An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling and Migration

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 evaluation of the benefits from combining the above techniques. We demonstrate that, under certain conditions, a strategy that combines backfilling, gang-scheduling, and migration is always better than the individual strategies for all quality of service parameters that we consider.

1 Introduction

Large scale parallel machines are essential to meet the needs of demanding applications at supercomputing environments. In that context, 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, 4, 7, 10, 13, 14, 17, 18, 21, 22]. Early strategies used a space-sharing approach, wherein jobs can run concurrently 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.

We analyze three approaches to alleviate the problems with space sharing scheduling. The first is a technique called backfilling [6, 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 [16, 22]. 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.

The third approach is to dynamically migrate tasks of a parallel job. Migration delivers flexibility of adjusting your schedule to avoid fragmentation [3, 4]. 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. 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, this paper analyzes (i) the impact of overestimating job execution times on the effectiveness of backfilling, (ii) a strategy for combining gang-scheduling and backfilling, (iii) the impact of migration in a gang-scheduled system, and (iv) the impact of combining gang-scheduling, migration, and backfilling in one scheduling system.

We 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 after considering the associated overheads, such as context-switch overhead. 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 first-come first-serve (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 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 4 describes gang-scheduling, and the various phases involved in computing a time-sharing schedule. Section 5 demonstrates the significant improvements in performance that can be achieved with time-sharing techniques, particularly when enhanced with backfilling and migration. Finally, Section 6 presents our conclusions and possible directions for future work.

2 Evaluation methodology

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

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 Lawrence Livermore National Laboratory (LLNL), an IBM RS/6000 SP.

Our modeling procedure involves the following steps:

1. First we group the jobs into classes, based on the number of nodes they require for execution. 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. 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 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 overestimation 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 used to define the job finish event.

In this paper, we adopt what we call the Φ model of overestimation. 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 distribution is shown in Figure 1. It has been shown to represent well actual job behavior in large systems [6]. 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 $(1 - \Phi)/(1 - y)$.

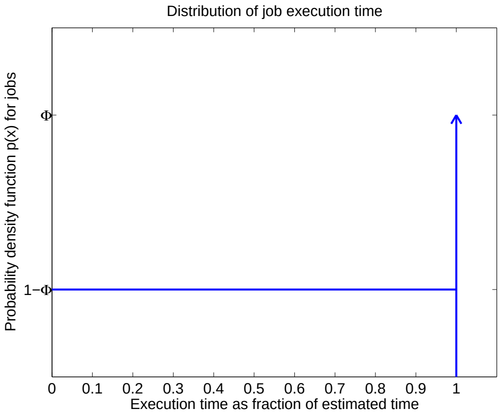
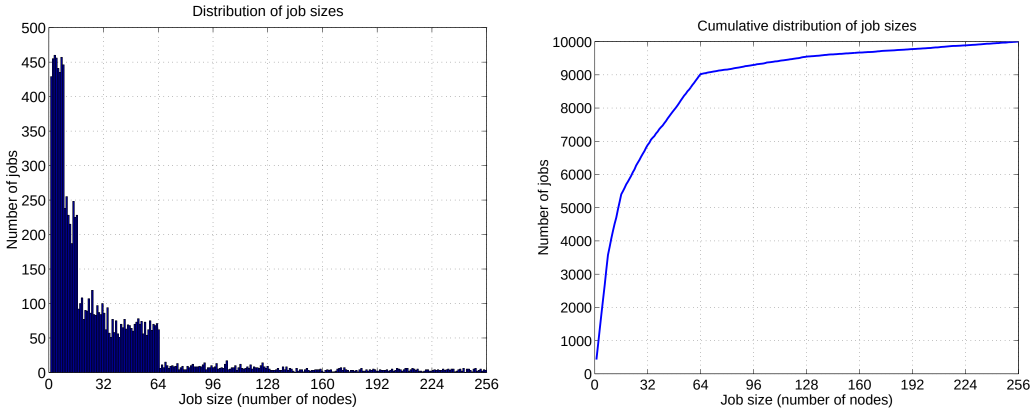


Figure 1: The Φ models for overestimation.

