Efficiently Scheduling Advance Reservations in Grids

Umar Farooq, Shikharesh Majumdar, Eric W. Parsons
Department of Systems and Computer Engineering
Carleton University, 1125 Colonel By Drive, Ottawa, ON, Canada K1S 5B6
{ufarooq, majumdar}@ sce.carleton.ca, eparsons@acm.org

Carleton University
Department of Systems and Computer Engineering
Technical Report SCE-05-14, August 2005.

Abstract

Advance reservations (ARs) were introduced for application-level dynamic scheduling of resources in a Grid infrastructure. Advance reservations of resources for a specific time in future not only ensure that all resources would be simultaneously available at the execution time of the application but also ensure that the QoS constraints of the Grid applications would be met. Previous research shows that ARs can meet their objectives but at a significant performance cost. In this paper, we argue that laxity in the reservation window of an AR can help improve performance of scheduling with advance reservations. Scheduling ARs with given laxities is an NP-Hard problem and the paper presents a scalable algorithm for scheduling on-demand and advance reservation requests with laxities. The paper then investigates in detail the effect of proportion of advance reservations, laxity and distribution of the size of tasks on performance through extensive experimentation. The paper also investigates that how much improvement in performance can be gained by task preemption and up to what percentage of overheads is preemption justified in scheduling of on-demand and advance reservation requests. We demonstrate how, for some workloads, laxity can be exchanged for preemption to achieve high utilization. Finally, the paper studies resource level policies to prevent starvation of on-demand requests.

Key Words: Computational/Data Grids, Resource Management in Grids, Advance Reservations, Grid Scheduler, Grid System Performance, Scheduling with Deadlines.

1. INTRODUCTION

Grid computing aims to provide dynamic, secure and coordinated sharing of heterogeneous resources across administrative and geographical boundaries. Virtually everything ranging from computers and networks to databases and scientific instruments can be viewed as potential Grid resources and can be shared among Grid users in accordance with certain policies and service level agreements. Grid computing applications varies from distributed supercomputing [1, 2] to high-throughput computing [3], from data-intensive computing as in high-energy physics [4] to collaborative computing [5, 6], from remote medicine to digital sky survey. In addition, in future, enterprise computing and utility grids will provide on-demand access to virtually limitless computing, storage and network capacities through the formation of virtual organizations. Each virtual organization may spread over multiple administrative domains.

Many Grid applications have requirements for heterogeneous resources that are independently controlled or administered. Since resources belonging to different administrative domains do not share their schedules, if a user's application needs to access more than one resource simultaneously, the user either has to arrange for it through the domain administrators or submit the tasks of the job to queues of different resources without any guarantees that all resources would be available simultaneously. In order to address this problem, advance reservations (ARs) were introduced as a part of *Globus Architecture for Reservation and Allocation* (GARA) [7]. Advance reservations of resources for a specific time in future ensure that all resources would be simultaneously available at the execution time of the application. As by reserving resources in advance, one can provide an upper bound on the response time, ARs can also be used for ensuring end-to-end quality of service. For jobs with sequential tasks, the response time of the first resource in sequence can become the start time of the reservation for the second resource and so on; thus guaranteeing the end-to-end response time.

Despite their attractive features, advance reservations have a downside – they can cause severe performance degradation. They result in lower utilization of the resources as they leave fragments of time in resource schedules where no tasks can be scheduled. They also increase wait times of the requests that are submitted to the queues of the resources instead of being submitted as advance reservations. Such requests are commonly known as on-demand requests (ODs) in the Grid community. Sulistio [8] and Smith [9] have studied the performance of ARs. Their results show that with only 20% of the tasks arriving as advance reservations, the utilization can go as low as 66% of the case where none of the tasks are advance reservations while mean wait times of the on-demand requests can increase by 71%. Their analysis is based on the assumption that the deadline to finish the advance reservation request is equal to reservation start time plus the worst-case execution time of the task on the resource. However, we argue that by having some amount of *laxity* in the reservation window we can improve the performance of advance reservation based scenarios by having more flexibility in task scheduling. Laxity of a task on a certain resource is the difference between its deadline and the time at which it would finish executing on that resource if it starts executing at its start time. The idea of laxity is not new to the scientific community where task requests are commonly specified with a ready time, an execution time and a deadline, and the deadline is usually greater than the sum of the ready and execution times. Lavian et al. [10] also follow the same approach for specifying data transfer requests in a Grid based dynamic optical network. However, they have not presented any algorithm for scheduling such requests.

It is to be noted that the idea of laxity may not work for some scenarios where, for instance, parallel tasks of an application running on different resources need to communicate while executing. In such cases, it may be necessary to ensure that all parallel tasks are scheduled on different resources at exactly the same time. For single-task and sequential applications and for applications with parallel branches that do not communicate, we can take advantage of laxity in the reservation window. Applications with multiple tasks, for instance, can divide their end-to-end response time into response times for each phase of the job, each having some laxity.

Scheduling advance reservation requests with a given start time, execution time and a given amount of laxity is an NP-Hard problem [11]. Algorithms presented in the real-time domain [11, 12] for solving similar problems have different goals – instead of maximizing utilization of the resource they try to minimize the response time of the critical task. In addition, these algorithms assume that all tasks to be scheduled are known in advance, which is not true for the Grid domain where tasks arrive one by one. Moreover, these algorithms do not scale for large number of tasks that are common in the Grid domain. It has been reported in [12] that for a problem size of 100 tasks, the algorithm presented in [11] was unable to terminate after generating several tens of thousand of *nodes* (temporary

schedules) while the algorithm in [12] generated a few thousand nodes. The number of nodes grows exponentially as the problem size increases [12]. These algorithms in their current form thus do not seem capable of scheduling thousands of requests in a Grid. Furthermore, the algorithm presented in [11] does not support preemptable tasks while the algorithm presented in [12] requires explicit *preempt relations* between tasks for supporting preemption. We have modified the algorithm presented in the real-time domain for making it suitable for the Grid domain. The algorithm, that we call the SSS (Scaling through Subset Scheduling) algorithm, is presented in this paper. The SSS algorithm can successfully schedule hundreds of thousands of tasks, and is capable of studying scenarios involving both non-preemptable and preemptable tasks. The contributions of this paper are as follows:

- It studies in detail scheduling of advance reservations in Grids and argue that notion of laxity can improve its performance.
- It presents a scalable algorithm for scheduling on-demand and advance reservation requests and studies its performance.
- It presents results of the extensive sets of experiments that we have conducted to study in detail the effect of proportion of advance reservations, laxity and task preemption on the performance of scheduling with advance reservations for a variety of workload parameters.
- As the proportion of advance reservations increases, system utilization goes down because of the generation of time slots in the schedule where no requests can be accommodated. Preempting tasks that can be resumed later from the point of interruption can potentially improve performance by filling in small empty time slots in the resource schedule. However, overheads are usually associated with task preemption and its resumption. This paper investigates that how much improvement in performance can be gained by preempting tasks and up to what percentage of overheads is preemption justified in scheduling of on-demand and advance reservation requests in Grids.
- The paper also attempts to answer questions such as "Can laxity be exchanged for preemption to achieve high performance?" and "Does the effect of preemption diminish as laxity increases?"
- Finally, the paper discusses resource level policies to prevent starvation of on-demand requests.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work and presents the system model used in the research. In Section 3, we discuss the NP hard problem and present our modified algorithm for finding a solution. We provide details on our experimental setup in Section 4. Section 5 studies in detail with the help of experimental results the effect of workload parameters, laxity and task preemption on the performance of scheduling with advance reservations. We conclude the paper in Section 6 and give directions for future research.

2. BACKGROUND AND RELATED WORK

Resource management in Grids is a challenging task because a typical Grid application needs multiple heterogeneous resources that may span over administrative boundaries. The application needs to satisfy local service level agreements of the resources and meet its QoS requirements within its budget constraints. Software infrastructures required for resource management and other things such as security, information dissemination and remote access are provided through Grid toolkits such as Globus [13] and Legion [14]. For scheduling computational and data resources required by Grid applications, Globus Resource Allocation Manager (GRAM) [15] was presented as a part of Globus Toolkit. Later, for application-level dynamic scheduling of collection of resources, co-allocation and advance reservations were introduced as a part of *Globus Architecture for Reservation and Allocation* (GARA) [7]. Since then advance reservations have been studied in numerous contexts such as architecture for ensuring end-to-end quality of service for network applications [16], architecture for data-intensive collaboration [17], scheduling of data placement activities [18], job scheduler for clusters and supercomputers [19] and Grid based architecture for dynamic optical networks [10].

Smith [9] and Sulistio [8] have investigated the performance of scheduling with advance reservations. Naikastam et al. [20] have developed a tool for specifying on-demand and advance reservation requests for dynamic optical networks. They also study the performance of different algorithms for choosing appropriate light paths requested by ARs [20]. However, as in [8] and [9], it is also assumed in [20] that no reservation request has any laxity. Hence, all incoming advance reservation requests that overlap with any of the previously committed reservations are rejected in [20]. However, as in [10], this paper considers laxity in the reservation window.

