Towards Provision of Quality of Service Guarantees in Job Scheduling

MOHAMMAD ISLAM, PAVAN BALAJI, P. SADAYAPPAN AND DHABALESWAR K. PANDA

Technical Report
OSU-CISRC-5/04-TR24

Towards Provision of Quality of Service Guarantees in Job Scheduling*

Mohammad Islam Pavan Balaji P. Sadayappan D. K. Panda

Computer and Information Science,
The Ohio State University,
2015 Neil Avenue,
Columbus, OH43210
{islammo, balaji, saday, panda}@cis.ohio-state.edu

Abstract

Considerable research has focused on the problem of scheduling dynamically arriving independent parallel jobs on a given set of resources. There has also been some recent work in the direction of providing differentiated service to different classes of jobs using statically or dynamically calculated priorities assigned to the jobs. However, the potential and usability of a Quality of Service based scheme has not been much studied. In this paper, we extend a previously proposed scheme (QoPS) to provide Quality of Service to submitted jobs; we propose extensions to the algorithm in multiple aspects: (i) studying the effect of user tolerance towards missed deadlines on the overall profit attainable by the supercomputer center, (ii) providing artificial slack to some jobs to maximize the overall profit and (iii) utilizing a Kill-and-Restart mechanism to further improve the profit attainable.

Keywords: *QoS, Parallel Job Scheduling, Real-Time deadlines*

1 Introduction

A lot of research has focused on the problem of scheduling dynamically arriving independent parallel jobs on a given set of resources. The metrics evaluated include system metrics such as the system utilization, throughput [**?**, **?**], etc., and user metrics such as the turnaround time, wait time [**?**, **?**, **?**, **?**, **?**, **?**], etc. Recently, there has been some work in the direction of providing differentiated service to different jobs. The schemes that provide differentiation can be classified into two broad categories.

The first category comprises of approaches that provide "best effort" relative prioritization for individual jobs or different classes of jobs. Such prioritization may either be assigned statically to the jobs (e.g., jobs from a group of users might be given a higher priority compared to others), or may dynamically vary during the queue time of the job (e.g., if a job has been waiting in the queue for a long time, its priority is increased). NERSC [**?**] is an example of such a scheduler. NERSC offers different queues which have different costs and priorities: in addition to the normal priority queue, a high priority queue which double the usual charge, and a low priority queue with half the usual charge. Jobs in the high priority queue get priority over the normal queue, until some threshold on the number of serviced jobs is exceeded.

The second category of schemes to provide differentiated service comprises of those which guarantee a certain Quality of Service (QoS) in the turnaround time for the submitted job. With such schemes, the users have an option of specifying the deadline they need with each submitted job. We are unaware of any production job schedulers that implement such a scheme, but our recent work *QoPS* [**?**] is an example of such a scheme. QoPS implements an admission mechanism for incoming jobs, attempting various schedule rearrangements of previously admitted jobs to make a decision on whether the requested deadline is achievable without violating the deadlines provided for the other admitted jobs. If achievable, QoPS admits the job and guarantees the requested deadline to the job.

The overall issue of providing QoS for job scheduling can be viewed in terms of two related aspects:

- **Cost Model for Jobs**: The quicker the sought response time, the larger should be the charge. The charge will generally be a function of many factors, including the resources used and the load on the system.
- **Job Scheduling with Response-time Guarantees**: If jobs are charged differently depending on the response time demanded by the user, the system must provide guarantees in the completion time.

The QoPS scheduling algorithm presented at last year's Job Scheduling Workshop addressed only the second as-

*This research is supported in part by NSF grants #CCR-0204429 and #EIA-9986052

pect, i.e., how to implement admission control for deadline-based scheduling of parallel jobs, effectively exploiting the deadline flexibility among previously admitted jobs in order to maximize system utilization without violating deadlines of any admitted jobs. But a scheduling mechanism like QoPS, that implements admission control and provides deadline guarantees to admitted jobs, will be ineffective in providing effective differentiated services to users unless a suitable charging model is imposed. If the charge differential for provision of rapid versus slow turn-around for jobs is relatively small, all users might submit jobs demanding very tight deadlines, so that no effective differentiation will be achieved between urgent and non-urgent jobs submitted to the system. In this paper, we propose extensions to the QoPS algorithm, that are motivated by considerations with respect to the job charging model. We perform evaluations to characterize trends with respect to a QoS cost-component and a resource-usage cost component. The issue of how to effectively combine the QoS component and resource-usage component to form the overall charging function is a difficult and open problem that is beyond the scope of this paper. Here, we study two separate cost components for QoS and resource usage (as explained later in Section 3).

The enhancements to QoS-based scheduling that we present in this paper are summarized below:

- Feedback on earliest feasible completion of unadmitted jobs: In our previous trace-based evaluation of QoPS, we associated deadlines with each job, and only admitted jobs whose deadlines could be met without violating any prior commitments. Thus, if the requested deadline of a job was not satisfiable, it was simply dropped and not further considered for scheduling. In practice, it is likely that in some circumstances users would be willing to accept a slower response time than their original request, while in other situations they would not - they may choose to submit their job at some other center or choose to submit a different job instead. With a basic QoS-based scheduling scheme like QoPS, that implements admission control and deadline guarantees, users of unadmitted jobs with flexibility may need to iteratively resubmit their jobs with looser and looser deadlines until acceptance (or abandonment if the achievable deadline was unacceptably late). We develop a mechanism in QoPS to provide users of inadmissible jobs with feedback on the earliest guaranteeable deadline for the job.

- Modeling User Tolerance: Given the feedback mechanism described above, it is feasible to perform trace-based evaluations of QoS-based scheduling, where some jobs that cannot meet their originally requested deadline are nevertheless re-submitted with a more relaxed deadline. We performed simulations under different assumptions of user tolerance with respect to deadlines. We parameterize our studies with a Tolerance Factor (TF), that specifies the relative increase in response-time that a user is willing to tolerate. The effect on the QoS and resource components of cost are studied as a function of TF. A surprising result is that the total QoS component of revenue of a center does not monotonically increase with increasing tolerance on the part of the user.

- Incorporation of Artificial Slack: Although giving a job its best possible deadline would maximize the revenue achievable from that job, this might result in a tight schedule for the rest of the jobs causing many later arriving (and potentially more urgent jobs) to be dropped. On the other hand, providing an artificial slack to some of the jobs (which do not provide too much revenue) might result in later arriving urgent jobs to be admitted, causing an overall increase in the supercomputer center revenue. We study the effect of different degrees of artificial slack for various assumed tolerance factors.

- Enabling Kill-and-Restart: Some supercomputer centers implement mechanisms for Kill-and-Restart, where a running job can be killed to enable a different job to be started. Later, the killed job is re-started, but from scratch. We evaluate whether such a mechanism can be utilized to improve the overall profit in the supercomputer centers in the QoS-based scheduling context.

The remaining part of the paper is organized as follows. In Section 2, we provide some brief background about this work and other related work. Section 3 deals with the current cost model used by current supercomputer centers, our proposed cost model and its various components. In Section 4, we discuss the feedback based QoPS algorithm developed to provide the best possible deadline achievable by the job. Section 5 deals with the experimental setup we used in our simulation test-bed. We describe the impacts of user-tolerance and various approaches to negate the negative impacts in Sections 6, 7 and 8 and present some concluding remarks and possible future work in Section 9.

2 Background and Related Work

In this section, we provide some background information about this and other related work. In particular, we discuss our previous work, the QoPS scheduling algorithm (Quality of Service for Parallel Job Scheduling) [**?**]. Two other related scheduling algorithms, namely the Slack-Based scheduler by Feitelson et al., and the Real-Time scheduler by Ramamritham et al., are discussed in [**?**, **?**] and omitted here due to space constraints.

2.1 QoPS Algorithm

The QoPS algorithm, standing for **Qo**S for **P**arallel Job **S**cheduling, provides Quality of Service based scheduling for independent parallel jobs. For dynamic systems with more that one processor, a polynomial-time optimal scheduling algorithm does not exist [**?**, **?**, **?**]. The QoPS scheduling algorithm uses a heuristic approach to find feasible schedules for the jobs.

The scheduler considers a system where each job arrives with a corresponding completion time deadline requirement. When each job arrives, the QoPS scheduler attempts to find a feasible schedule for the newly arrived job. A schedule is said to be feasible if it does not violate the deadline constraint for any job in the schedule, including the newly arrived job. However, it does allow the flexibility of reordering the jobs in any order as long as the resultant schedule remains feasible. Figure 1 presents the pseudo code for the QoPS scheduling algorithm.

The QoPS scheduler allows flexibility in the order in which jobs are considered for scheduling. The amount of flexibility offered is determined by the K-factor denoted in the pseudo code illustrated by Figure 1. Suppose jobs J_1, J_2, ..., J_N are the jobs which are currently in the schedule but not yet started. When a new job arrives, the scheduler considers $log_2(N)$ points in time where its insertion into the schedule is attempted, corresponding to the reserved start-times of jobs {0, N/2, 3N/4, ... } respectively, where *N* is the number of jobs currently in the schedule. Considering the first attempted insertion point (corresponding to job 0), we start by removing all the jobs from the schedule and placing them in a temporary list (TL). The newly arrived job is also added to TL. TL is then sorted according to some heuristic function (such as laxity first, earliest deadline first, etc). Finally, we try to place the jobs in the reservation schedule in that sorted order. For the second choice of insertion point, we do not start with an empty schedule. Instead, we only remove the latter N/2 jobs in the original schedule, chosen in scheduled start time order, place them in the temporary list TL, and sort this temporary list (based on the heuristic function). We then create a reservation for the newly arrived job, and finally generate reservations for the remaining N/2 jobs in the order specified by TL. For the third choice of insertion point, we only remove the last N/4 of the reserved jobs from the reservation schedule. Thus, at most $log_2(N)$ alternative insertion points are considered when a new job arrives.

For each insertion point considered for a newly arrived job, the algorithm tries to schedule the jobs based on the ordering in the temporary list TL. If a job misses its deadline, this job is considered as a critical job and is pushed to the head of the list (thus altering the order in TL). This altering of the temporary schedule is allowed at most 'K' times; after this the scheduler decides that the new job cannot be scheduled while maintaining the deadline for all of the already

Checking the admissibility of Job J into an existing profile of size N:

Original_QoPS(Job J):

```
A. For each time slot ts in position (0, N/2, 3N/4, 7N/8, ...)
     starting from current time
     1. Remove all the waiting jobs from position ts to the
          end of the profile and place them into a Temporary
          List including the new job J
     2. Sort the Temporary List using the heuristic function
     3. Set the Violation Count = 0
     4. For each job Jc in the Temporary List
          i. Add job Jc into existing schedule
          ii. if(there is a deadline violation for job Jc at slot T) then
               a. Violation Count =  Violation Count + 1
               b. If Violation Count  > K-FACTOR break
               c. Remove all the jobs from the schedule of position
                    mid(T + ts) to position T and add them into
                    Temporary List
               d. Sort Temporary List again using the same heuristic
               e. Add the failed job Jc into the top of Temporary List
                    to make sure it will be scheduled at mid (T + ts)
          End if
        End for
     5. If (Violation Count > K-FACTOR) then
             return FAILED.   // Job rejected
        End if
      End for
B. If (Violation Count <= K-FACTOR) then
        return SUCCESS. // Job Accepted
   End if
```

Figure 1. The QoPS Scheduler: Pseudo Code

accepted jobs and rejects it. This results in a time complexity of *O(KNlog N)* for the QoPS scheduling algorithm.

3 Cost Model for Supercomputer Centers

In this section, we discuss the charging model in current supercomputer centers. We extend this cost model to incorporate the QoS capabilities of the QoPS scheduler and present the various components associated with such a charging model.