2.2 Workload characteristics

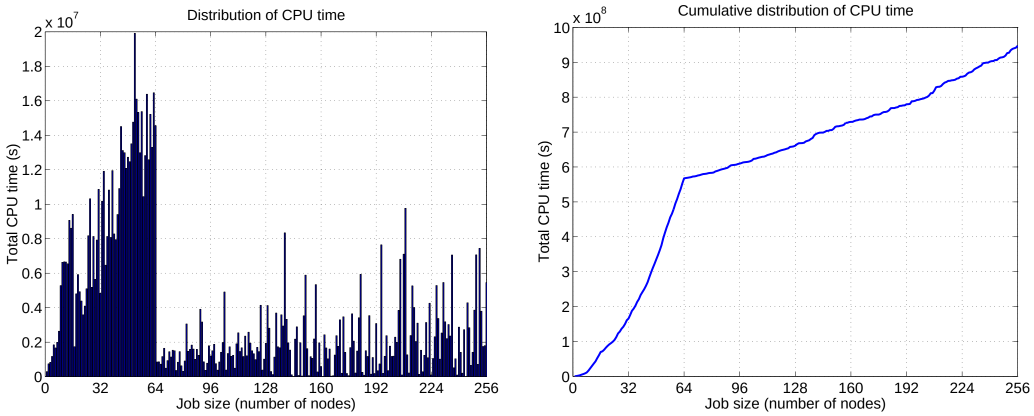
The baseline workload is the synthetic workload generated from the parameters directly extracted from the actual ASCI Blue-Pacific workload. It consists of 10,000 jobs, varying in size from 1 to 256 nodes, in a system with a total of 320 nodes. Some characteristics of this workload are shown in Figures 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 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.)

In addition to the baseline workload of Figures 2 and 3 we generate 8 additional workloads, of 10,000 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



(a) (b)

Figure 2: Workload characteristics: distribution of job sizes.



(a) (b)

Figure 3: Workload characteristics: distribution of cpu time.

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. 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 [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

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

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 to keep track of what happens in each time-slice. Please note that here one time-slice is not exactly equal to one row in the matrix since the last time-slice could be shorter than a row in time due to the fact that a scheduling event could happen in the middle of a row. Let s_i be the number of time slices between scheduling event i and scheduling event $i + 1$. We can then define

$$\kappa = \frac{\sum_{i=1}^{2m-1} \left[e_i(\psi_{i+1} - \psi_i) + T \times CS \times s_i \times n_i\right]\delta_i}{t_m^f \times N} \tag{3}$$

where

- T is the duration of one row in the matrix;
- CS is the context-switch overhead (as a fraction of T);
- n_i is the number of occupied nodes between scheduling events i and $i+1$, more specifically, $n_i + e_i = N$.

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, $\kappa = 1 - \rho_{max}$.

3 The impact of overestimation on backfilling

Backfilling is a space-sharing optimization technique. With backfilling, we can bypass the priority order imposed by the job queuing 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.

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 [6] that there is not necessarily 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 for the Φ model. 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 Φ. We observe very little impact of overestimation. For

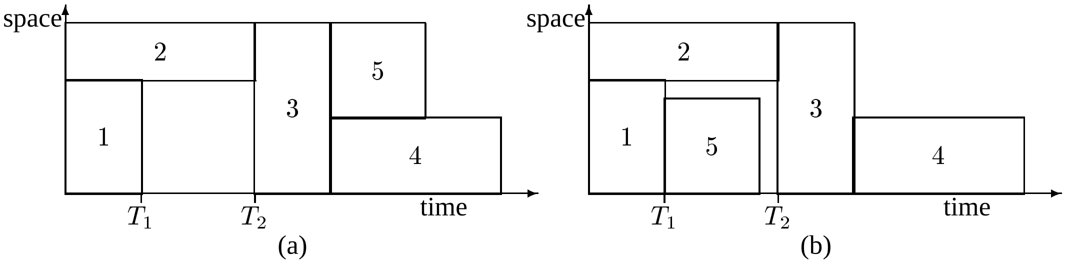


Figure 4: FCFS policy without (a) and with (b) backfilling. Job numbers correspond to their position in the priority queue.

utilization up to $\rho = 0.90$, overestimation actually helps in reducing job slowdown. However, 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: 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.

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. 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 6 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.

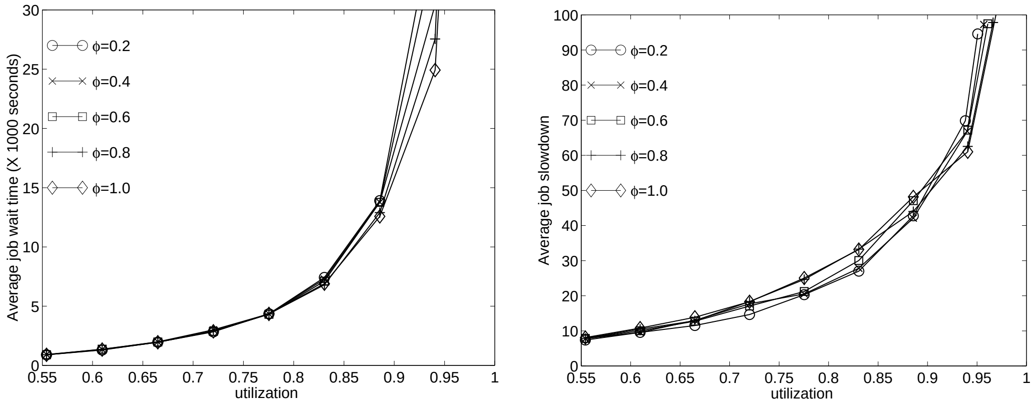
Our findings are in agreement with the work described in [19]. In that paper, the authors describe mechanisms to more accurately predict job execution times, based on historical data. They find that more accurate estimates of job execution time lead to more accurate estimates of wait time. The authors 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, 9].