2.1 System Model

The abstract simulation model used in this research is based on Grid resources and tasks. A Grid resource may be a single CPU or a multiple CPU based computing device, a storage system or a system consisting of both CPU and storage devices. A task is a schedulable entity. It can be a Grid application consisting of a single or multiple components with precedence relationship among the components. The application is often associated with a start time and a deadline by which its processing must be completed. System management consists of two components: *mapping* that concerns the choice of the resource on which the application is to be executed and the *scheduling* of applications mapped to a given resource. This research is concerned with the second component. Using an open model we assume that a stream of tasks arrives at a resource and a scheduler is deployed for scheduling of the tasks on the resource. In a different scenario, the application may be decomposed into its constituent components each of which is characterized by a start time that depends on the precedence relationships and a deadline. Each component is then mapped to a specific resource. Note that the decomposition of the end-to-end deadline of the application into deadlines for the different components and the techniques for mapping of the application components to resources are beyond the scope of this research. Techniques for mapping application's components to different available resources have been studied in several papers [21, 22, 23]. This research concerns the scheduling of the components mapped onto a given resource that is subjected to such task arrivals. The components of the different applications, in this scenario, are the schedulable task entities. The results of this research are applicable in both of these scenarios. Based on a simulation model of a resource subjected to task arrivals, this paper focuses on task scheduling and the impact of laxity on system performance. Proportion of ARs in the total number of task arrivals is varied as one of the parameter.

3. SCHEDULING ADVANCE RESERVATION REQUESTS WITH GIVEN LAXITIES

3.1 Problem Definition

In order to understand the problem of scheduling advance reservations with given laxities, let us consider the following definitions and assumptions.

Ready or Start Time (t_i): *Ready time* is the time at which the task associated with an on-demand request is sent to the queue of a resource and it is available to be picked up for execution. *Start time* of an advance reservation request is the time by which the application must make the task associated with the request available to the resource. A task can never start executing before its ready or start time.

Service Time (e_{ib}): *Service time* of a task *i* on a resource *b* is the time the task *i* takes to complete on the resource *b*. For example, in case of data transfer requests service time is the actual time the network takes to transfer the data from source to the destination. For compute tasks, service time is the time taken the by task to finish executing on a resource. As in [8, 9, 20], we assume that properties of the task such as the number of compute cycles in case of a compute task or bytes to be transferred in case of a network task are either known in advance or some statistics are available for their calculation. We also assume that when a task is executing on a resource, the resource does not execute any other task and hence service time of the task can be calculated by using appropriate statistics for the resource and the task.

Deadline (d_i): *Deadline* of a task is the absolute deadline before which the task must be finished. Once committed a resource ensures that a deadline associated with each AR is met. In case of on-demand requests, the deadline to finish the task is assumed to be infinity. We discuss in Section 5.3, how a resource can implement policies to prevent starvation of the ODs.

Laxity and Percentage Laxity: Laxity of a task i on a resource b is given by:

$$\text{Laxity} = d_i - t_i - e_{ib} \quad (3.1)$$

We define *percentage laxity L'* of a task i on a resource b as,

$$L' = \text{Laxity of task i on resource b} / e_{ib} \quad (3.2)$$

Non-Preemptable and Preemptable Tasks: *Non-preemptable* tasks are those that cannot be resumed later. Hence, if for some reason their execution is interrupted they need to be restarted from beginning. *Preemptable*

tasks, on the other hand can be resumed later from the point of interruption. We consider the effect of overheads associated with task preemption and its resumption on performance.

Scheduled-Time: Scheduled-time of a certain task is the time at which it is scheduled to start its execution under the current schedule of the resource. Different segments of a preemptable task may be scheduled to be executed at different times.

A Grid resource would receive requests from different applications for execution of different tasks. The problem for a given resource b to solve is:

Problem 1: *Given a set of tasks {i, j, …, k} and sets of ready or start times {t_i, t_j, …, t_k}, service times {e_{ib}, e_{jb}, …, e_{kb}} and deadlines {d_i, d_j, …, d_k}, schedule tasks {i, j, …, k} such that each task i starts executing after its ready or start time t_i and finishes before its deadline d_i.*

3.2 Algorithm for Scheduling Advance Reservation Requests with Given Laxities

Since in a Grid environment considered in the research new requests arrive continuously, the resource at the arrival of every new request tries to find a feasible schedule for the set of tasks already in schedule and the new request. If a feasible schedule is found, the request is committed and the new task is added to the set of scheduled tasks. Otherwise, the request is rejected. As each task finishes executing on the resource, it is removed from the set of scheduled tasks.

It has been shown previously [11] that Problem 1 is an NP-Hard problem. Algorithms presented in the real-time domain [11, 12] for solving this problem do not scale for large number of tasks as mentioned in Section 1 and their completion times grow exponentially with the number of tasks. These algorithms first find an initial schedule using one of the well-known strategies such as the earliest-deadline-first strategy and then improve on the initial solution to reduce the lateness of the task that realizes the value of maximum lateness. They use branch and bound technique to find the optimal solution.

In order to scale the algorithm for much larger number of tasks in the Grid domain, we present an algorithm in this paper that we call the SSS (Scaling through Subset Scheduling) Algorithm. Whenever a new request arrives, the SSS algorithm first finds all those tasks in the resource schedule that can affect the feasibility of the new schedule with the new request and then tries to work out a feasible schedule for only that subset of tasks S. This prevents the completion time of the algorithm from growing exponentially with the number of tasks. Since Problem 1 is an NP-Hard problem, it is possible to come up with pathological situations (when the set S equals the set of all tasks in the system) in which the completion time of the SSS algorithm would grow exponentially with the number of tasks. However, we argue that such situations are not likely to arise in the Grid domain.

By definition, *S contains at least all those tasks that if not included in S, the resulting new schedule is infeasible while a feasible schedule exists*. Thus, given the new request and the current resource schedule, the SSS algorithm always accommodates the new request in the resource schedule if it is feasible to do so. The discussion on finding that subset S is explained later in this section. Once the exact subset of tasks S is known, an initial solution can be worked out for that subset and the new request using any of the well-known strategies such as the earliest-deadline-first and the least-laxity-first. We have modified the SCHRAGE heuristic [24] used in [11], for scenarios involving both non-preemptable and preemptable tasks for finding an initial solution. The modified heuristic can be given as a sequence of following steps:

Step 1: Declare and initialize variable t and pT equal to 0.

Step 2: Find among S, all tasks with $t_i <= t$. If no such task exists, set t equal to the earliest t_i among all tasks in S.

Step 3: Among all tasks in S with $t_i <= t$, select a task with the earliest d_i. Break ties by selecting a task with the highest e_{ib}. If the task selected is preemptable, go to Step 5.

Step 4: Schedule the selected task at time t and set $t = t_i + e_{ib}$. Remove the task from S and go to Step 7.

Step 5: If e_{ib} is the execution time of the selected task and d_i its deadline, then among all tasks in S with $t_k <= t + e_{ib}$ find all tasks with deadline less than d_i. If no such task exists, go to Step 4. Otherwise, among that set of tasks find a task with the earliest t_k and set pT equal to its ready or start time.

Step 6: Place the task selected in Step 3 from t to pT. Preempt that task at time pT. Update the execution time of that task in S as $e_{ib} = e_{ib} – (pT – t)$. Set t = pT.

Step 7: If S is not empty, set pT equal to 0 and go to Step 2. Otherwise, the initial solution is complete.

The sequence of steps presented above shows that for non-preemptable tasks we do not remove the task until it finishes its execution on the resource even if during its execution a new task with a lower deadline becomes available for execution. This is because the task is non-preemptable. For preemptable tasks, we strictly use the earliest-deadline-first strategy.

The initial solution, as obtained above, consists of periods of continuous utilization of the resource called *blocks* with idle periods separating the blocks. If the initial solution is feasible, we accept the new request and update the overall schedule of the resource. If it is infeasible, we check by calculating lower bounds on the lateness of tasks using equations given in [11], whether the lateness of the task that misses its deadline by more time than any other task, known as *critical task*, can be improved. If the lateness of the critical task cannot be improved, the solution is called *optimal* and we reject the new request. If the solution is infeasible but not optimal, we find a subset of tasks that if scheduled after the critical task can improve its lateness. Such a subset is known as *generating set*. It can be shown that if the initial solution is calculated using SCHRAGE heuristic or any of its derivates, generating set consists of the set of non-preemptable tasks that belongs to the same block as the critical task and have deadlines greater than that of the critical task [11]. For each task in the generating set, we can find a new solution by scheduling that task after the critical task. This can be done by setting the start or ready time of that task equal to the start time of the critical task and then obtaining a new solution using the modified SCHRAGE heuristic. The resulting set of solutions where each solution corresponds to each task in the generating set, if infeasible, can further be improved by finding their critical tasks and corresponding generating sets. This process is continued and we get a tree of solutions with the initial solution at its root. Each solution in a tree is known as a *node*. This process of node generation is continued until either a feasible solution is found or the algorithm determines that no feasible solution exists.

In the algorithms presented in the real-time domain, all tasks are known in advance. However, in a Grid domain tasks arrive one by one. Hence, if the initial solution is infeasible but there exists more than one feasible solutions corresponding to more than one tasks in the generating set, there is no way to find out, selection of which task in generating set to produce the child node can get the maximum benefit later on. This is because some of the tasks might have already executed when a certain new request arrives. Selecting a task with the least execution time can maximize current utilization of the resource while selecting one with the highest deadline can decrease the probability of infeasibility of a potential future schedule. In order to balance the two factors – current utilization and flexibility in scheduling – the SSS algorithm first calculates the effective laxity of generating set tasks as seen at the start time of the critical task. If i is the critical task and j one of the tasks in the generating set, effective laxity of j at t_i can be given as $d_j – e_{jb} – t_i$. The algorithm then selects the task with the highest effective laxity to produce the child node. Note here that the SSS algorithm selects a task with the highest flexibility and hence minimizes the chance of an infeasible schedule. At the same time, if two tasks have equal deadlines, a task with a lower execution time would be selected to produce a child node. This would increase the current utilization of the resource.