The charging model in current supercomputer centers is mainly based on the resources utilized by the submitted jobs, and is unrelated to the responsiveness of the system. Thus, a 16-processor job that ran for one hour would be charged for 16 CPU-hours irrespective of whether the turnaround time were one hour or one day. Further, on most systems, even if a user is willing to pay more to get a quicker turn-around on an urgent job, there is no mechanism to facilitate that. As mentioned earlier, some systems such as NERSC [**?**] offer the users some choice by providing three queues: a normal queue, a high priority queue (with double the usual charge) and a low priority queue (with half the usual charge) but don't give users any guarantee on the response time provided. However, the idea of charging the user based on the service or priority assigned to them is still relevant and critical to the practical applicability of a QoS based scheduler.

In this paper, we break the total charge of a submitted job into different components: (i) Resource charge and (ii) QoS charge.

The resource charge is similar to that used by current supercomputer centers and is based on the resources requested by the job. In general, this would depend on the various resources provided by the supercomputer center, e.g., CPU, memory, disk space, etc. In our environment, we only consider the CPU resource, i.e., the resource charge for a job would be equal to the product of the processors requested by the job and the time for which the job runs. This idea can easily be extended to other resources too.

The QoS charge on the other hand depends on the urgency of the job. For example, if two similar jobs are submitted where one of them is urgent while the other is not urgent, the resource charge for both the jobs would be similar, whereas the QoS charge would be much different. Further, if two different jobs request for similar urgencies, they could still have different QoS charges based on the "difficulty" of the supercomputer center in meeting the requested urgency.

In this paper we use the slowdown of the job as the base metric for deciding the QoS charge for the job. The "difficulty" of the supercomputer center in meeting the requested deadline depends on two components: (i) the current load on the system (the number of queued processor seconds) and (ii) the average slowdown of the category to which the job belongs. For example, typically short-wide jobs (ones which use a lot of processors but run for a small amount of time) have huge slowdown values while long-narrow jobs have lesser slowdown values. Thus, we categorize short-wide jobs to be more "difficult" to schedule within the requested slowdown as compared to long-narrow jobs.

We use the following equations for the resource and QoS charges for the jobs:

$$ResourceCharge = Processors \times Runtime$$

$$QoSCharge = \frac{CategorySlowdown}{RequestedSlowdown}$$

4 Feedback based QoPS Algorithm

As discussed earlier, in our previous QoPS algorithm we associated deadlines with each job and only admitted jobs whose deadlines could be met without violating any prior commitments. Thus, a job whose requested deadline can not be satisfied is dropped and not considered further for scheduling. In practice, however, users might not be so strict about their requested deadline. Ideally, the user would have liked to have it done by the requested deadline. If this is not possible, a different and less stringent deadline might also be fine with the user. However, by blindly dropping the job, such possibilities had not been considered previously. With the basic QoPS scheme, users of unadmitted jobs with flexibility may need to iteratively resubmit their jobs with looser and looser deadlines.

In this section, we discuss the provision of providing a feedback mechanism with the QoPS based algorithm which lets the users know the best possible deadline the system can provide to their job. The basic idea of the scheme is to first try to provide the requested deadline to the submitted job. If the system is able to admit the job for this requested deadline, it just accepts the job. On the other hand, if the system fails to admit the job right away, the algorithm tries a fixed number of other possible deadlines (logarithmically refining the search for the best deadline). Based on these trials, it provides the user with what it thinks is the best possible deadline for the job that the system can provide. Figure 2 illustrates the pseudo code for this algorithm.

5 Experimental Setup

Job Scheduling strategies are usually evaluated using real workload traces, such as those available at the Parallel Workload Archive [**?**]. However, real job traces from supercomputer centers have no deadline information. After studying the existing approaches for deadline inclusion in real traces, we decided to go ahead with the approach we had previously used in [**?**].

We start with an existing trace from Feitelson's archive (a 5000-job subset of the CTC trace). Using the completion time of the EASY backfill output, we assign deadlines for the different jobs. A deadline stringency factor determines how tight the deadline is to be set, compared to the

Checking the admissibility of a job J into an existing profile of size N:

```
Updated_QoPS(Job J):
  A. Set status = Original_QoPS(Job J)
  B. If (status =  SUCCESS)
       1. Return requested deadline of Job J
     Else
       1. Set StartPos = Requested  deadline for Job J
       2. Set deadline = INFINITE and schedule
       3. Set EndPos = Completion Time of Job J in the
            above schedule
       4. Loop through the following RETRY_COUNT times
            i. Set deadline of Job J =  (StartPos + EndPos) /2
            ii. Set status = Original_QoPS(Job J)
            iii. If (status = SUCCESS) then
                   Set EndPos = (StartPos + EndPos)/ 2
                 else
                   Set StartPos = (StartPos + EndPos) /2
                 End if
          End Loop
       5. Return deadline of Job J in the final schedule
```

Figure 2. The Feedback based QoPS Scheduler: Pseudo Code

EASY backfill schedule. The deadline for the job is calculated using the formula: Deadline = max (Job's Runtime, (1-S)*(EASY Response time)). It can be seen that for a stringency factor of '0', the deadline would be the same as the completion time generated by the EASY backfill schedule. As 'S' is increased, the deadlines become less flexible making it harder to find a schedule.

Also, we used the duplication approach for varying the load on the supercomputer center (number of jobs submitted). This is done by duplicating randomly selected jobs in the trace. For example, we start with a trace and call this the base trace (load = 1.0). To generate a new trace with load = 1.2, we randomly pick 20% of the jobs in the base trace and introduce extra duplicate jobs at the same points in the trace. The deadline for the newly introduced job is retained as the original job's deadline (from the EASY backfill schedule).

As mentioned in Section 3, the QoS charge of the jobs is based on the slowdown of the category of the jobs to which this job belongs, i.e., short-wide jobs have a high category-slowdown and hence would have a higher QoS charge for a requested slowdown as compared to long-narrow jobs which have a low category-slowdown.

We classify different categories for the jobs by using two parameters: the runtime of the job and the number of processors requested by the job. In particular, we spilt up both the time and processors into sixteen different categories ranging from short-narrow jobs to long-wide jobs. We run the EASY backfill for different loads with several stringency factors and calculate the corresponding category slowdown for the various types of jobs.

6 Modeling User Tolerance

As discussed earlier, when a new job is submitted, the Feedback based QoPS scheme (FQoPS) tries to find a schedule without violating the deadline constraints of either the new job or the existing jobs in the schedule. If it is able to satisfy the deadline, it accepts the job. However, if it is unable to satisfy the deadline, it doesn't drop it instantaneously. Instead, it finds the best possible deadline it can provide to the job and returns it to the user. Then it is in the user's discretion to accept or reject the offered deadline. In the modeling of our scheme, we tried to emulate the most rational behavior of the users. For instance, we emulate user tolerance in terms of the factor of extension in the requested deadline that the user might be willing to accept in case the scheme fails to accept the job within the initially requested deadline. We quantify this factor by a parameter named Tolerance Factor (TF). This parameter is completely based on the user characteristics. In our simulations, we show the impact of such tolerance on the part of the user on the overall profits attainable by the supercomputer center by choosing different assumed values for this paramter.

Whenever the FQoPS scheme fails to schedule a job within the user requested deadline, it finds a schedule at the earliest possible time. If the earliest possible deadline the scheme can provide is within the user tolerance given by (TF x Requested Deadline), we assume that the user would accept the provided deadline. Otherwise, we assume that the user would not accept the provided deadline.

6.1 Impact of User Tolerance

In our first set of simulation experiments, we study the behavior of the FQoPS scheme with respect to admittance capacity, resource charge and QoS charge metrics as a function of TF. As it is hard to predict or simulate the user characteristics, we use the TF in two different ways - Fixed TF and Random TF. For Fixed TF, we assume that all users have the same amount of tolerance (equal TF for all jobs). These experiments try to show critical insights into the real impact of the user tolerance without being diluted by the difference in the tolerance amongst various users. For Random TF, we add some degree of randomness by arbitrarily choosing the effective TF value between 0 and 2 times the TF value (defined as a simulation parameter). These experiments try to provide the real life environment where different users have different tolerance values. The TF defined as the simulation parameter shows the general characteristics of the users in this case, e.g., a high value of TF would create a scenario where the users are more tolerant while a low value of TF would create a scenario where the users are less

tolerant.

Figure 3a shows the variation in the number of jobs accepted with respect to TF. We can see that fewer jobs are accepted as the TF increases initially; after that the number of accepted jobs increases monotonically and can reach to 100% acceptance for even higher TF. The reason for the initial downward trend is attributed to the acceptance of many heavy jobs (w.r.t. processor seconds). The same TF value would provide a higher absolute tolerance for larger jobs when compared to smaller jobs. When the absolute tolerance was less (for small TF values), earlier arriving small jobs could use up the space in the schedule and force the large job to be dropped. When the tolerance increases, the large job can still be accepted due to the additional absolute tolerance available. This however would mean that later arriving smaller jobs would be dropped since the large job would use up the space in the schedule. This hypothesis is strengthened by the variation of the accepted Processor-Seconds as shown in Figure 3b. As the TF value increases, the overall accepted processor-seconds increase. When the TF value becomes sufficiently large, however, the trend is more intuitive with more and more jobs being accepted. To further verify this, we show the category-wise breakup of the number of jobs and processor-seconds accepted for each category of the jobs for load 1.3 (Figure 4) and for load 1.6 (Figure 5). It can be seen that as the TF increases, more larger jobs are accepted at the cost of the smaller jobs.

We next look at the variation of the resource charge of the system with TF (Figure 6). The resource charge increases as the TF increases. This is pretty intuitive as the resource charge is directly proportional to the accepted processor-seconds.

Figure 7 shows the QoS Charge as a function of TF for different loads. This figure shows a counter-intuitive result of a monotonic drop in the QoS charge with user tolerance, i.e., if the users are more tolerant, the supercomputer center gets lesser profit from the jobs! This is attributed to the per-job profit achieved with increasing user tolerance. With an increase in the TF value, jobs which would have failed in the original no-tolerance scheme are now accepted. Due to this, later arriving jobs which would have been accepted with the user requested deadline are unable to be admitted with this deadline. However, due to the increased user tolerance, they are still admitted with a looser deadline, i.e., the later arriving jobs are still admitted, but for a lesser QoS charge. This effect cascades for later arriving jobs causing an overall decrease in the QoS charge the supercomputer center can get.

7 Providing Artificial Slack

As discussed in Section 6, users are most of the times not very strict in their requested deadlines. If the supercomputer center is not able to meet their requested deadline, we try to find the best possible deadline the center can provide and let the user decide if this deadline is acceptable. However, as we have seen, though the resource charge achieved by the supercomputer center increases as the users become more and more tolerant, the overall QoS charge falls.

Depending on the ratio of the QoS charge to resource charge imposed by the supercomputer center, a high tolerance might provide a higher or lower overall profit. One extreme is for the supercomputer center to have no explicit charge for the QoS provided, in which case the center has to do nothing; the tolerance of the users would automatically improve the profit achievable by the center. The other extreme is to charge the users only based on the QoS they requested. In this case, the original QoPS algorithm would perform the best, since its behavior would be equivalent to that of no user tolerance (TF = 1.0). It is to be noted that user tolerance is completely dependent on the user characteristics and is not a parameter under the control of the supercomputer center. So, in the general case where the overall profit depends on both the QoS charge as well as the resource charge, we need to come up with a different approach to try to negate the effect of the user tolerance to any required degree.

In this section, we introduce the concept of an "Artificial Slack" provided to jobs. If we are not able to accept a job with the user requested deadline, providing the best possible deadline would maximize the revenue achievable from that job. However, this might result in a tight schedule for the rest of the jobs causing many later arriving (and potentially more urgent) jobs to be dropped. Instead, we provide a certain artificial slack to such jobs and return an even looser deadline to the user. If the user agrees to submit the job with this deadline, the supercomputer center would gain more flexibility to admit later arriving jobs. We model this slack with an additional parameter called Slack Factor (SF). The offered deadline to the user is given by (Arrival Time + (Earliest possible deadline - Arrival Time) x SF).