4 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, 4, 8, 11, 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 (also called the *Ousterhout matrix*) that defines the processors 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 (J_1 and J_2) and several smaller jobs (J_3, J_4, J_5, J_6). All tasks of a parallel job are always coscheduled to run concurrently, which means that all tasks of a job should be assigned to the same row in the matrix. 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 jobs can appear in multiple rows (such as jobs J_4 and J_5).

4.1 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 7 requires, in principle, only adding one more row to the Ousterhout matrix. However, 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



(a) Wait time

(b) Slow down

Figure 5: Average job slowdown and wait time for backfilling under Φ model of overestimation.

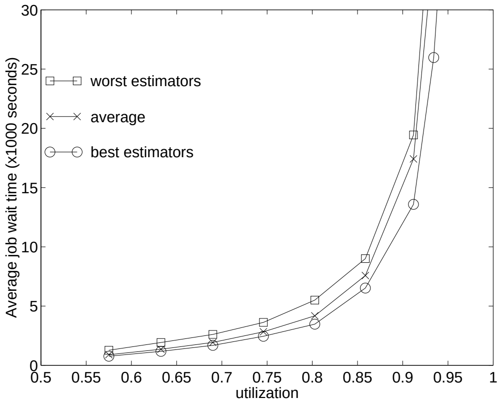


Figure 6: The impact of good estimation from a user perspective for the Φ model of overestimation.

system with multiprogramming level of 1 reverts back to a space-sharing system.

In our particular simulation of gang-scheduling, we make the following assumptions and scheduling strategies:

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. More specifically, in our simulations, we use MPL ≤ 5, and CS (context switch overhead fraction) $\leq 5\%$.

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 7.)

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 4.2.

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

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

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

in [4, 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.

4.2 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 particular scheduling strategy and the details will be presented as we discuss each strategy.

5 Scheduling strategies

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 IBM 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 overhead 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. Typically, context switch overhead values should be between 0 to 5% of time slice.

5.1 Gang-scheduling (GS)

The first scheduling strategy we analyze is plain gang-scheduling (GS). This strategy is described in Section 4. For gang-scheduling, we implement the four scheduling steps of Section 4.2 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 job
```

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 i+1
    for each job in row i
      if it can be moved to row j, then move job
```

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 pack the most jobs in the least number of rows.