As the number of tasks in the initial solution increases, the number of nodes in the search tree grows exponentially, making the process inscalable for large number of tasks. As mentioned earlier, we solve this problem by finding a suitable subset of tasks in the current resource schedule before obtaining the initial solution. In order to find that subset S, we first create a timeline of all tasks in queue and all advance reservations already committed, using their scheduled-times as determined during the previous run of the algorithm. Then we find out a suitable *start point* and *end point* on that timeline such that all tasks that belong to S lie between these two points. Initially, we set S = {New Task} and start point SP and end point EP as follows:

Initially,

S = {New Task}

SP = max {current system time, min {t_i for all tasks i in S}}

EP = max $\{d_i$ for all tasks i in S$\}$

Then we add tasks in S during each iteration according to the following rules:

- If any task i not in S has $t_i >= SP$ and its scheduled time < EP, then add i to S.
- If any task i not in S has scheduled time < EP and it finishes its execution after SP, then add i to S.

When S is obtained using the given methodology, all tasks that can affect the feasibility of the new schedule are included in S and we call S is complete. We further optimized the algorithm to reduce its completion time and we present the optimized algorithm next.

Let TIU be total time units encountered during the algorithm at which the resource was idle and let TIUS be total idle time units in the current idle segment of resource schedule. Let SP and EP be the current start point and end point respectively and let t_i, e_{ib} and d_i be the start time, execution time and deadline of the new task request.

Step 1: Initialize TIU and TIUS = 0 and SP and EP = t_i. Set S = {New Task}.

Step 2: Check if any of the following conditions is true,

A. If EP < d_i and the new task is a preemptable task and TIU equals e_{ib}.

B. EP is equal to the highest deadline among all tasks in S.

C. TIU = x*e_{ib}.

D. TIUS = y*e_{ib}.

Where x and y are the tuning parameters for the algorithm. If any of the above conditions is true, break out of the algorithm.

Step 3: If no task is scheduled at EP, increment TIU, TIUS and EP by 1 and go to Step 2. Otherwise, go to step 4.

Step 4: Set TIUS = 0. Call the task scheduled at EP, task j. Increment EP by 1 recursively as long as in the resource schedule task j is scheduled at EP. If task j is already in S, go to Step 2. Otherwise, add it to S, and check if t_j < SP. If it is not, go to Step 2. Otherwise, check if j has a finite deadline. If it does not have a finite deadline, set SP equal to scheduled-time of j and go to Step 2.

Step 5: If t_j < current system time, add all tasks scheduled between current system time and SP to S and set SP = current system time and go to Step 2. Otherwise, add all tasks that are scheduled between t_j and SP to S.

Step 6: Find all tasks scheduled between t_j and SP that have a start time less than t_j and have a finite deadline. If no such task exists set SP = t_j and go to Step 2.

Step 7: Set SP = t_j and select among the tasks found in Step 6, the one with the earliest start time and call it task j. Go to Step 5.

If S is obtained using the above algorithm, all tasks that are scheduled before start point but after the current system time are those that finish their execution before the start time of any of the tasks in S. Hence, they will not affect the scheduled-time of any of the tasks in S and hence of the new request. Note that, in Step 4 and Step 6 if the task that has a start time less than start point is an OD request we do not set SP equal to its start time. The reason is that an OD can be delayed indefinitely and hence it can never make an AR miss its deadline. Note that in the above algorithm, the terminating condition A is derived from the fact that a preemptable task can execute whenever the resource is idle. Hence, the resulting schedule will always be feasible. Thus by definition, S is complete. If condition B is true and the resulting schedule is an infeasible schedule, then the new task cannot be accommodated in the current schedule even if we include all the tasks that are scheduled after the end point. The reason is that the tasks that are scheduled after the end point can not improve the lateness of the critical task. Hence, for Condition B as well, S is complete. Parameters x and y (x > y) in condition C and D respectively are the tuning parameters for the algorithm and depends on the nature and size of tasks in the schedule. No tasks that are scheduled after the end point can delay the execution of any of the tasks in S. However, the tasks in S may affect the scheduled time of the tasks scheduled after the end point. If such a condition occurs then during the schedule update process, there exist conflicts in the resulting schedule. In order to avoid such conflicts, it is advisable to over estimate the end point when terminating through conditions C and D. This can be done by suitably selecting x and y. However, if a conflict still occurs, we re-obtain S and the new schedule. This process is repeated until no conflict occurs.

The SSS main algorithm can be written as a sequence of following steps:

Step 1: Whenever a new request arrives, obtain S using the methodology described earlier.

Step 2: Determine the initial solution using modified SCHARGE heuristic and check it for feasibility and optimality. If the initial solution is feasible, go to Step 5. If the initial solution is infeasible and optimal, go to Step 6. Otherwise, initialize a new vector of nodes v. Determine the generating set for the initial solution and add initial solution to v.

Step 3: If v is empty, go to Step 6. Otherwise, produce a child node of the last node in v. This is done by setting the start or ready time of the task with the highest effective laxity in the generating set equal to t_i. Remove that task from generating set of last node in v. If the resulting generating set size of that node is zero, remove that node from v.

Step 4: Obtain a solution for the new child node using modified SCHARGE heuristic and check it for feasibility and optimality. If the solution is feasible, go to Step 5. If the solution is infeasible but not optimal, determine its generating set and add the solution to vector v. Go to Step 3.

Step 5: Check if the new task is accepted would there be any schedule conflicts during the schedule update process. If there are schedule conflicts, re-obtain S using the algorithm given earlier setting initial value of end point equal to the current end point and go to step 2. Otherwise, accept the task and break out of the algorithm.

Step 6: Reject the new task.

Note that the SSS algorithm limit the growth of nodes with the number of tasks by, a) running the algorithm for a subset of tasks, b) by logically selecting and producing only one node at a time. Selecting and producing only one node at a time and removing the nodes that consists of optimal solutions reduces the number of open nodes in memory by a large factor. As opposed to at least thousands of nodes, in algorithms presented in [11, 12], for just 100 tasks, the SSS algorithm never had more than 120 nodes in memory for even 100,000 tasks. We discuss the performance of the SSS algorithm in Section 5.4.

3.3. Scheduling Non-Preemptable and Preemptable Advance Reservation Requests with Given Laxities and Given Overheads of Task Preemption

In order to study the specific scenarios in which overheads are associated with task preemption and its resumption, we make some modifications in the SSS algorithm. First, the SCHRAGE heuristic [24] is modified in the following manner to account for overheads in task preemption.

Step 1: Declare and initialize variables t and pT equal to 0.

Step 2: Find among S, all tasks with $t_i <= t$. If no such task exists, set t equal to the earliest t_i among all tasks in S.

Step 3: Among all tasks in S with $t_i <= t$, select a task with the earliest d_i. Break ties by selecting a task with the highest e_{ib}. If the task selected is preemptable, go to Step 5.

Step 4: Schedule the selected task at time t and set $t = t_i + e_{ib.}$ Remove the task from S and go to Step 8.

Step 5: If e_{ib} is the execution time of the selected task and d_i its deadline, then among all tasks in S with $t_k <= t + e_{ib}$ find all tasks with deadline less than $d_i - o$, where o is the overhead in terms of time in preempting and resuming the selected task. If no such task exists, go to Step 4. Otherwise, among that set of tasks find a task with the earliest t_k and set pT equal to its ready or start time.

Step 6: If $pT - t > o$, go to Step 7. Otherwise, among all tasks in S with $t_j <= t$ find all tasks with $e_{jb} <= (pT - t)$. If no such task exists set $t = pT$ and $pT = 0$ and go to Step 2. Otherwise, select the one with earliest d_j and go to Step 4.

Step 7: Place the task selected in Step 3 from t to pT. Preempt that task at time pT. Update the execution time of that task in S as $e_{ib} = e_{ib} - (pT - t)$. Set $t = pT + o$.

Step 8: If S is not empty, set $pT = 0$ and go to Step 2. Otherwise, the initial solution is complete.

The sequence of steps presented above is similar to the modified SCHRAGE heuristic that we use in the SSS algorithm. However, in Step 5 we do not preempt a selected task for another task whose deadline is earlier than that

of the selected task by a difference less than the overheads involved in preempting and resuming the task. This prevents unnecessary missing of deadlines as depicted in Fig. 1. In step 6, we make sure that we do not allocate the resource to a task i for less time than that spent in overheads in preempting that task later. Instead, we try to allocate the resource to another task j that can finish its execution before another task k with $d_k < d_i < d_j$ becomes available for execution.