We study different cost metrics with different slack (SF) and tolerance factor (TF) values. It is worthy to note that the value of the Slack Factor could be 1 or more (SF = 1 means no slack). Also for any TF value less than SF, the scheme behaves like our original QoPS, i.e. all the jobs whose initially requested deadline could not be met, are dropped immediately. Further, in general, an increase in the value of SF tends to negate user tolerance, i.e., increasing SF tends to be equivalent to decreasing TF. We evaluate the metrics for both moderate load (1.3) and high load (1.6).

Figure 8 shows the variation of the resource charge with the Slack Factor (SF) for different Tolerance Factors. With increasing SF, again, larger jobs have a higher absolute slack value. This lets more small jobs to be admitted by utilizing this slack. The admittance of these small jobs, however, uses up the slack present in the schedule and forcing later arriving larger jobs to be dropped. So, in general, increasing SF has the inverse effect of increasing TF.

Figure 9 shows the variation of the QoS charge with the

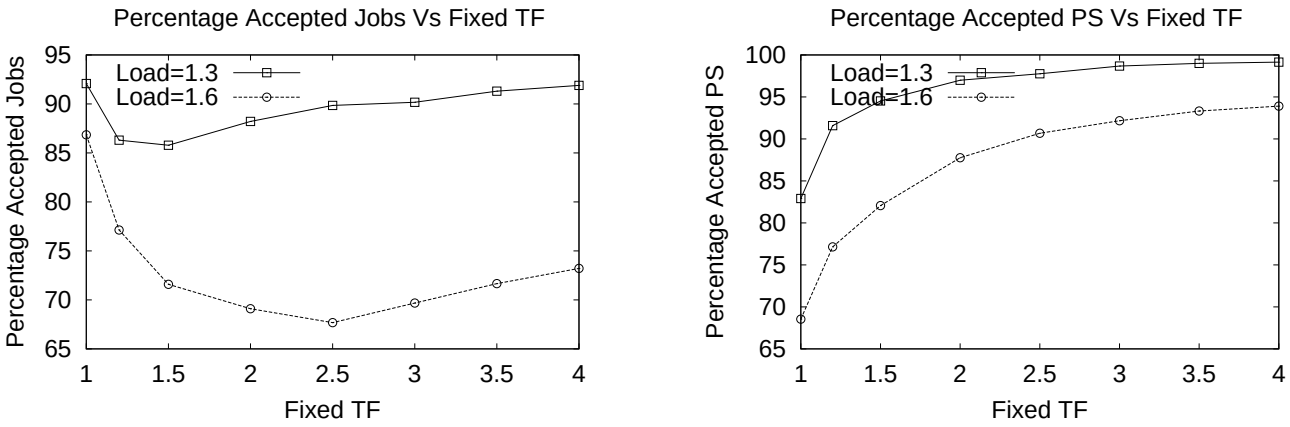

Figure 3. Effect of Tolerance Factor for different loads with Stringency Factor = 0.2 on (a)Job admittance capacity (b) Processor Seconds acceptance

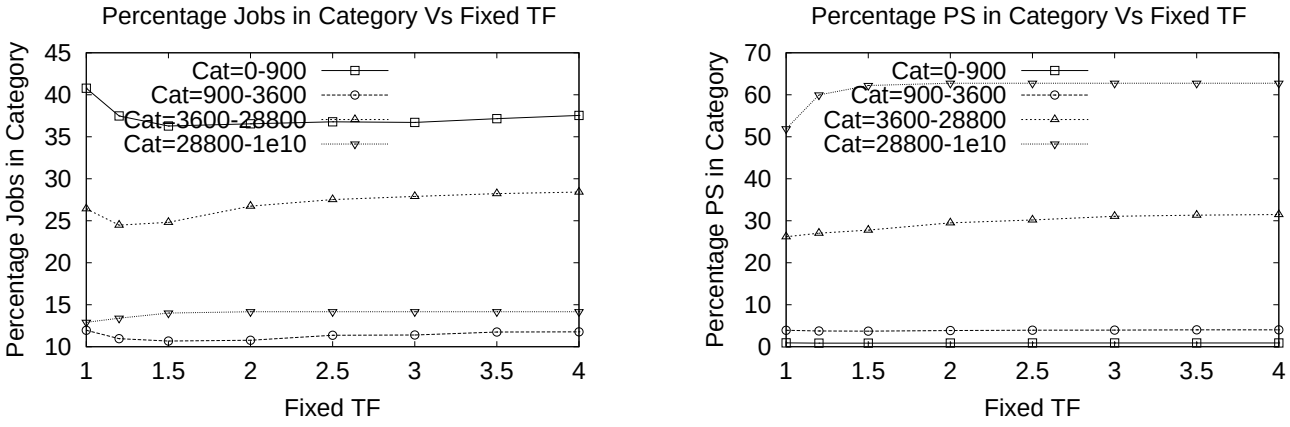

Figure 4. Acceptance rate for different categories of accepted Jobs with Stringency Factor = 0.2 and load = 1.3: (a) Number of Jobs Accepted (b) Processor Seconds Accepted

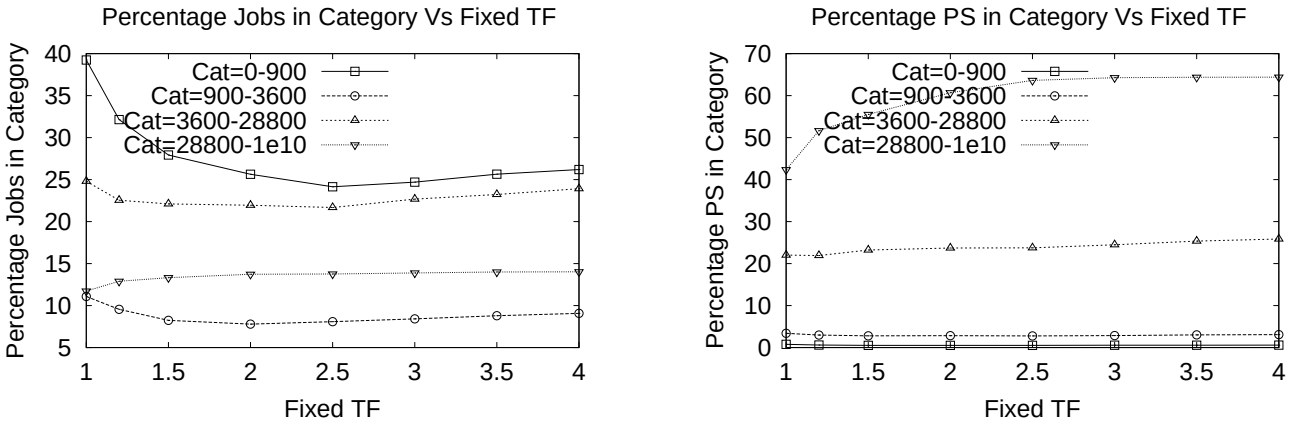

Figure 5. Acceptance rate for different categories of accepted Jobs with Stringency Factor = 0.2 and load = 1.6: (a) Number of Jobs Accepted (b) Processor Seconds Accepted

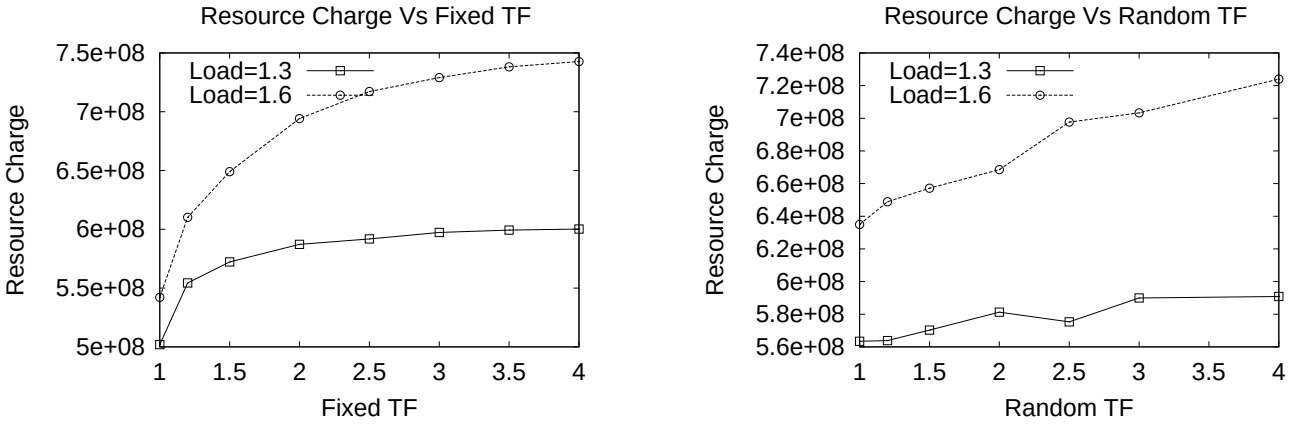

Figure 6. Resource Charge for different loads with Stringency Factor = 0.2: (a) Fixed Tolerance Factor (TF) (b) Random Tolerance Factor (TF)

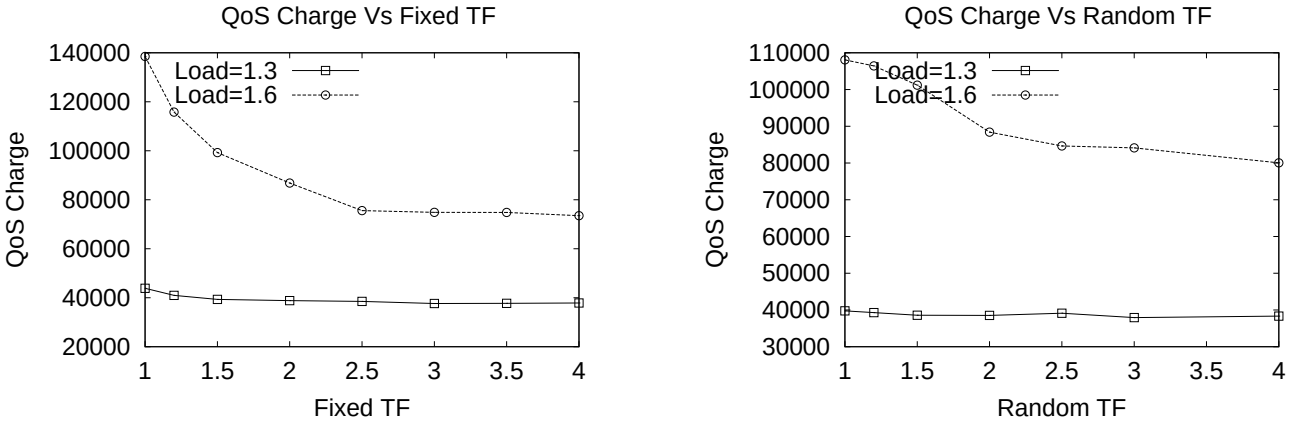

Figure 7. QoS Charge for different loads with Stringency Factor = 0.2 (a) Fixed Tolerance Factor (TF) (b) Random Tolerance Factor (TF)

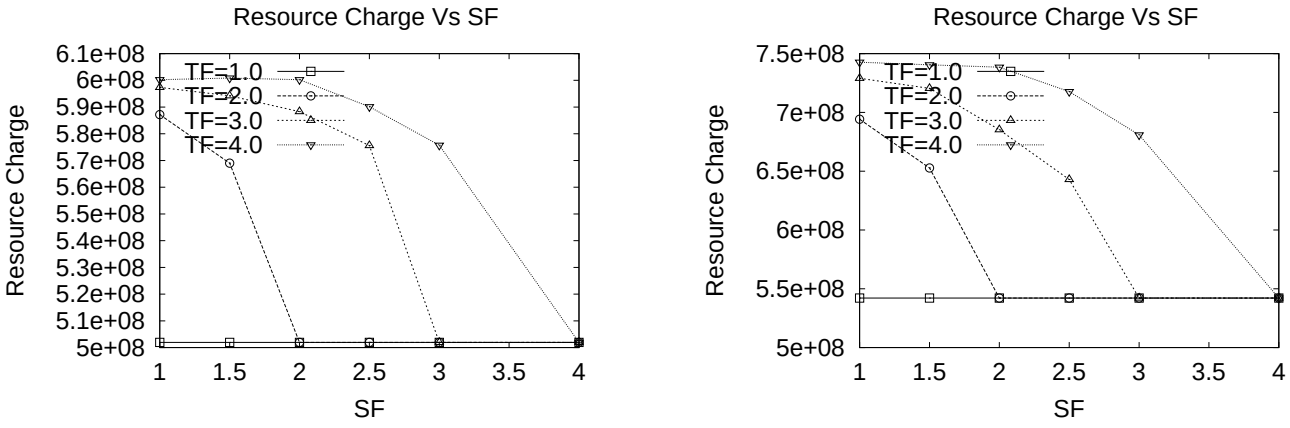

Figure 8. Resource Charge for different Slack Factors with Stringency Factor = 0.2: (a) For Load = 1.3 (b) Load = 1.6