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 columns 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 each row in matrix,
      if job can be replicated in same columns
        do it and break
} while matrix changes
```

The algorithm attempts to replicate each job at least once (In the algorithm, once a chance of replicating a job is found, we stop looking for more chances of replicating the same job, but instead, we start other jobs) , although some jobs can be replicated multiple times. We go through the jobs in starting time order, but other ordering policies can be applied.

5.2 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 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 3, 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.

To verify that the assumption that overestimation of job execution times indeed do not impact overall system performance, we experimented with various values of Φ. Results are shown in Figure 8. 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.

5.3 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 multiprogramming level 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.

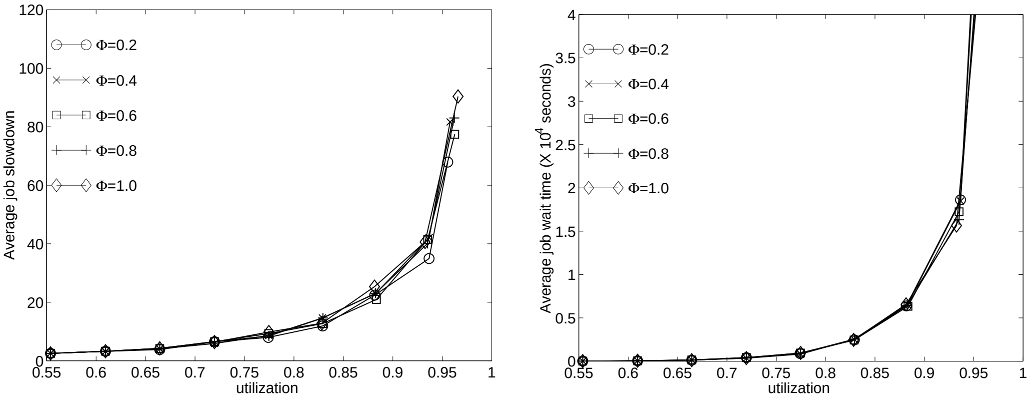


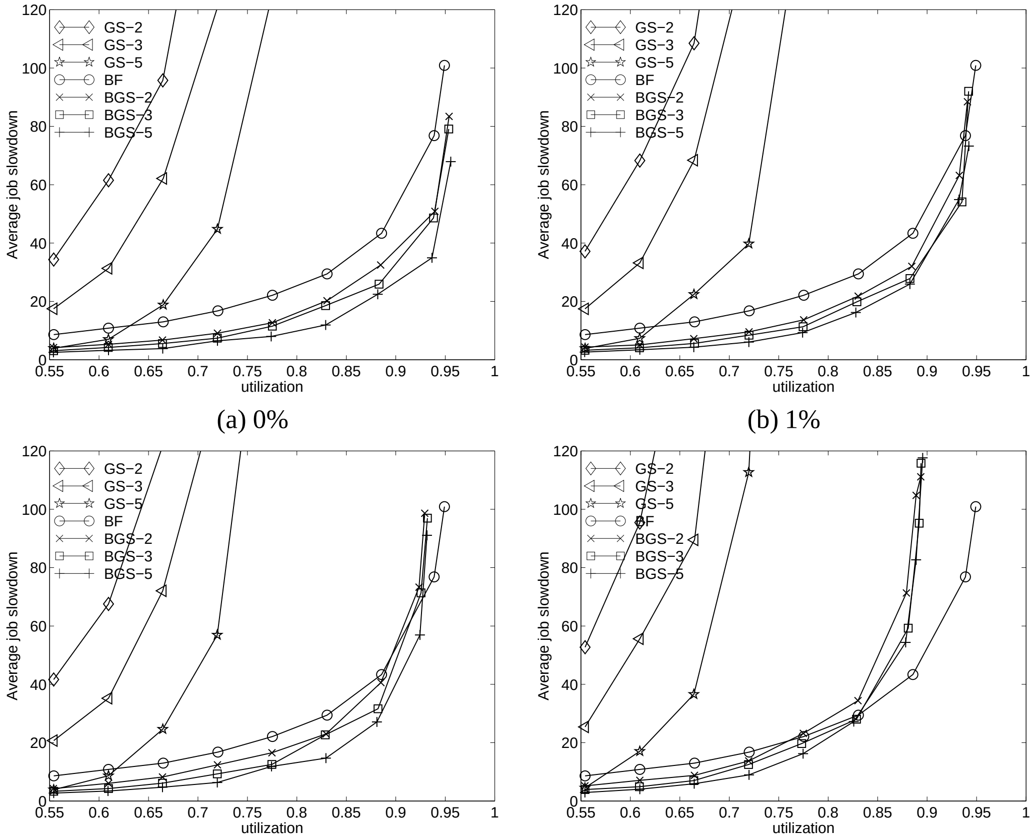
Figure 8: Average job wait time and slow down for BGS (best) with Φ model of overestimation.

Figure 9 shows the average job slow down for all our seven configurations. Each plot ((a), (b), (c), and (d)) is for a different value of context switch overhead as a fraction of time slice. The time slice is 200 seconds. If we look only at the case of zero 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. 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. 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 $\rho = 0.87$.

At all combinations of context switch overhead and utilization, BGS outperforms GS with the same multiprogramming level. 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.

Figure 10 shows the average 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 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.