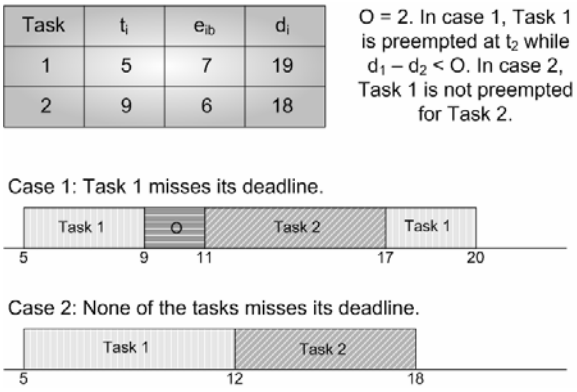

Figure 1: Scheduling Preemptable Tasks with Overheads

Secondly, we change the condition A in Step 2 of the procedure to determine subset S in the SSS algorithm in the following way.

A. If $EP < d_i$ and the new task is a preemptable task and TIU equals $e_{ib} + n*o$, excluding those segments of idle time units which have a size less than o. Here o is the overhead in terms of time in preempting the new task and n is the number of times the new task would be preempted if placed in the idle time units after t_i.

The rest of the algorithm remains unchanged.

4. EXPERIMENTAL SETUP

In order to study the performance of scheduling with advance reservations, we wrote a simulator. The scenario simulated, along with the assumptions about resources and Grid applications and their parameters, has been described in Section 2.1 and Section 3.1.

4.1 Performance Metric

The following performance metrics have been used for evaluation.

Probability of Blocking (P_b): P_b is the probability that a resource would reject a new task request because it cannot accommodate it in its current schedule. It can be calculated as:

$$P_b = \text{Total Number of Requests Rejected / Total Number of Requests} \quad (4.1)$$

Utilization (U): U of a resource is the fraction of the total time the resource is busy executing tasks. This does not include the time spent in overheads associated with preempting and resuming a preemptable task.

Mean Response Time of On-Demand Requests (R_{OD}): R_{OD} is the mean time between the submission of an on-demand request and its completion.

Mean Response Time of Advance Reservation Requests (R_{AR}): R_{AR} is the mean time between the start time of an advance reservation request and the time of completion of the task associated with that request.

Performance of the SSS Algorithm: For determining the performance of the SSS algorithm, the total number of nodes N produced to schedule a given number of tasks is determined. The maximum number of nodes in

memory at any point N_{max} is also determined, as it is this number that grows exponentially with the increase in number of tasks.

4.2 Workload Parameters

Following are the workload parameters of interest.

Service Times of Tasks: For some of our experiments, we have used a uniform distribution for the size of the tasks. A uniform distribution for modeling size of tasks has been used in other researches on advance reservation based scheduling (see [8], for example). The preliminary experiments demonstrated that for uniformly distributed size of the tasks, the shape of the curves obtained is independent of the mean as long as the ratio of the largest and the smallest size in the distribution remains the same. This research models service time of the tasks uniformly distributed between 10 minutes and 90 minutes with a mean of 50 minutes. For data transfer requests on a Grid based dynamic optical network, such a distribution of service times corresponds to data transfer requests between 10GByte and 100GByte on the OMNInet optical network [10, 25]. For compute tasks, such a distribution of execution times corresponds to compute tasks sizes between 500,000 Millions Instructions (MI) and 5,000,000 MI to be executed on a compute cluster capable of executing 1000 MI per second.

Uniform distribution has a relatively small co-efficient of variation. To study the effect of high variability in the sizes of the data transfer requests we have also used a bi-phase hyper-exponential distribution with a co-efficient of variation of 2. We choose the mean of hyper-exponentially distributed execution time E to be equal to 50 minutes which is equal to the mean E of the uniform distribution.

Mean Arrival Rate (λ): The preliminary experiments showed that a low arrival rate, which results in very poor utilization of the resource, does not generate interesting results while a very high arrival rate saturates the system by generating more requests than a resource can execute. Thus, a mean arrival rate λ of 0.014 requests per minute is used. The arrival process followed a Poisson distribution. If all requests are accepted then this arrival rate will result in a utilization of 0.7 (= 0.014 * 50). This value of maximum utilization is consistent with earlier research on advance reservation based scheduling [9].

Mean Time between the Arrival of an Advance Reservation Request and its Start Time (T_S): Unlike on-demand requests that submit their tasks to the resource at the time of request, advance reservation requests reserve the resource for some time in future. In our experiments, ARs can request to reserve the resource for any time between the current system time and the next 12 hours. This time between the arrival of an advance reservation request and its start time is modeled as a uniform distribution. Such a distribution is also used in other studies such as [9]. The results show that if we change T_S, it has a negligible effect on the overall system performance.

Proportion of Advance Reservations (PAR): PAR is the proportion of advance reservation requests in the total number of requests. We study system performance for different values of PAR by varying it between 0, where all requests are on-demand, and 1, where all requests are advance reservations, in the steps of 0.1. Total number of requests always remains the same.

Mean Percentage Laxity (L): A uniform distribution for the laxity of tasks is used with the lowest value of the distribution fixed at 0. Mean percentage laxity L is varied between 0% and 500%.

Task Preemption: For the experiments, we use both non-preemptable and preemptable tasks. Cases where overheads are associated with job preemption and its resumption are also studied.

Priority of Advance Reservations over On-Demand Requests: We have done the experiments for two types of scenarios with respect to the priority of advance reservations over on-demand requests. In the first type, ARs have high priority and they can delay any number of ODs arbitrarily. Such scenarios can result in large wait times for the on-demand requests. In the second type, ARs can delay ODs only to a certain degree. To simulate this, we associate a virtual deadline with each on-demand request equal to some multiple of its execution time (see Section 5.3).

Table 1 summarizes the cases studied in this paper. The values of the fixed parameters in these experiments are λ = 0.014 requests/minute, E = 50 minutes and T_S = 6 hours. Levels of PAR = 0 to 1 in the steps of 0.1 and levels of L = {0, 20, 60,100,150,200,500}%

Table 1: Cases Studied in the Experiments.

Case No.	Task Preemption	Distribution of Task Service Times	Virtual Deadline of ODS	Symbol
1	No	Uniform	Infinity	NUI
2	Yes, with and without overheads	Uniform	Infinity	PUI
3	50% non-preemptable tasks and 50% preemptable tasks with no overheads	Uniform	Infinity	HUI
4	No	Hyper-Exponential	Infinity	NHI
5	Yes, with and without overheads	Hyper-Exponential	Infinity	PHI
6	No	Uniform	$t_i + 6*e_{ib}$	NUF
7	Yes, with no overheads	Uniform	$t_i + 6*e_{ib}$	PUF

4.3 Accuracy of the Results

Tests were run long enough, and repeated multiple times, to obtain reasonably small confidence intervals for the performance metric. For uniformly distributed E, we obtained confidence intervals of ±1% for P_b, ±0.2% for U and ±1.5% for R_{OD} and R_{AR} at a confidence level of 99%. For hyper-exponentially distributed E, confidence intervals of ±3% for P_b, ±0.6% for U and ±5% for R_{OD} and R_{AR} were obtained at a confidence level of 95%. For all graphs to be presented in Section 5, the mean of the performance metric is plotted.

5. SYSTEM PERFORMANCE

5.1 Effect of Proportion of Advance Reservations and Laxity

This section presents the effect of the proportion of advance reservations and laxity on the performance of scheduling advance reservation and on-demand request in Grids. The results for cases 1 to 5 in Table 1 are presented in this section.

5.1.1. NUI Case: Non-preemptable Tasks with Uniformly Distributed Service Times and No Bounds on the Waiting Times of On-Demand Requests

Probability of Blocking: Figure 2(a) shows the effect of PAR on P_b. The figure shows that P_b increases with the increase in PAR. The reason is that unlike ODs that do not have any deadlines ARs have finite deadlines. As the proportion of ARs in the workload increases, it becomes increasingly difficult to schedule them while meeting their deadlines. Thus, more and more ARs are rejected which increases P_b. The results show that this increase is not linear and there exists a knee on the curve for a given value of laxity after which P_b increases more rapidly. Given the

mean laxity of the tasks, a resource can limit the ratio of requests it accepts as ARs equal to the value of PAR corresponding to the knee of the curve to keep P_b at reasonable levels.

Figure 2(b) shows that with an increase in L, P_b decreases substantially. As expected, laxity can improve the performance of scheduling AR requests significantly by providing more flexibility in task scheduling. The effect becomes more pronounced with the increase in percentage of ARs. Thus for 80% requests arriving as ARs, L = 200% can decrease P_b by 82.93% compared to the case in which ARs have no laxity. The curves shown in Figure 2(b) are characterized by a knee. The knee of the curve, that brings P_b to a small value, is reached for much smaller value of L compared to the one required to make P_b exactly equal to 0.