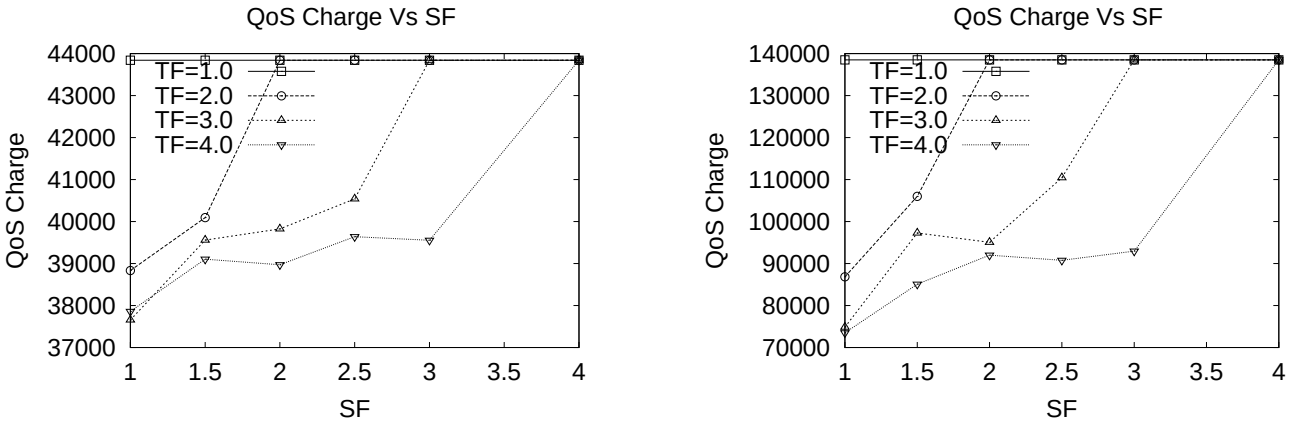

Figure 9. QoS Charge for different Slack Factors with Stringency Factor = 0.2: (a) For Load = 1.3 (b) Load = 1.6

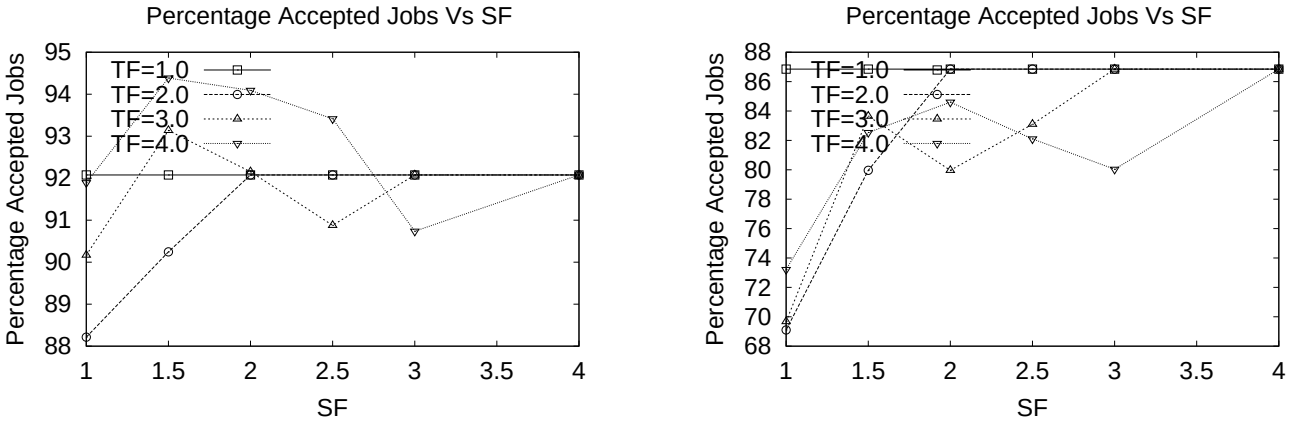

Figure 10. Admittance capacity for different Slack Factors with Stringency Factor = 0.2: (a) For Load = 1.3 (b) Load = 1.6

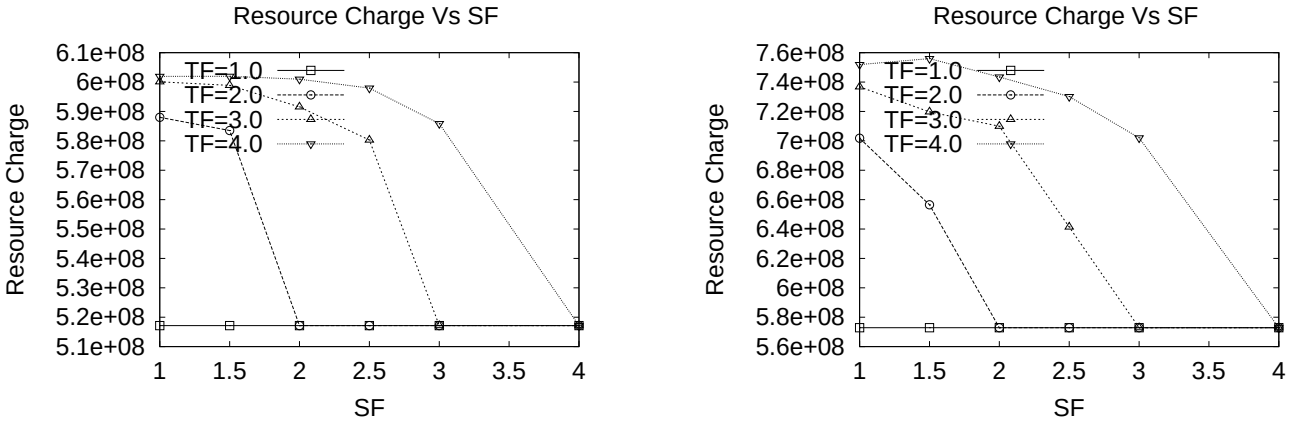

Figure 11. Resource Charge for different Slack Factors with Stringency Factor = 0.2 and 80% deadline jobs: (a) For Load = 1.3 (b) Load = 1.6

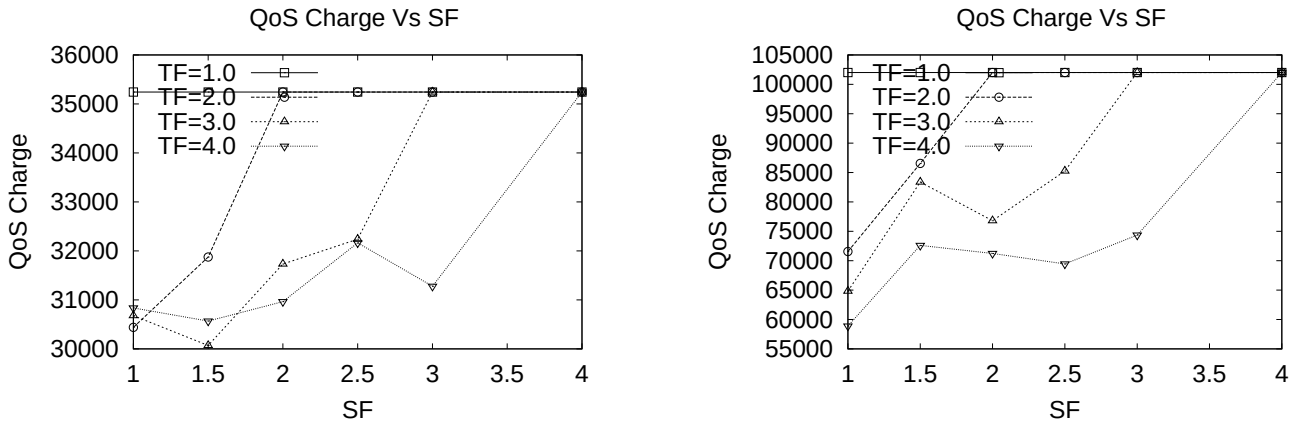

Figure 12. QoS Charge for different Slack Factors with Stringency Factor = 0.2 and 80% deadline jobs: (a) For Load = 1.3 (b) Load = 1.6

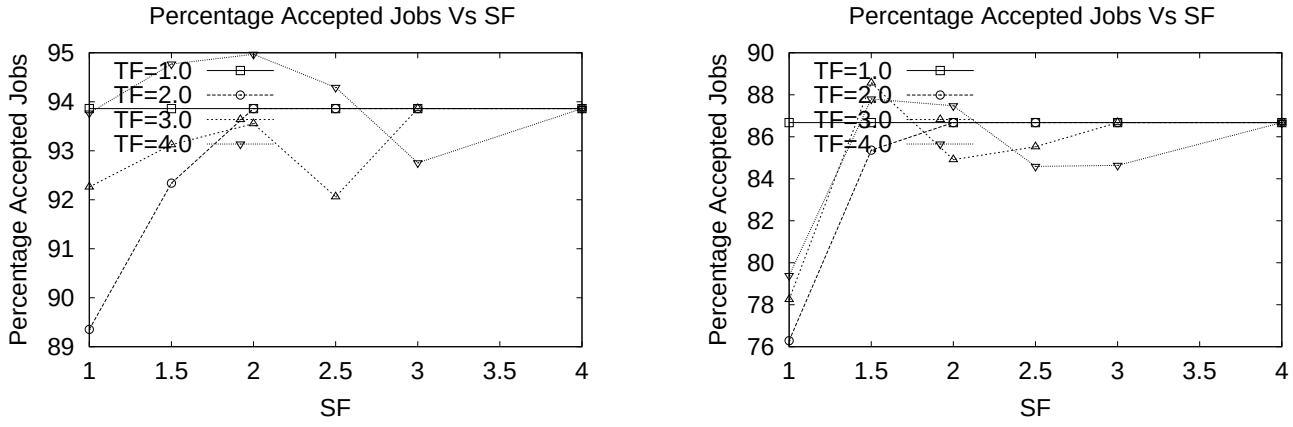

Figure 13. Admittance capacity for different Slack Factors with Stringency Factor = 0.2 and 80% deadline jobs: (a) For Load = 1.3 (b) Load = 1.6

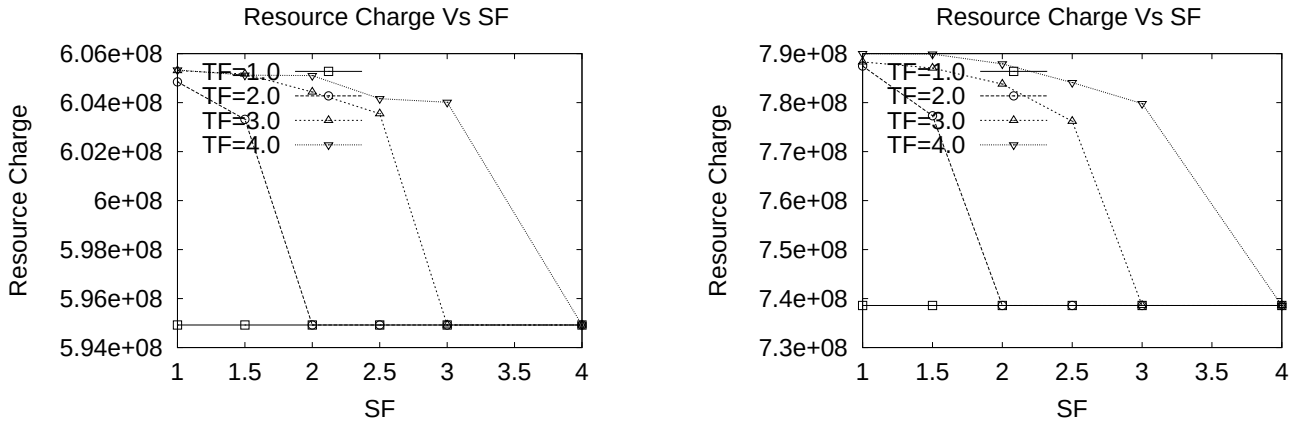

Figure 14. Resource Charge for different Slack Factors with Stringency Factor = 0.2 and 20% deadline jobs: (a) For Load = 1.3 (b) Load = 1.6