We further analyze the scheduling strategies by comparing the behavior of the system for large and small jobs. (As defined in Section 2.2, a 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 11, when a 0% context switch overhead is used. With respect to slowdown, we observe that, BGS-5 always per-



(a) 0%

(b) 1%

(c) 2%

(d) 5%

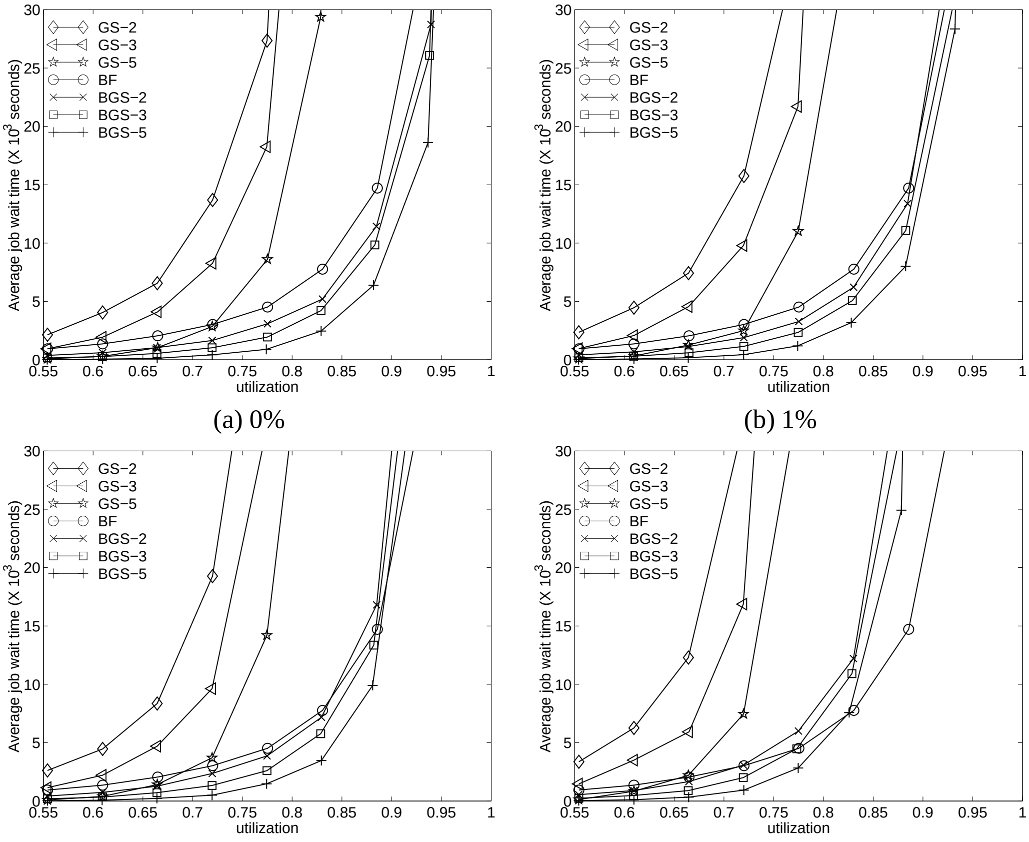
Figure 9: Average job slowdown for four different values of context switch overhead.

forms better than BF for either large or small jobs. 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. 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 more equal to all kinds of jobs.

Whereas Figures 9 through 11 report performance from a user's perspective, we now turn our attention to the system's perspective. Figure 12 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 $\kappa = 1 - \rho_{max}$, which is indicated by the dashed line in Figure 12. 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 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 summarize our observations, we have shown that the combined strategy of backfilling with gang-scheduling (BGS) can consistently outperforms the other strategies



(a) 0%

(b) 1%

(c) 2%

(d) 5%

Figure 10: Average job wait times for four different values of context switch overhead.

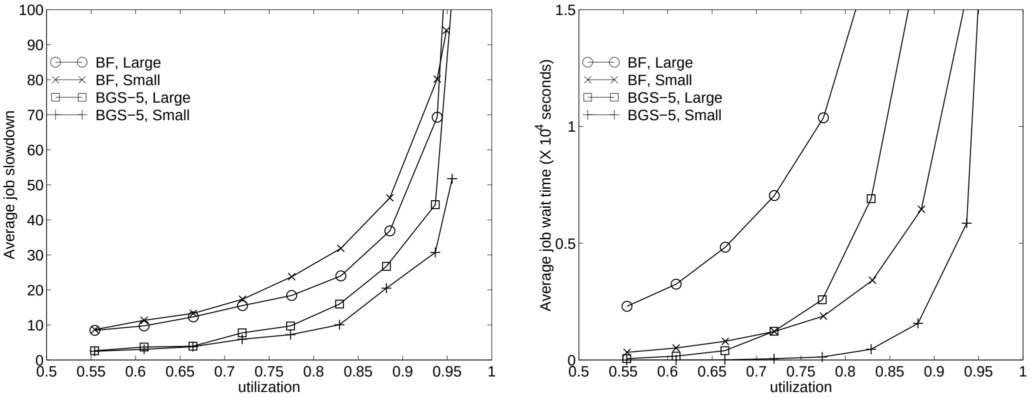
(backfilling and gang-scheduling separately) from the perspectives of responsiveness, slow down, fairness, and utilization. For BGS to realize this advantage, context switch cost must be kept low. We have shown BGS to be superior to BF over the entire spectrum of workloads when the context switch overhead is 1% or less of the time slice.

5.4 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 (not just on the same columns). 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 free nodes):

- *Option 1*: We migrate the jobs in row p that execute on the CPUs where the processes of A reside, to make space for A in row p. This is shown pictorially in figure 13 where 3 processes of job J in row 2 occupy the same columns as job A in row 1. Job J is migrated to 4 other processes in the same row and job A is replicated in this row. Consequently when we move from row 1 to row 2 in the scheduling cycle, job A does not need to be migrated (one-time effort).