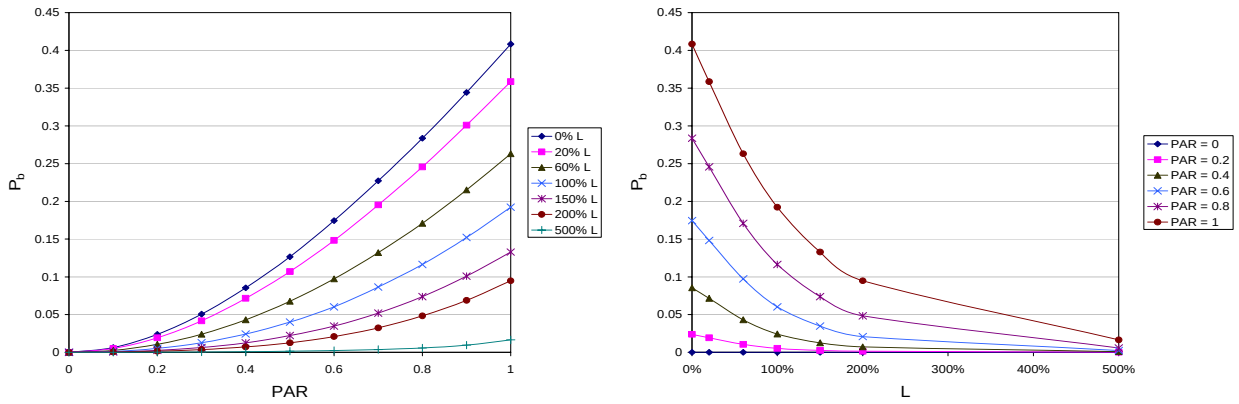

Figure 2: Probability of Blocking for the NUI Case
(a) Effect of PAR on P_b (b) Effect of L on P_b

Utilization: Figure 3(a) shows that with the increase in PAR, U decreases. The reason is that the increase in PAR increases P_b which decreases the number of tasks in the schedule. Figure 3(b) shows that by having some laxity in the tasks we can substantially counter this decrease in U with the increase in PAR. Just like the curves for P_b, the curves for U are also characterized by a knee that can act as a suitable operating point. In fact, P_b and U are related and the exact relation between them can be given as:

$$U = \lambda*(1 - P_b)*(R - W) \quad (5.1)$$

Where R is the mean response time of the tasks accepted and W their mean wait time. Note that R – W is not always equal to E since tasks with higher service times have a high probability of being rejected.

Since U and P_b are related, for the rest of the paper, instead of plotting both P_b and U, we present curves only for U.

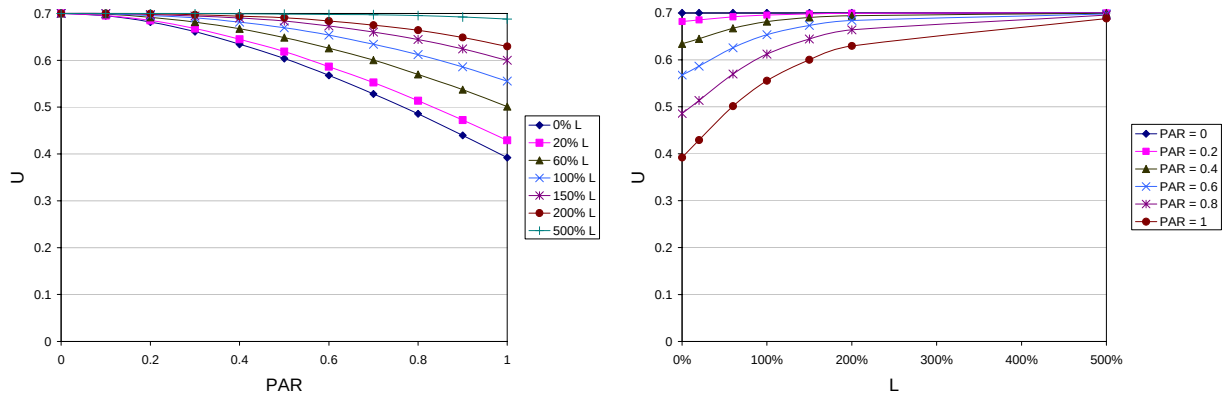

Figure 3: Utilization for the NUI Case
(a) Effect of PAR on U (b) Effect of L on U

Response Time of Advance Reservations and On-Demand Requests: Figure 4(a) shows that for smaller values of L, R_{OD} first increases with the increase in PAR until it reaches a maximum value and then it starts decreasing. The reason is that ARs can delay ODs arbitrarily. Thus, as PAR increases, more and more ODs are delayed for longer periods. This increases R_{OD} with the increase in PAR until it reaches its maximum value. As PAR increases further, due to an increase in P_b, U decreases significantly and this decreases the mean wait time of ODs. Hence, R_{OD} decreases. For a given value of PAR, an increase in L (from 0% to 100%) decreases R_{OD} by increasing the wait times of ARs. However, as L is increased beyond 100% there is no significant decrease in U with the increase in PAR. This increases mean wait times of both ARs and ODs. Hence, for large L values, R_{OD} is observed to increase with PAR.

Figure 4(b) shows that for L = 0%, R_{AR} decreases slightly with the increase in PAR. The reason is that for L = 0%, mean wait times of ARs is zero. As PAR increases, more and more schedule conflicts occur. Under this situation ARs with higher service times have a higher probability of being rejected. As more and more ARs with higher service times are rejected, the mean service times of ARs accepted decreases. This decreases the mean response time of ARs. As can be seen in the curves, the mean response time of ARs for L = 0% is actually lower than E. As L increases, mean wait times of ARs tend to increase which increases R_{AR} with L for the given value of PAR. For L values between 0% and 150%, R_{AR} does not change significantly with PAR as the increase in wait time of ARs with PAR is balanced by the decrease in the mean service times of ARs accepted. For larger L values, P_b does not increase significantly with PAR. This increases the mean wait times of ARs substantially and hence R_{AR} is observed to increase with PAR.

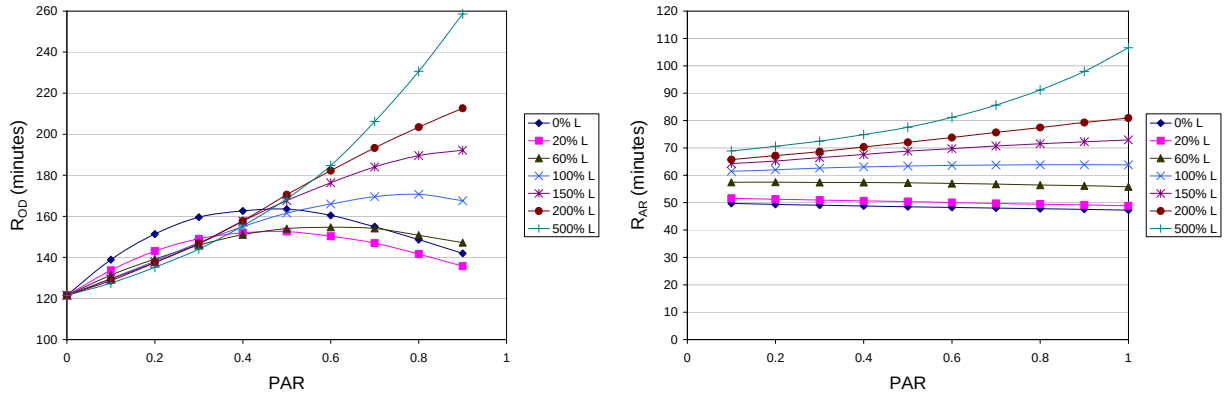

Figure 4: Response Time for the NUI Case
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

5.1.2. PUI Case: Preemptable Tasks with Uniformly Distributed Service Times and No Bounds on the Waiting Times of On-Demand Requests

In this section, we present results only for the cases in which no overheads are associated with task preemption and its resumption. Results with overheads will be presented in Section 5.2.

Utilization: Figure 5(a) and Figure 5(b) show the effect of PAR and L respectively on the utilization of the resource for preemptable tasks. The figures show that the behavior of preemptable tasks is similar to the one obtained for non-preemptable tasks and discussed in detail in Section 5.1.1. However, the figure shows that preemptable tasks results in slightly higher utilization than non-preemptable tasks. Detailed comparison of Figure 3 and Figure 5 is presented in Section 5.2.

Response Time of Advance Reservations and On-Demand Requests: Figure 6(a) and Figure 6(b) show the effect of PAR on R_{OD} and R_{AR} respectively for preemptable tasks without overheads. The figures show that PAR affects R_{OD} and R_{AR} of preemptable tasks in the same way as that of non-preemptable tasks in Section 5.1.1. Detailed comparison of Figure 4 and Figure 6 will be presented in Section 5.2.

