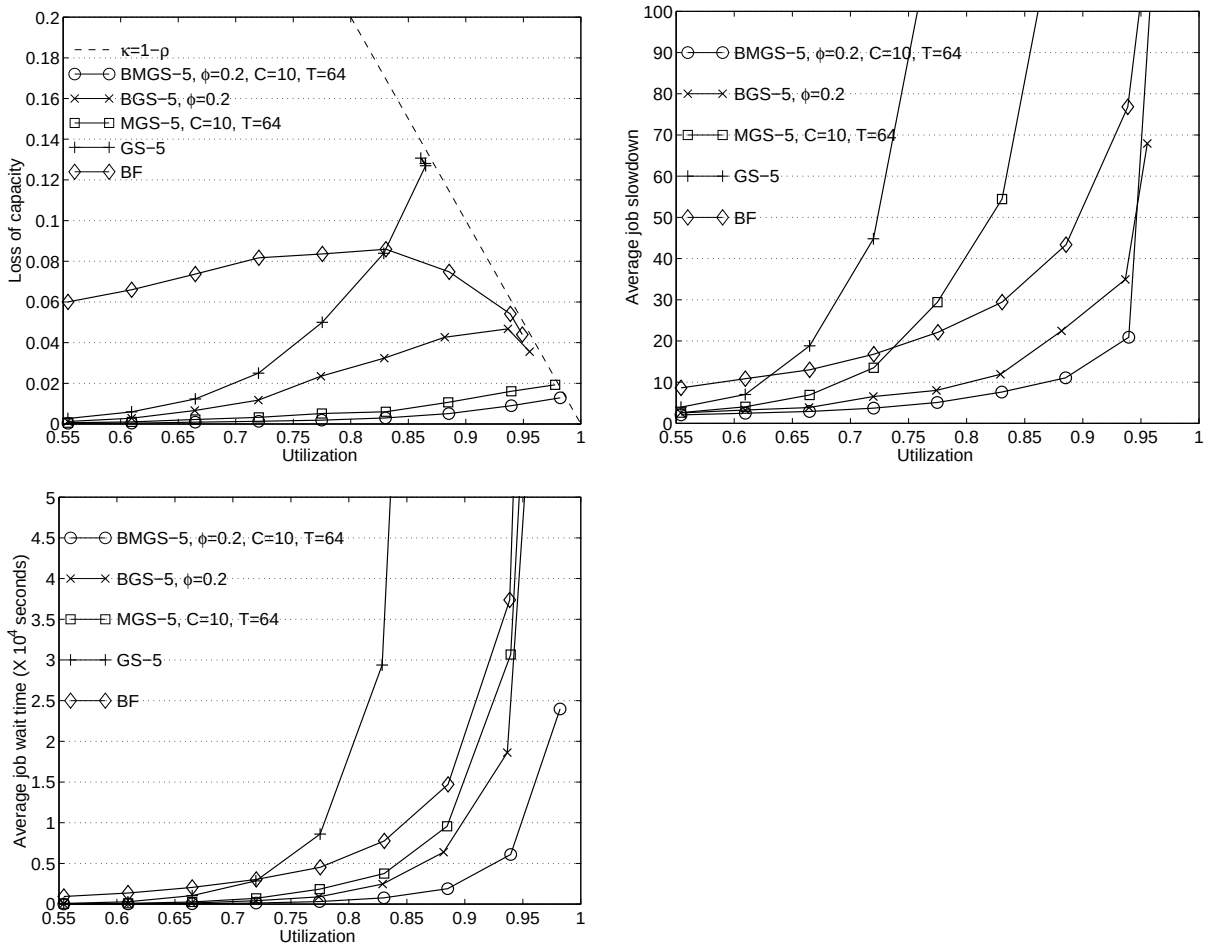


Figure 25: Slowdown and wait time as a function of utilization for GS, MGS, BGS, and MBGS.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] S. Petri and H. Langendörfer. **Load Balancing and Fault Tolerance in Workstation Clusters – Migrating Groups of Communicating Processes**. *Operating Systems Review*, 29(4):25–36, October 1995.

[19] J. Pruyne and M. Livny. **Managing Checkpoints for Parallel Programs**. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop*, volume 1162 of *Lecture Notes in Computer Science*, pages 140–154. Springer, April 1996.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.