- *Option 2*: Instead of migrating job J to make space for A, we can directly migrate job A to those slots in row p that are free. This approach lets other jobs in row p proceed without migration, but the down side is that each time we come to row p, job A incurs migration costs (recurring). This is again shown pictorially in figure 13.

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. 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



(a) slowdown

(b) wait time

Figure 11: Slowdown and wait time for large and small jobs.

and restart each 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 × time) wasted during this process is the overhead for the migration operation.

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 13(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

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

The migration for option 2 is illustrated in Figure 13(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 $C \times |A|$.

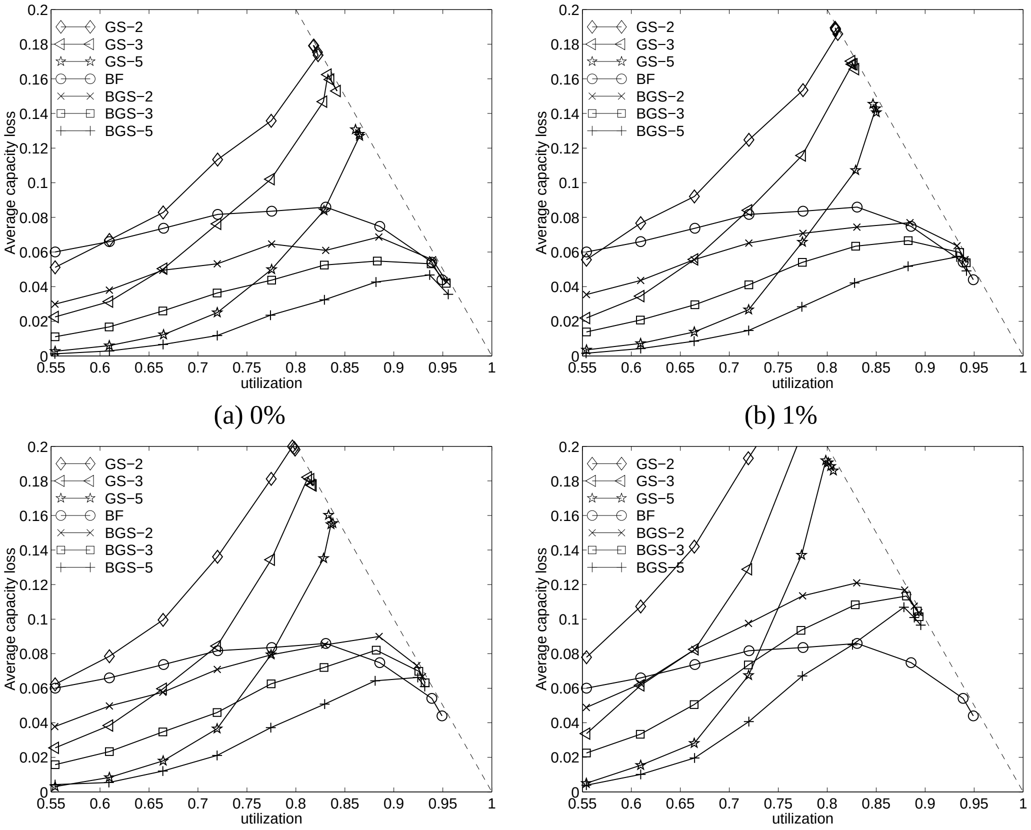
As discussed, migration in the IBM RS/6000 SP requires a checkpoint/restart operation. Although all tasks can perform a checkpoint in parallel, resulting in a C that is independent of job size, there is a limit to the capacity and bandwidth that the file system can accept. Therefore we introduce a parameter 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	Accommodate new jobs after compaction
CollapseMatrix-2	Compaction with migration
Schedule-2	Accommodate new jobs after migration
FillMatrix-1	Replicate jobs without migration
FillMatrix-2	Replicate jobs after migrating destination

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 accommodate new jobs. After we are done accommodating new jobs, we do FillMatrix-1 first because it does not incur 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) 0%

(b) 1%

(c) 2%

(d) 5%

Figure 12: Loss of capacity for BGS, GS, and BF, with different context-switch overheads.

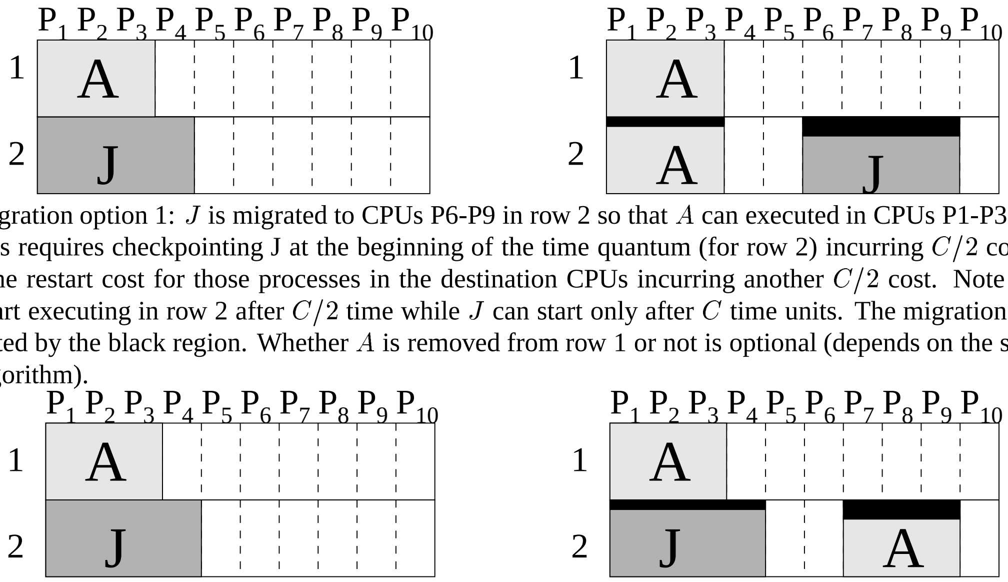