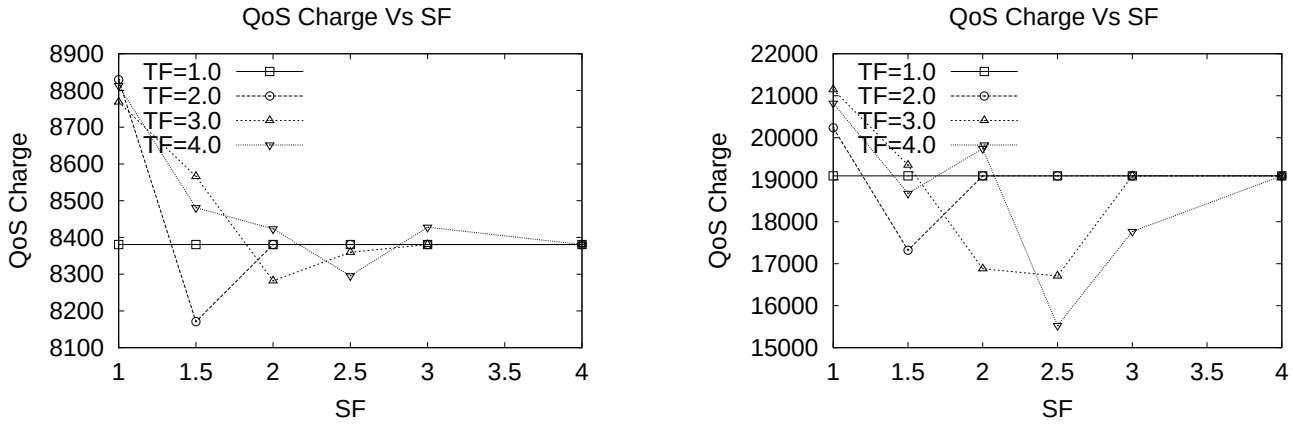

Figure 15. QoS Charge for different Slack Factors with Stringency Factor = 0.2 and 20% deadline jobs: (a) For Load = 1.3 (b) Load = 1.6

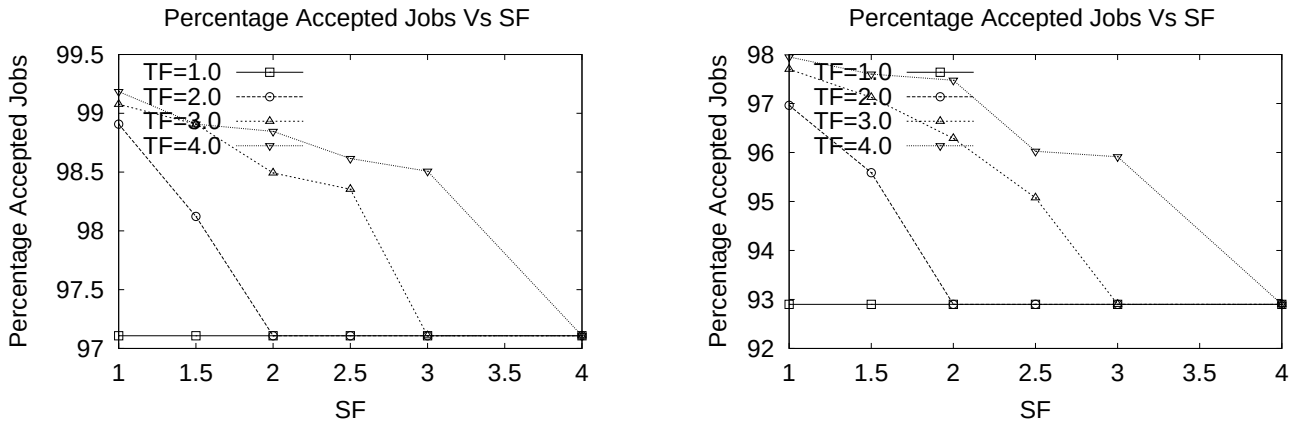

Figure 16. Admittance capacity for different Slack Factors with Stringency Factor = 0.2 and 20% deadline jobs: (a) For Load = 1.3 (b) Load = 1.6

Slack Factor (SF) for different Tolerance Factors. Again, we see a similar effect as the previous graph: increasing SF has the inverse effect of increasing TF, i.e., QoS charge increases with increasing SF. Admittance capacity for different slack factors are presented in Figure 10 for load 1.3 and load 1.6. It is obvious from these graphs that when the SF value is greater that TF value, the scheme behaves like job-shedding QoPS.

In the above cases, we consider all the jobs are urgent and submitted with a deadline. But in a more practical system, some jobs can be urgent and some are not. We examine the behavior of different metrics for our scheme in this mixed environment. Figure 11 to 13 show the graphs when non-urgent jobs are few (80% deadline jobs). Data for the dominantly non-urgent jobs(20% deadline jobs) are presented in the Figure 14 to 16

8 Kill and Restart

Some supercomputer centers support a kill-and-restart mechanism - a running job can be killed and restarted as a new job if no permanent files are modified during the run. The model of I/O for such jobs at the Ohio Supercomputer Center is that all input files are first copied into a special temporary directory created for the job, and all output is written into files in that temporary directory during execution of the program; after successful completion of the program, the output files are‘copied from the temporary directory to the persistent files. If such a job is aborted and then restarted from scratch and run to completion, the final results would be exactly the same as running to completion on the first initiation. In this section, we present an approach to utilize such capabilities provided by the supercomputer center to further improve the profit achievable.

The basic idea of the algorithm is to schedule a new job by killing a running job if the QoPS scheme fails to find a schedule for the new job within its deadline. In this approach, instead of trying to find the earliest possible deadline for a job whose initially requested deadline could not be satisfied, we try to satisfy the requested deadline by killing and restarting an already running job. If we are able to satisfy the job's requested deadline, we try to reschedule the killed job within its guaranteed deadline. If we are able to schedule this job too, the final schedule is accepted.

Our scheme uses a heuristic approach to determine the order in which running job should be killed. The running jobs are sorted in the increasing order of used processors seconds (the processor seconds for which they have already run so far). We then kill the first job in the list and try to schedule the new job. If an acceptable schedule is found for the new job, we try to schedule the killed job again within its guaranteed deadline. If both jobs are thus scheduled without violating their deadlines, we accept the new schedule. Otherwise we keep the first job running and follow the same steps with the second running job from the sorted list. If no schedule is possible after trying all the running jobs, we try to find the earliest possible deadline we can provide and return it to the user.

Figures 17 and 18 show the resource and QoS charges achievable by the supercomputer center for varying slack factors with tolerance factors of 2 and 4 for load 1.6. We see that for small tolerance factors, there's no difference in either charge (Resource or QoS) between using Kill-and-Restart and not using it. However, as the tolerance factor increases to 4, we see that Kill-and-Restart provides a better QoS charge while losing out on the resource charge. The gain in the QoS charge is attributed to the capability of the Kill-and-Restart scheme to admit later arriving urgent jobs by killing one of the already running jobs. On the other hand, the loss in the resource charge is attributed to the wastage of processor-seconds due to the restarting of the jobs. We don't observe any difference between the two schemes for a low tolerance factor due to the low amount of flexibility available in the schedule. Figures 19 presents the the effect of admittance capacity in the above cases. From graph, we can see Kill and Restart performs better in admittance capacity for both low and high TF, though at high TF the differences are more observable. We also study the behavior of FQoPS in a moderate load (1.3) where get nearly the identical trend as we see at a higher load (1.6). Figures 20 to Figures 22 demonstrate the same different metrics for load 1.3.

In general, if the system has a mix of urgent and non-urgent jobs, Kill-and-Restart allows us to admit more urgent jobs by killing already running non-urgent jobs and increase the overall QoS cost in the system. To further strengthen this argument, we study the impact of the Kill-and-Restart based scheme on workloads where not all jobs require deadlines. This is a more realistic scenario for real supercomputer centers. Some jobs request a hard deadline and pay more (in the form of the QoS charge) while others don't request any hard deadline and pay lesser (only the resource charge). We however provide an artificial deadline to even the non-deadline jobs to ensure that there's no starvation in the system. At the same time, these jobs are given sufficient slack so that they don't interfere in the admission of true deadline based jobs.

Figures 23 to 34 show the resource charge, QoS charge, admittance capacity metrics for load 1.3 and load 1.6 in two scenarios: (i) where 80% of the jobs are deadline jobs and (ii) where 20% of the jobs are deadline jobs. We can see that the Kill-and-Restart based scheme effectively utilizes the slack provided by the non-deadline jobs to improve the QoS charge achieved by the supercomputer center. Further, for a high percentage of deadline jobs (80%), Kill-and-Restart suffers with lesser resource charge due to the wastage of processor-seconds. However, when the performance of deadline jobs is reduced to 20%, Kill-and-Restart is beneficial even in the resource charge.

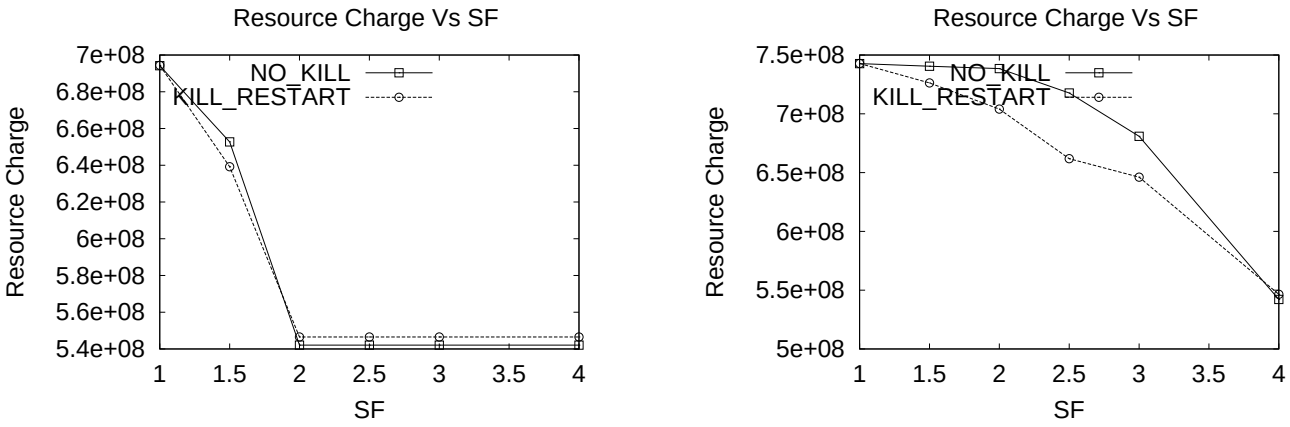

Figure 17. Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2 and load = 1.6 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