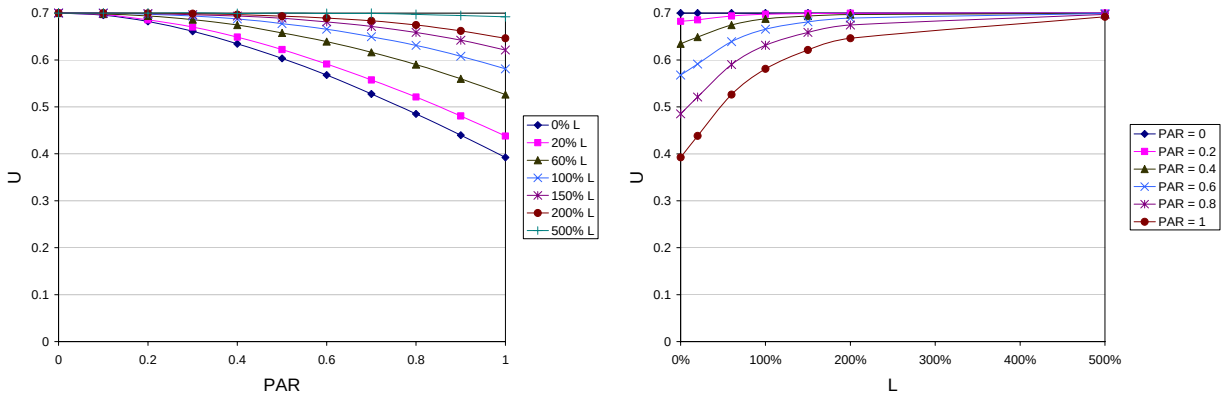

Figure 5: Utilization for the PUI Case
(a) Effect of PAR on U (b) Effect of L on U

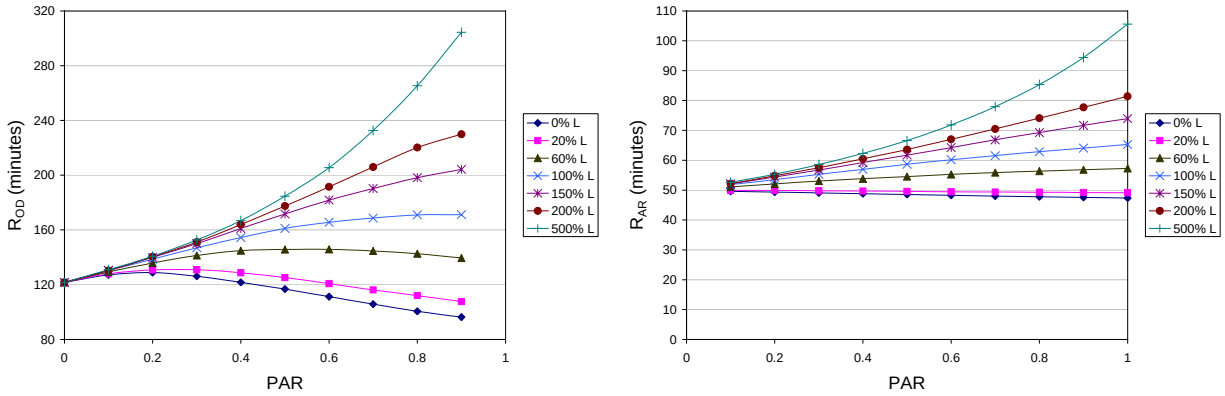

Figure 6: Response Time for the PUI Case
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

5.1.3. HUI Case: 50% Hybrid Tasks with Uniformly Distributed Service Times and No Bounds on the Waiting Times of On-Demand Requests

The results obtained with 50% preemptable and 50% non-preemptable tasks are shown in Figure 7 and Figure 8. The figures show that results for this case are approximately the mean of the results obtained in Section 5.1.1 and Section 5.1.2. This shows that when non-preemptable and preemptable tasks are combined, contribution of each type of tasks is approximately equal to its proportion among all the tasks.

5.1.4. NHI Case: Non-preemptable Tasks with Hyper-Exponentially Distributed Service Times and No Bounds on the Waiting Times of On-Demand Requests

Utilization: Figure 9(a) and Figure 9(b) respectively show the effect of PAR and L on system utilization when non-preemptable tasks with hyper-exponentially distributed service times are used. The curves are similar to the one obtained with uniformly distributed service times in Section 5.1.1 and just like the curves in Figure 3(a) and Figure 3(b), these curves are characterized by a knee. However, the results show that for the given value of PAR and L, U for hyper-exponentially distributed service times is lower than that with uniformly distributed service times. The difference in U increases with PAR and can be as much as 10.31%. This shows that the variability in the task service times can significantly affect system performance and higher variability results in more schedule conflicts and hence lower system utilization.

Response Time of Advance Reservations and On-Demand Requests: Figure 10(a) shows that R_{OD} with hyper-exponentially distributed service times increases almost linearly with the increase in PAR. The reason is that as PAR increases, more ODs are delayed to accommodate ARs which increases R_{OD}. Unlike the curves in Figure 4(a), where

for lower L values R_{OD} decreases with PAR after reaching a certain maximum value, R_{OD} with hyper-exponentially distributed service times in Figure 10(a) keeps on increasing with PAR. In addition, for a given value of PAR and L, R_{OD} in Figure 10(a) is substantially higher than that in Figure 4(a). The difference increases with PAR and can reach up to 800%. This shows that high variability in task sizes can severely affect the response times of the on-demand requests.

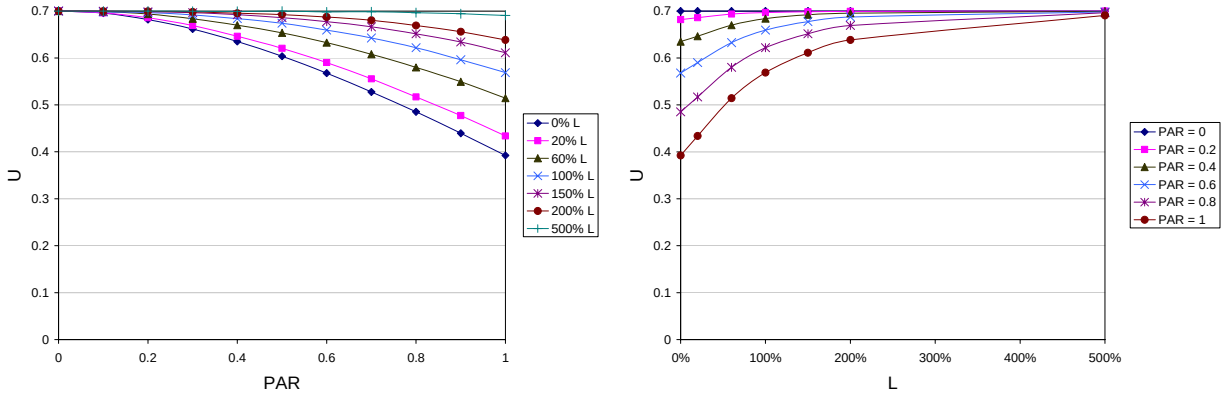

Figure 7: Utilization for the HUI Case
(a) Effect of PAR on U (b) Effect of L on U

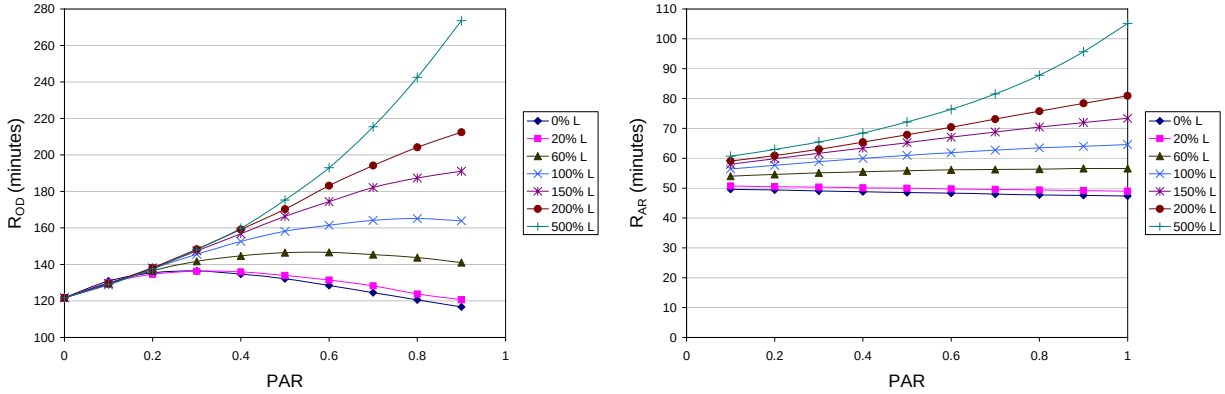

Figure 8: Response Time for the HUI Case
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

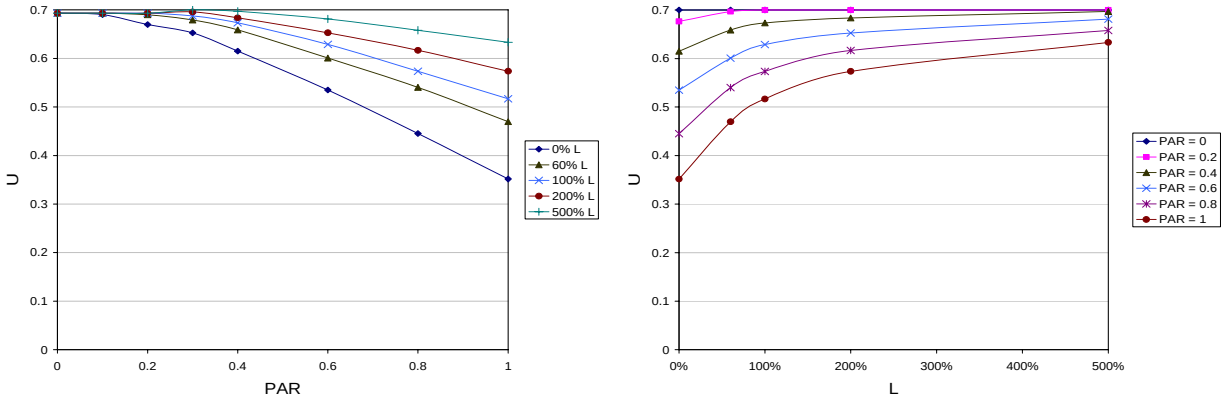

Figure 9: Utilization for the NHI Case
(a) Effect of PAR on U (b) Effect of L on U

The curves in Figure 10(b) for response times of hyper-exponentially distributed service times of ARs are similar to those in Figure 4(b) with uniformly distributed service times explained in detail in Section 5.1.1. Comparison of Figure 10(b) and Figure 4(b) shows that for lower L values R_{AR} for the NHI case is smaller than NUI case and the difference can be as much as 21.92%. The reason is that for smaller L values in NHI case there are more schedule conflicts due to variability in task sizes. Under such conditions, more tasks with large services times are rejected, substantially decreasing the overall execution time of the ARs accepted. Since for low L values mean wait time of ARs is small, substantial decrease in mean execution times of the ARs accepted significantly decreases response times of the ARs.