a new target row, we consider the two options described above (option 1 and option 2) and choose the one with the least estimated cost. FillMatrix-2 uses the same algorithm as FillMatrix-1, with the following constraints when deciding to replicate a job in a new row. First, the job must not already be 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.

5.5 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.

5.6 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 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



(a) Migration option 1: J is migrated to CPUs P6-P9 in row 2 so that A can executed in CPUs P1-P3 in row 2. This requires checkpointing J at the beginning of the time quantum (for row 2) incurring $C/2$ cost, and then the restart cost for those processes in the destination CPUs incurring another $C/2$ cost. Note that A can start executing in row 2 after $C/2$ time while J can start only after C time units. The migration cost is indicated by the black region. Whether A is removed from row 1 or not is optional (depends on the steps of the algorithm).

(b) Migration option 2: A is directly migrated to CPUs P7-P9. This requires checkpoint A at the beginning of the time quantum for row 2 (incurring $C/2$ cost), and restarting A in the destination CPUs subsequently (incurring another $C/2$ cost). Even though only A's processes are being migrated at P1-P3, J has to wait for $C/2$ time before it can execute (on all four of its CPUs). A can begin execution after C time units in CPUs P7-P9. The migration cost is indicated by the black region. Again, whether A is removed from row 1 or not is optional (depends on the steps of the algorithm). If it is not removed, a recurring migration cost is incurred each time we transition from row 1 to row 2 in the schedule.

Figure 13: The two migration options.

for each policy: zero 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 50% at a utilization of 89%, whereas with gang-scheduling we observe benefits as high as 92%, at utilization of 88%.

We note that the maximum utilization with gang-scheduling increases from 86% without migration to 94% with migration. Maximum utilization for backfilling gang-scheduling increases from 96% to 98% 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 14 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. Each line is for a different combination of scheduling mechanism and migration cost (*e.g.*, BGS/10 represents backfilling gang-scheduling with migration cost of 10 seconds. The time slice is 200 seconds). 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 mono-