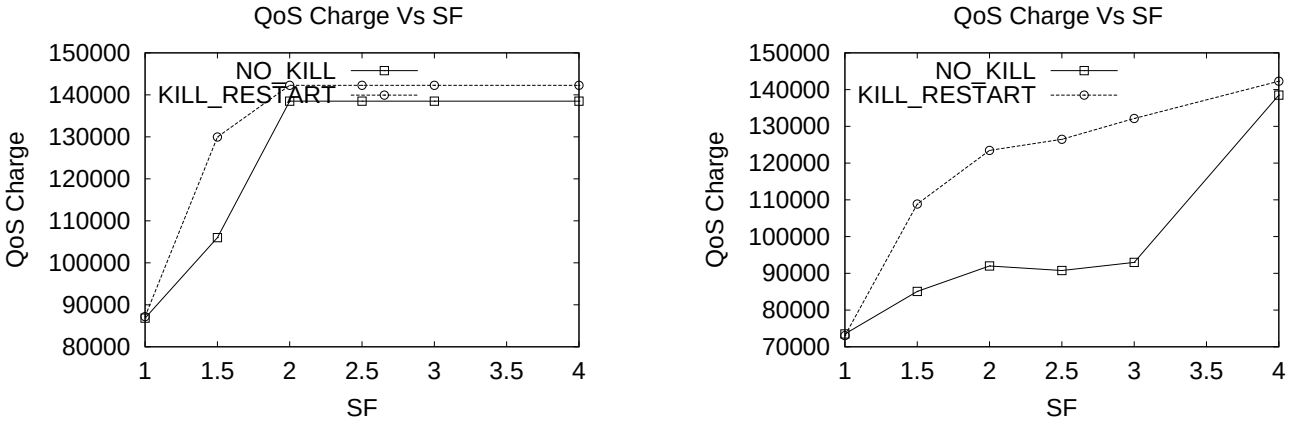

Figure 18. QoS Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2 and load = 1.6 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

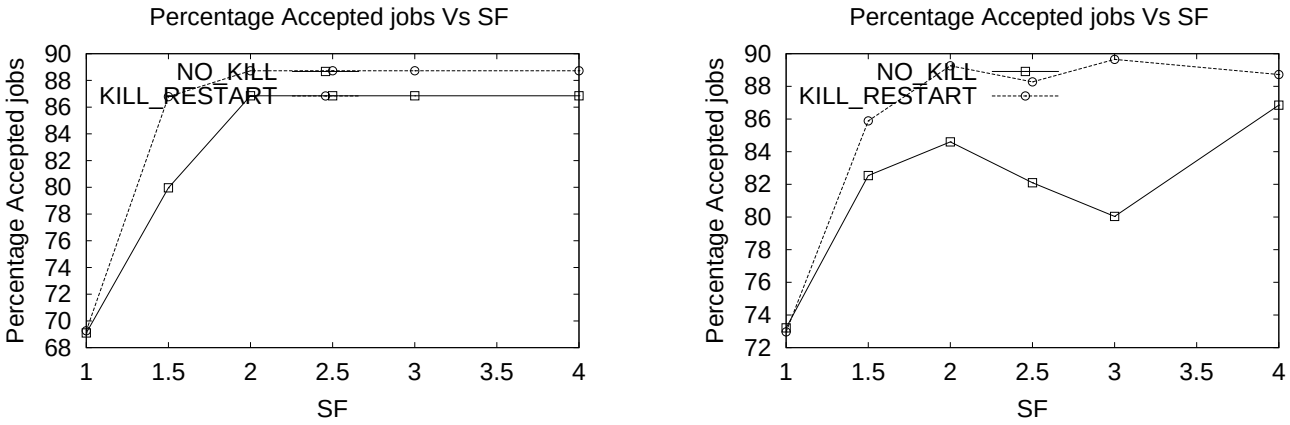

Figure 19. Admittance capacity with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2 and load = 1.6 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

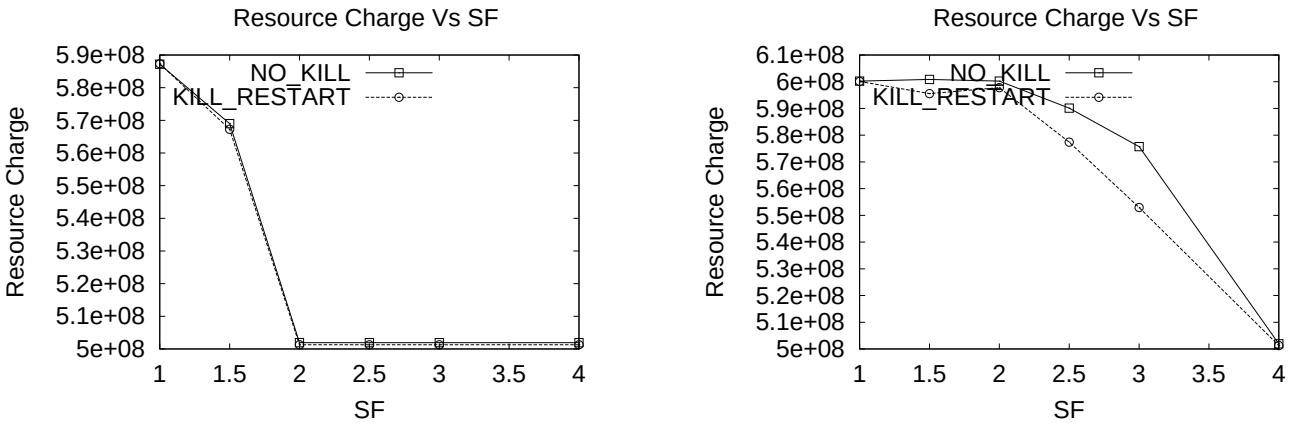

Figure 20. Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2 and load = 1.3 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

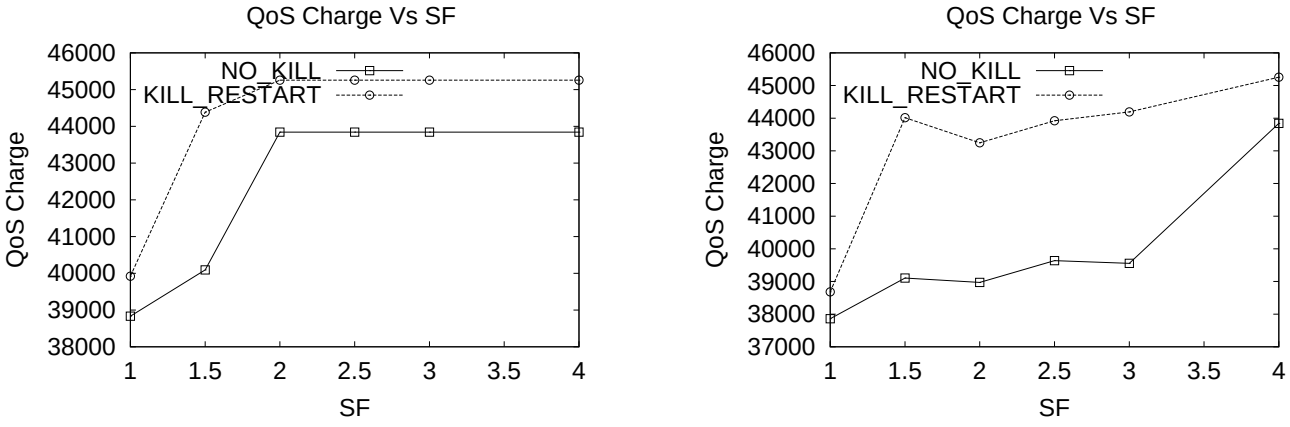

Figure 21. QoS Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2 and load = 1.3 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

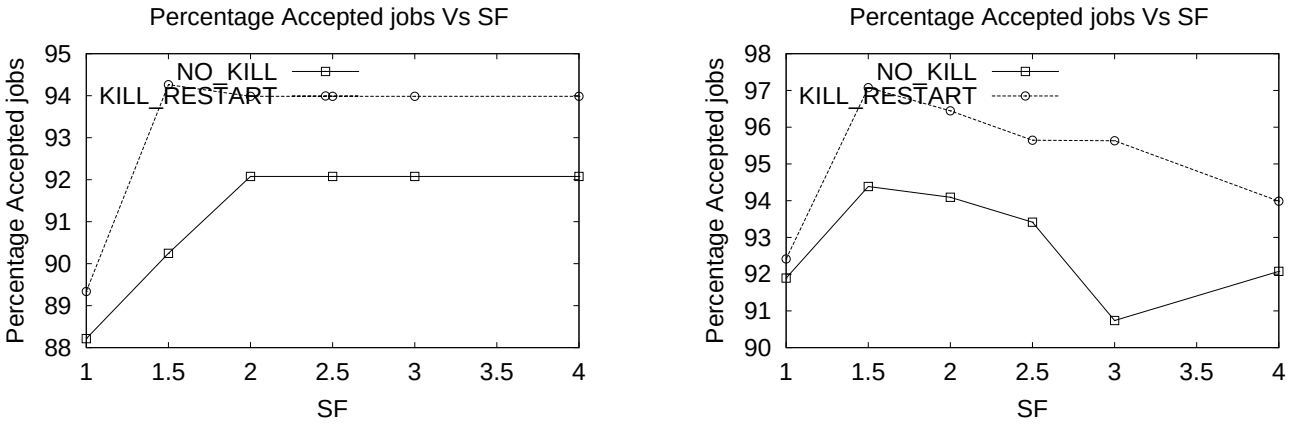

Figure 22. Admittance capacity with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2 and load = 1.3 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

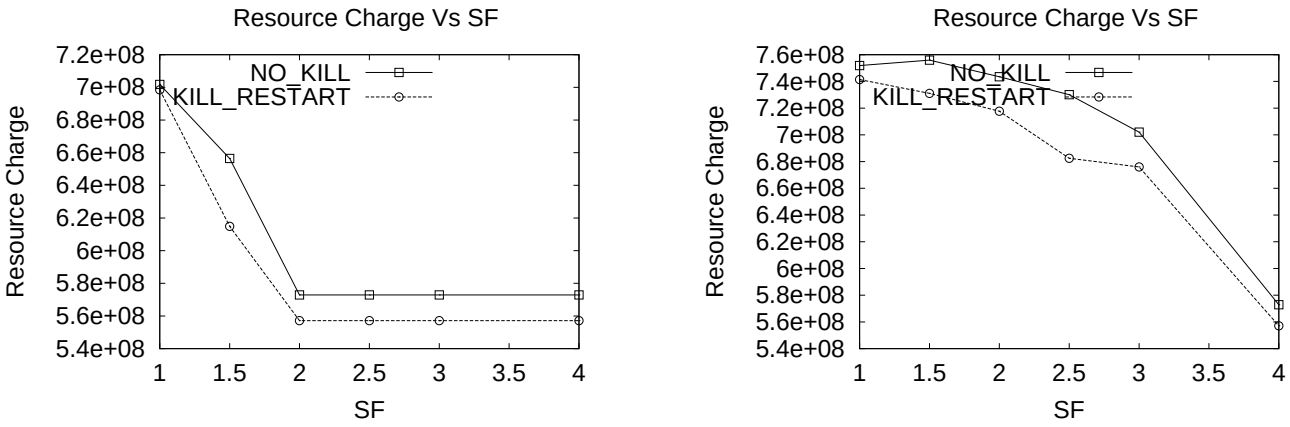

Figure 23. Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 80% deadline jobs and load = 1.6 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