For large L values, however, R_{AR} for NHI is higher than that with NUI case and the two can differ by as much as by 45.64%. The reason is that for high L values wait times of ARs for the NHI case are substantially higher than that for NUI case due to high variability in task sizes.

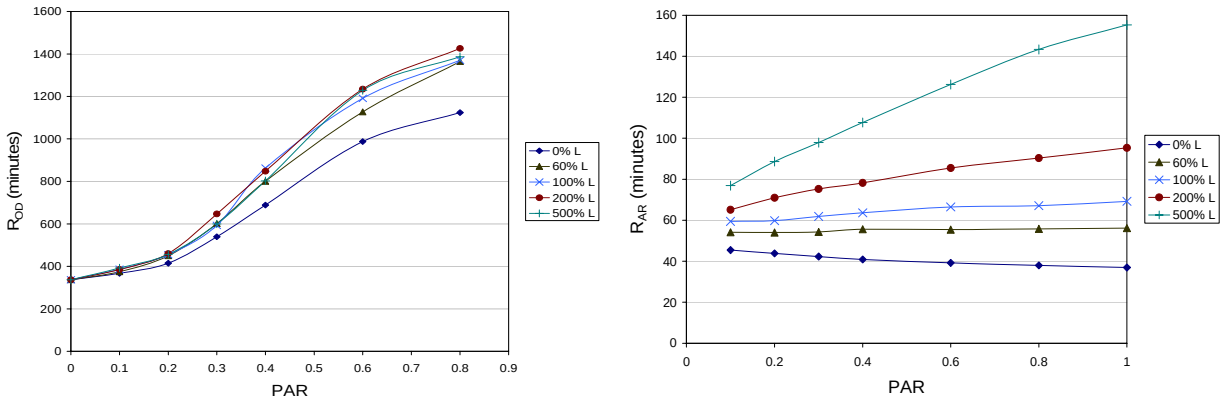

Figure 10: Response Time for the NHI Case
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

5.1.5. PHI Case: Preemptable Tasks with Hyper-Exponentially Distributed Service Times and No Bounds on the Waiting Times of On-Demand Requests

Utilization: Figure 11(a) and Figure 11(b) respectively show the effect of PAR and L on system utilization when preemptable tasks with hyper-exponentially distributed service times are used. The curves are similar to the one obtained with uniformly distributed service times in Section 5.1.2 and just like the curves in Figure 5(a) and Figure 5(b), these curves are characterized by a knee. However, the results show that for the given value of PAR and L, U for hyper-exponentially distributed service times is lower than that with uniformly distributed service times. This is because of high variability in task sizes as discussed in Section 5.1.4.

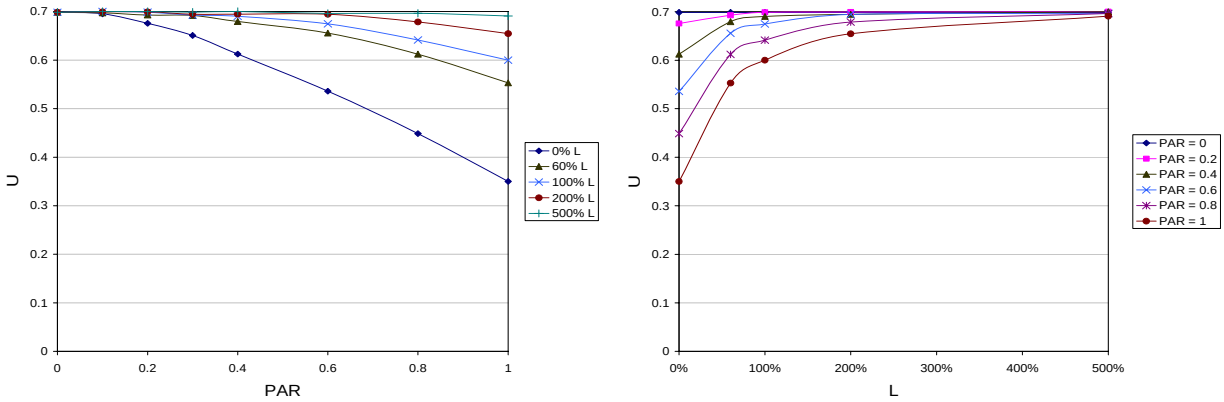

Figure 11: Utilization for the PHI Case
(a) Effect of PAR on U (b) Effect of L on U

Response Time of Advance Reservations and On-Demand Requests: Figure 12(a) shows that the curves for R_{OD} for the PHI case are similar to those for PUI case and discussed in detail in Section 5.1.2. However, R_{OD} for PHI case is significantly higher than that for PUI case due to variability in task sizes as discussed in Section 5.1.4.

The curves in Figure 12(b) for response times of hyper-exponentially distributed service times of ARs are similar to those in Figure 6(b) with uniformly distributed service times explained in detail in Section 5.1.2. Comparison of Figure 12(b) and Figure 6(b) yield the same conclusions as the comparison of Figure 10(b) and Figure 4(b) discussed in detail in Section 5.1.4.

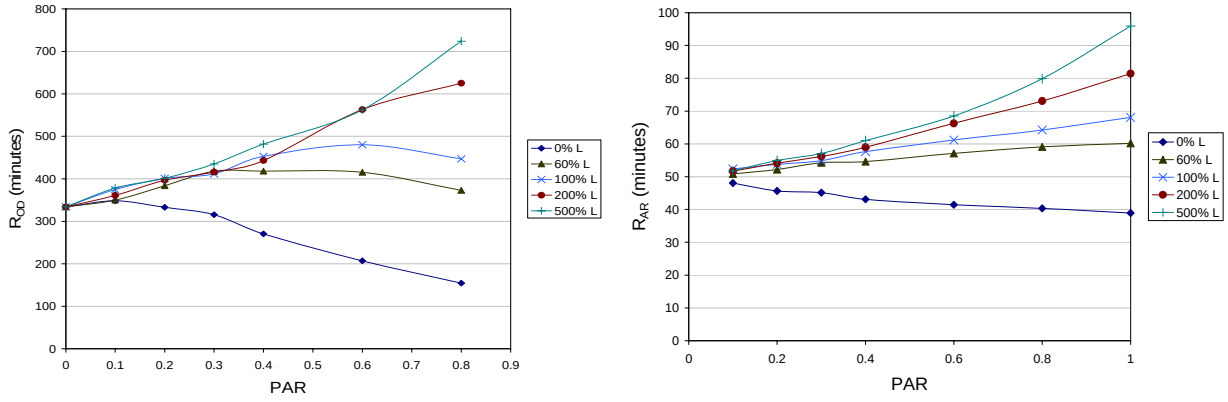

Figure 12: Response Time for the PHI Case
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

5.2. Effect of Preemption

This section presents the effect of task preemption on system performance. For network tasks, segmentation can be thought of as analogous to preemption for compute tasks. Segmentation means the ability to segment data into multiple requests and schedule different chunks of data at different times.

In this section, we compare the results for the NUI case in Table 1 where tasks are non-preemptable with the PUI case where tasks are preemptable. Similarly, the NHI case is compared with the PHI case. For the PUI and PHI cases, we not only consider the scenario in which no overheads are associated with preemption but also introduce overheads O as a percentage of mean service times of the tasks E. O represents time spent in overheads associated with task preemption and its resumption. For example, if tasks represent data transfers on a lightpath then O represents time spent in path tear down and network reconfiguration. When a client accesses the optical network through a control plane mechanism such as Optical Dynamic Intelligent Network Service (ODIN) [26], the total time spent in overheads is just less than 1 minute [10] including the ODIN server processing delay, path tear down and network reconfiguration. However, if the client accesses the optical network through a customer portal the total time spent in overheads can be order of few minutes. This is because in addition to the delays incurred in the control plane mechanisms there are delays at the portal server that handles billing and account aspects. We vary O between 0% of E (0 minutes) representing the ideal case to 30% of E (15 minutes) representing the worst-case scenario.

We first present the results obtained for PAR = 0.4 as we expect less than half of the requests to be ARs in future Grid based scenarios. Results for other values of PAR are presented next. In Figure 13, Figure 14 and Figure 15, we compare U, R_{OD} and R_{AR} respectively, for preemptable scenarios with those of non-preemptable scenarios. The curves show the percentage increase in the performance metric when tasks are preempted compared to the case when they are not preempted.