Table 1: Percentage improvements from migration.

work	backfilling gang-scheduling					gang-scheduling				
load	BGS		MBGS		% s	GS		MGS		% s
	ρ	s	ρ	s	better	ρ	s	ρ	s	better
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%

tonic 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 a benefit 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 15 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 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 MBGS, which combines all techniques (gang-scheduling, backfilling, and migration), 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.

6 Conclusions

This paper has reviewed several techniques 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.

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 high-

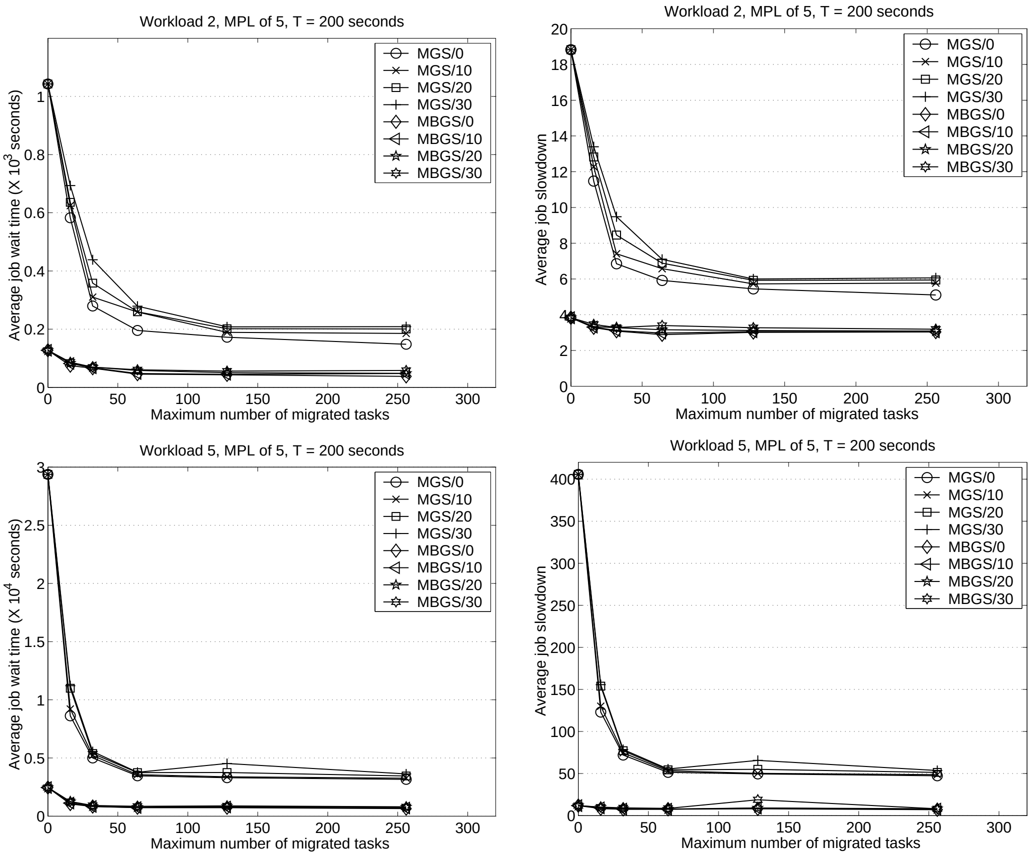


Figure 14: Slowdown and wait time as a function of number of migrated tasks.

est system utilization, allowing the system to reach up to 98% utilization. When a maximum acceptable slowdown of 20 is adopted, the system can achieve 94% utilization.

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. **Packing schemes for gang scheduling**. In *Job Scheduling Strategies for Parallel Processing, IPPS'96 Workshop*, pages 89–110, March 1996. LNCS 1162.

[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.

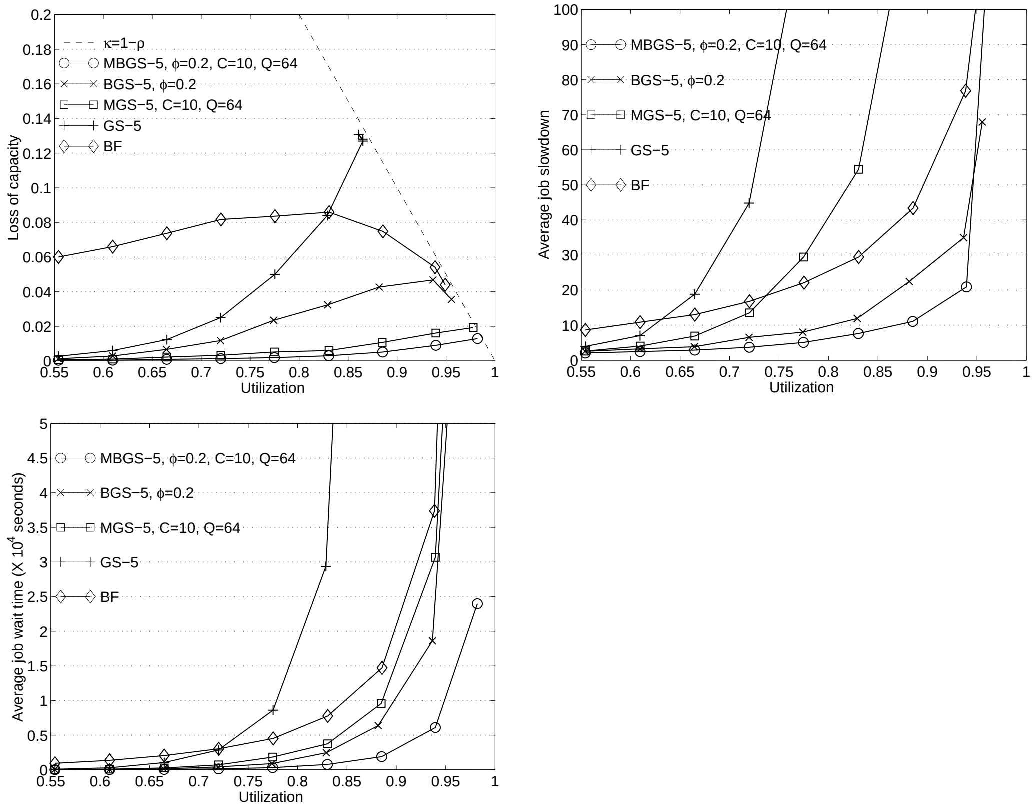


Figure 15: Average loss of capacity, job slowdown, and job wait time as a function of utilization for GS, MGS, BGS, and MBGS.

[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. K. Ousterhout. **Scheduling Techniques for Concurrent Systems**. In *Third International Confer-*

ence 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.

[22] B. B. Zhou, R. P. Brent, C. W. Jonhson, and D. Walsh. **Job Re-packing for Enhancing the Performance of Gang Scheduling**. In *Job Scheduling Strategies for Parallel Processing, IPPS'99 Workshop*, pages 129–143, April 1999. LNCS 1659.