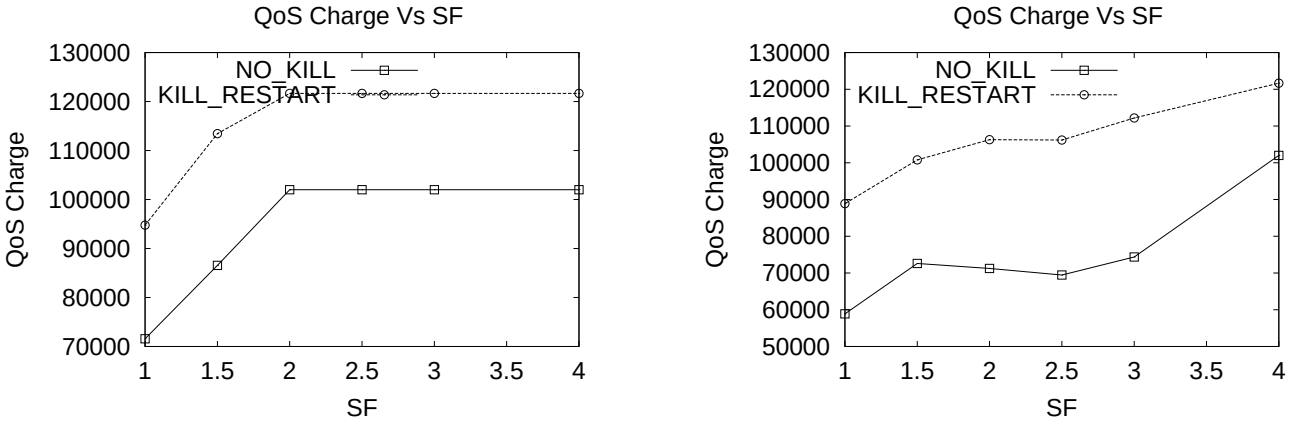

Figure 24. QoS Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 80% deadline jobs and load = 1.6 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

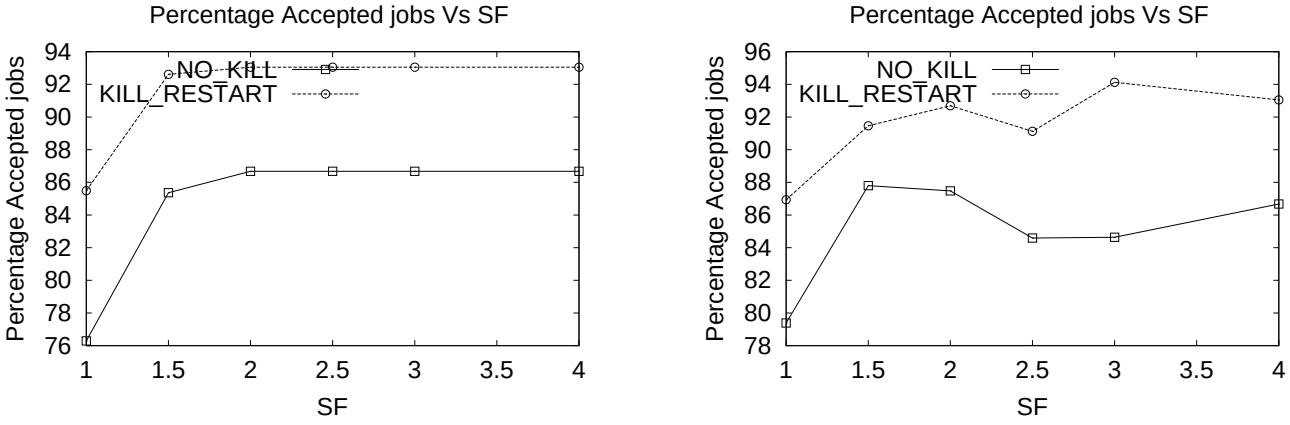

Figure 25. Admittance capacity with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 80% deadline jobs and load = 1.6 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

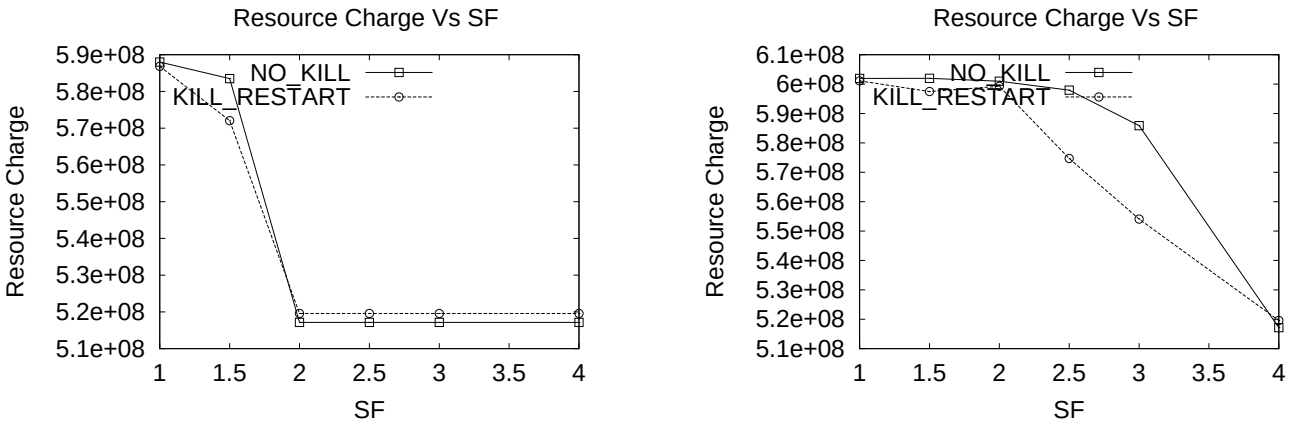

Figure 26. Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 80% deadline jobs and load = 1.3 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

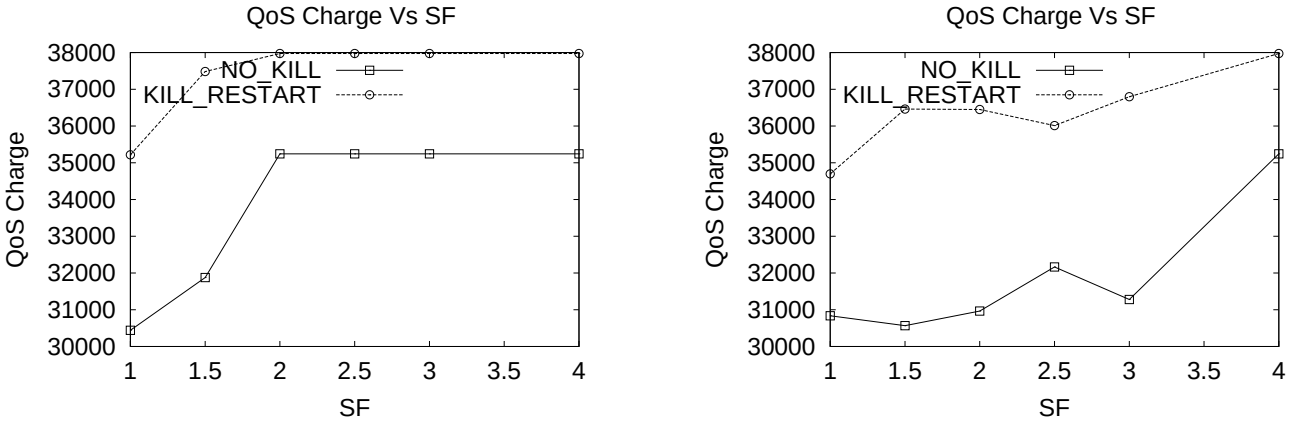

Figure 27. QoS Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 80% deadline jobs and load = 1.3 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

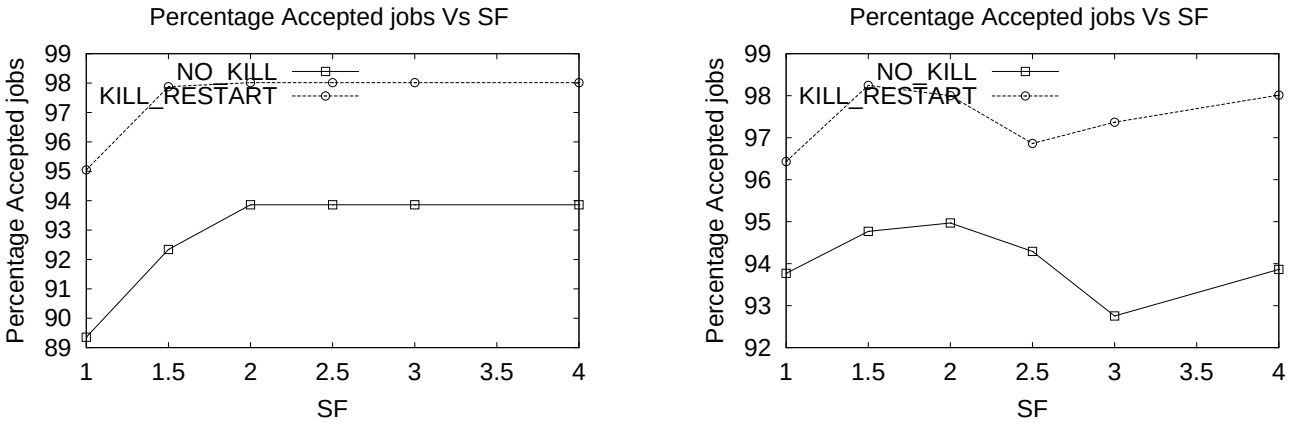

Figure 28. Admittance capacity with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 80% deadline jobs and load = 1.3 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

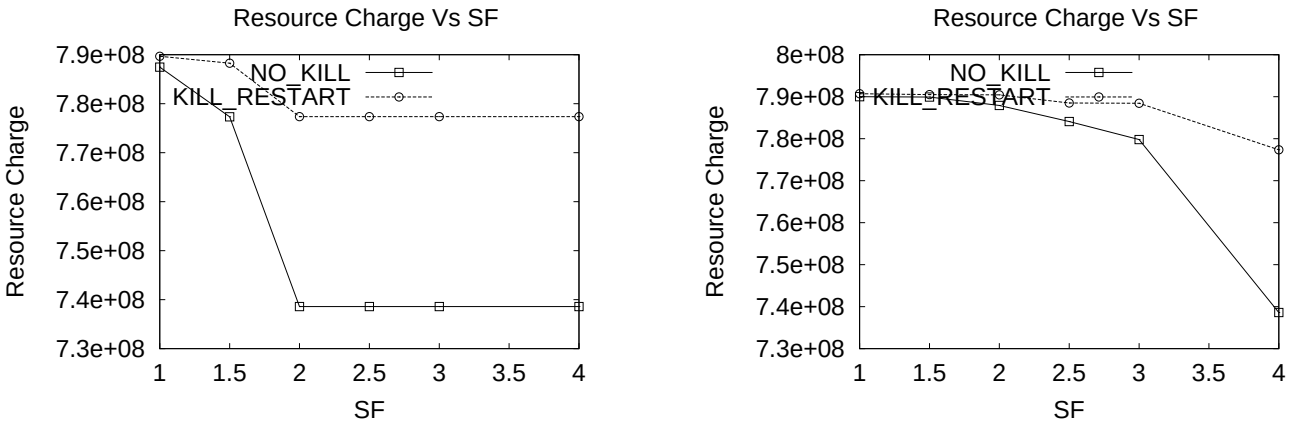

Figure 29. Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 20% deadline jobs and load = 1.6 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

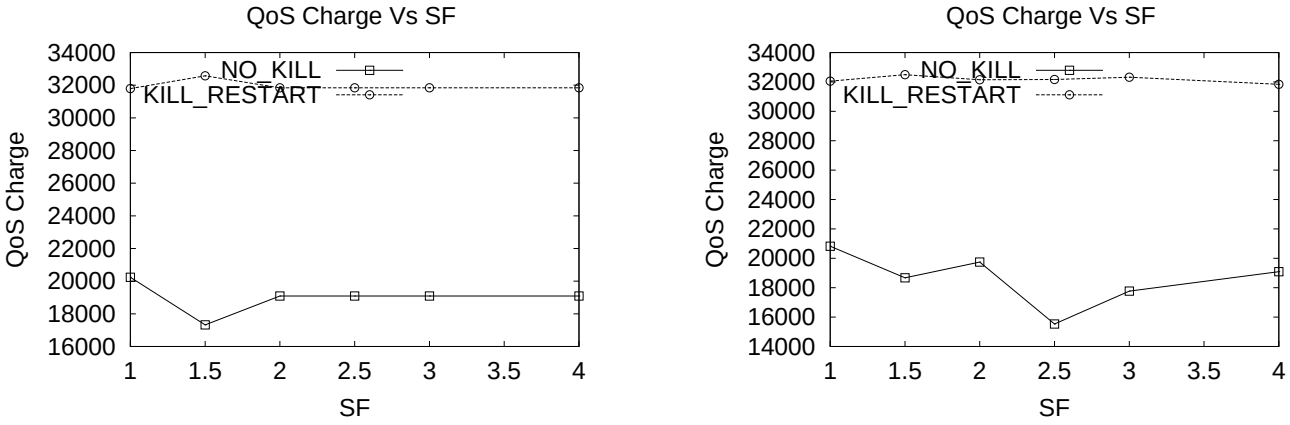

Figure 30. QoS Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 20% deadline jobs and load = 1.6 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

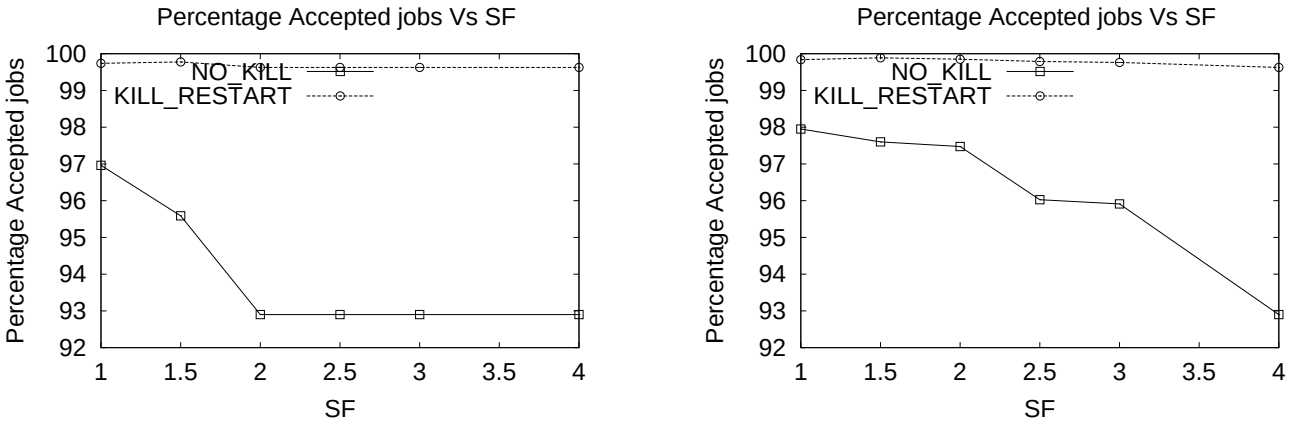

Figure 31. Admittance capacity with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 20% deadline jobs and load = 1.6 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

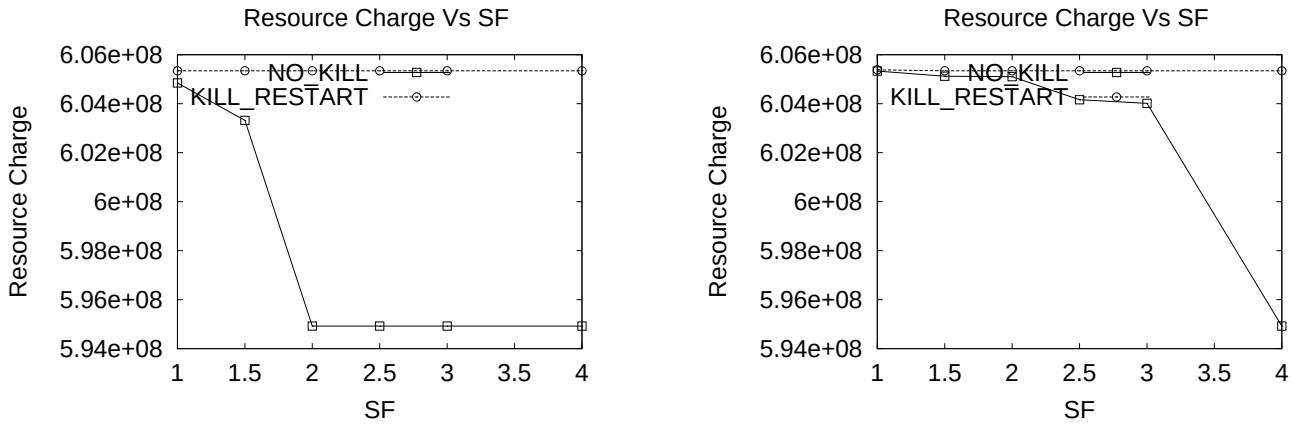

Figure 32. Resource Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 20% deadline jobs and load = 1.3 (a) Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

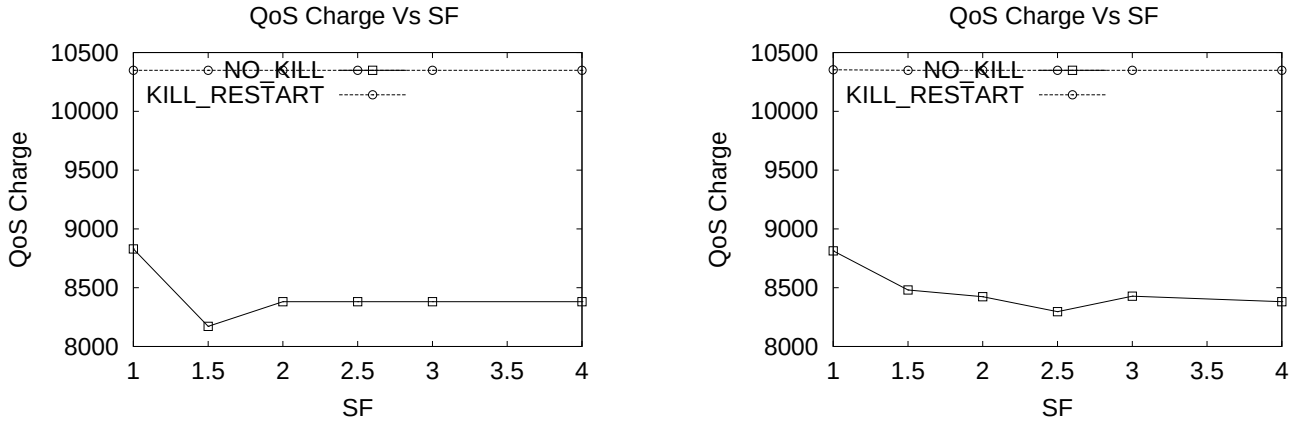

Figure 33. QoS Charge with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 20% deadline jobs and load = 1.3 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

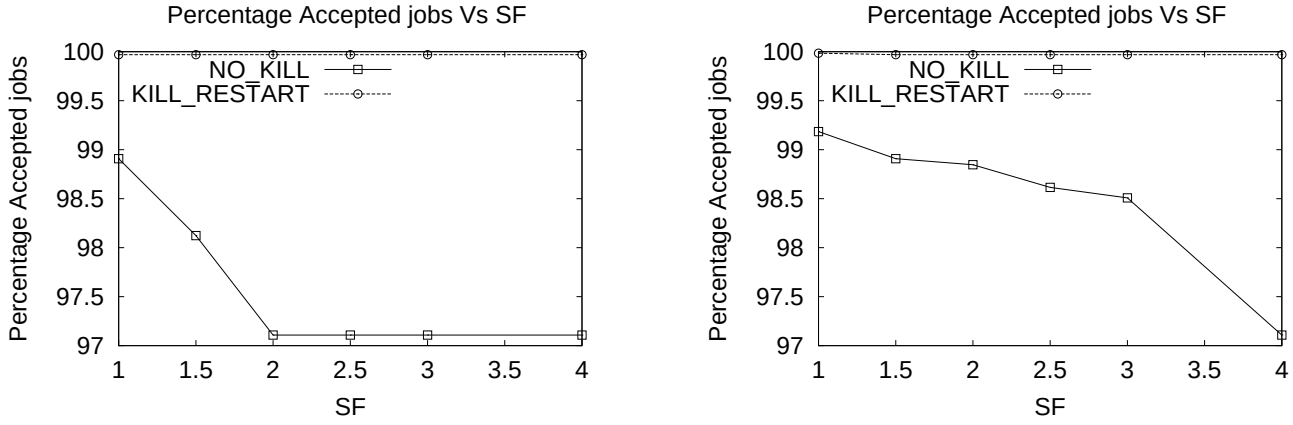

Figure 34. Admittance capacity with Kill-and-Restart and without Kill-and-Restart for Stringency Factor = 0.2, 20% deadline jobs and load = 1.3 (a)Tolerance Factor = 2.0 (b) Tolerance Factor = 4.0

9 Concluding Remarks

Although there has been considerable research on the topic of scheduling of parallel jobs, the issue of provision of QoS has received little attention. In this paper, we extended a previously proposed scheme (QoPS) to provide Quality of Service to submitted jobs; we propose extensions to the algorithm in multiple aspects: (i) a feedback mechanism to provide the best possible deadline for jobs whose requested deadline could not be met, (ii) providing artificial slack to some jobs to maximize the overall profit the supercomputer center can achieve and (iii) utilizing Kill-and-Restart improve the profit attainable.

As a part of the future work, we plan to incorporate QoS in the Maui/Moab and scheduler and deploy it the Ohio Supercomputer Center (OSC).

References

[1] NERSC. http://hpcf.nersc.gov/accounts/priority_charging.html.

[2] Su-Hui Chiang and Mary K Vernon. Production Job Scheduling for Parallel Shared Memory Systems. In *the Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, April 2001.

[3] Walfredo Cirne and Francine Berman. Adaptive Selection of Partition Size of Supercomputer Requests. In *the Proceedings of 6th workshop on Job Scheduling Strategies for Parallel Processing*, April 2000.

[4] D. G. Feitelson. Logs of Real Parallel Workloads from Production Systems.

[5] Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. Theory and Practice in Parallel Job Scheduling. In *the Proceedings of IEEE Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.

[6] Praveen Holenarsipur, Vladimir Yarmolenko, Jose Duato, D K Panda, and P Sadayappan. Characterization and Enhancement of Static Mapping Heuristics for Heterogeneous Systems. In *the Proceedings of the IEEE International Symposium on High Performance Computing (HiPC)*, December 2000.

[7] Mohammad Islam, Pavan Balaji, P. Sadayappan, and D. K. Panda. QoPS: A QoS based scheme for Parallel Job Scheduling. In *the Proceedings of the 9th workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, June 2003.

[8] Mohammad Islam, Pavan Balaji, P. Sadayappan, and D. K. Panda. Towards Provision of Quality of Service Guarantees in Job Scheduling. Technical report, The Ohio State University, Columbus, OH, April 2004.

[9] B. Jackson, Brian Haymore, Julio Facelli, and Quinn O. Snell. Improving Cluster Utilization Through Set Based Allocation Policies. In *IEEE Workshop on Scheduling and Resource Management for Cluster Computi ng*, September 2001.

[10] Pete Keleher, Dmitry Zotkin, and Dejan Perkovic. Attacking the Bottlenecks in Backfilling Schedulers. In *Cluster Computing: The Journal of Networks, Software Tools and Applications*, March 2000.

[11] A. K. Mok. *Fundamental design problems of distributed systems for the hard real-time environment*. PhD thesis, Massachussetts Institute of Technology, Cambridge, MA, May 1983.

[12] A. K. Mok. The design of real-time programming systems based on process models. In *the Proceedings of IEEE Real Time Systems Symposium*, December 1984.

[13] A. K. Mok and M. L. Dertouzos. Multi-Processor Scheduling in a Hard Real-Time Environment. In *the Proceedings of the Seventh Texas Conference on Computing Systems*, November 1978.

[14] A. W. Mualem and D. G. Feitelson. Utilization, Predictability, Workloads and User Estimated Runtime Esti mates in Scheduling the IBM SP2 with Backfilling. In *in IEEE Transactions on Parallel and Distributed Systems*, volume 12, June 2001.

[15] Dejan Perkovic and Peter J Keleher. Randomization, Speculation and Adaptation in Batch Schedulers. In *the Proceedings of the IEEE International Conference on Supercomputing*, November 2000.