Utilization: Figure 13(a) shows that for a given L, U in the PUI case where tasks are preempted is higher than that with the NUI case. The reason is that segments of tasks can be scheduled on small slots of idle periods where non-preemptable requests cannot be accommodated. This decreases P_b and increases U. However, the results show that for uniformly distributed service times of tasks with low co-efficient of variation this effect is not very pronounced for PAR = 0.4 and the effect diminishes as overheads are introduced. The difference in utilization in Figure 13(a)

peaks at 1.05% when no overheads are associated with preemption. Results presented in Figure 13(b) for the hyper-exponentially distributed service times are similar to Figure 13(a) but show that when the variability between the service times is large, the improvement in U is much more pronounced with a peak at 3.15%.
Figure 13 also show that difference in utilization is a non-linear function of laxity with maxima near L = 70%. The reason is that with L = 0%, even in the PUI and PHI cases none of the requests can be preempted, as preemption for a non-zero time would make them miss their deadline. As L increases, more and more requests can be preempted for non-zero times thus accommodating more requests and the difference in U increases until it reaches its maximum value. For very high L values, most of the requests even with non-preemptable scenarios can be successfully scheduled and hence the option of preemption does not bring a substantial difference in U. This shows that laxity can be exchanged for segmentation for achieving high utilization.

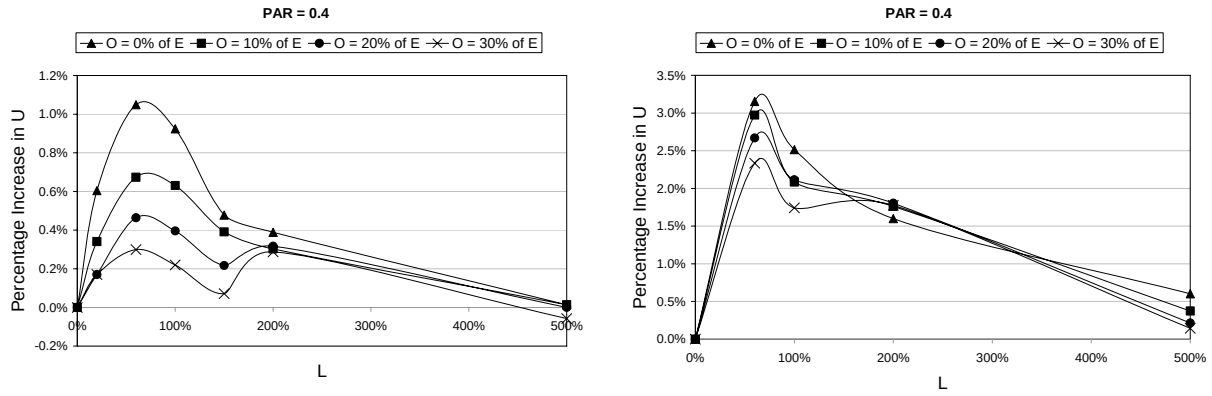

Figure 13: Effect of Task Preemption on Utilization for PAR = 0.4
(a) Effect on U for Uniformly Distributed Service Times
(b) Effect on U for Hyper-Exponentially Distributed Service Times

Response Time of Advance Reservations and On-Demand Requests: Figure 14(a) shows that for lower L values, R_{OD} in the PUI case is lower than that in the NUI case. This is because in the PUI case, the OD requests need not be delayed to accommodate ARs as their segments can be scheduled in small idle slots in the resource schedule. However, with the increase in L difference in R_{OD} starts decreasing until it becomes zero and then it starts increasing in the opposite direction i.e. R_{OD} for the PUI case starts becoming higher compared to that in the NUI case. On the other hand, in Figure 15(a) with the increase in L, R_{AR} with the PUI case becomes smaller and smaller compared to that with the NUI case. This shows that in the PUI case many OD requests with infinite deadlines are preempted by the scheduler for AR requests with smaller deadlines. This is because the SSS scheduler gives priority to the tasks with earlier deadlines while finding an initial solution. On the other hand, in the NUI case as L increases, comparatively less number of ODs needs to be re-scheduled at a later time to accommodate ARs. For higher L values, this results in R_{OD} for the PUI case to be higher than that for the NUI case while R_{AR} for the PUI case to be smaller than that for the NUI case.

The results in Figure 14(b) and Figure 15(b) are similar to those in Figure 14(a) and Figure 15(a) respectively. However, in Figure 14 (b), initially R_{OD} with the PHI case is so small in comparison to the NHI case that although the difference between R_{OD} with the PHI case and that with the NHI case becomes smaller and smaller with the increase in L, R_{OD} with the PHI case never becomes higher than that in the NH case even when 30% overheads are associated with preemption.

The results thus show that for PAR = 0.4, task preemption can result in an improvement in performance in terms of higher unitization and lower response times if the co-efficient of variation of the size of data transfer requests is high. However, preemption is not justified for uniformly distributed task service times with low co-efficient of variation.

Figure 16, Figure 17 and Figure 18 compares U, R_{OD} and R_{AR} respectively, for preemptable scenarios with those of non-preemptable scenarios for PAR = 0.1. The results are similar to those for PAR = 0.4 but shows that for PAR = 0.1, improvement in U and R_{OD} are smaller compared to PAR = 0.4. On the other hand, results in Figure 19, Figure

20 and Figure 21 for PAR = 1.0 shows that for high PAR values the effect of preemption becomes more pronounced with improvement in utilization of up to 5.25% for uniformly distributed service times and up to 18.1% for hyper-exponentially distributed service times. This suggests that in addition to the characteristics of the tasks such as the distribution of service times, proportion of advance reservations among the total requests should also be taken into account to decide about task preemption in compute tasks and data segmentation in network tasks.

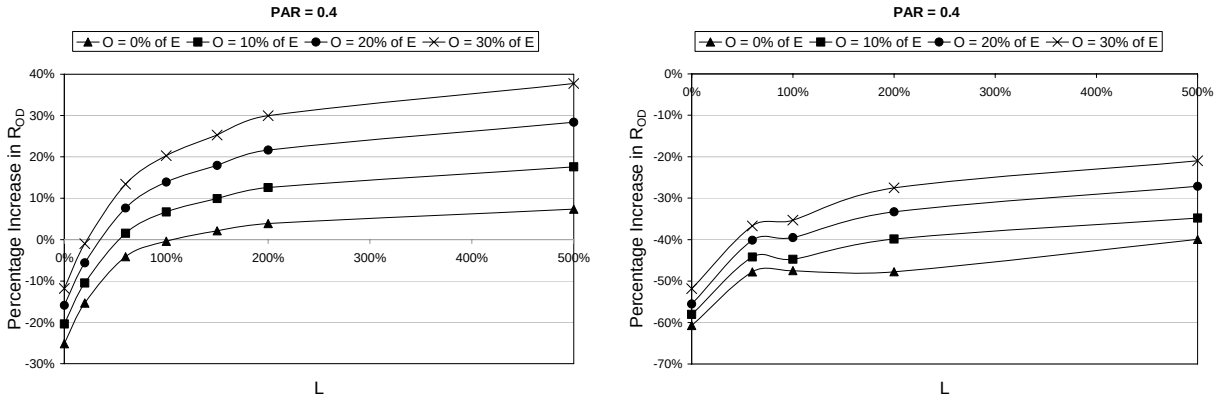

Figure 14: Effect of Task Preemption on Response Time of On-Demand Requests for PAR = 0.4
(a) Effect on R_{OD} for Uniformly Distributed Service Times
(b) Effect on R_{OD} for Hyper-Exponentially Distributed Service Times

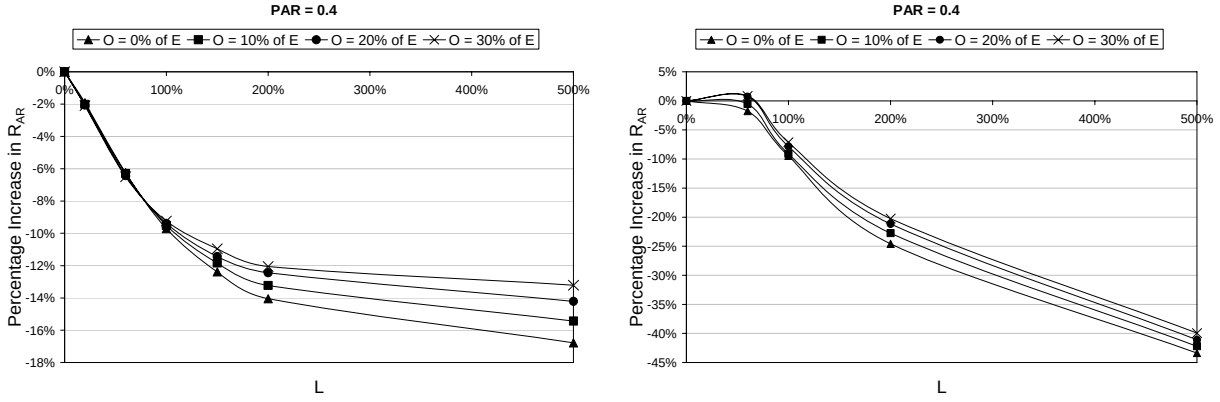

Figure 15: Effect of Task Preemption on Response Time of Advance Reservation Requests for PAR = 0.4
(a) Effect on R_{AR} for Uniformly Distributed Service Times
(b) Effect on R_{AR} for Hyper-Exponentially Distributed Service Times

5.3. Preventing Starvation of On-Demand Requests

Results in Section 5.1 show that with the increase in PAR, response time of on-demand requests increases significantly. The effect is more pronounced for higher L values. A very high response time for ODs will encourage all users to submit their tasks as ARs which would increase PAR. For lower L values, this would decrease their response time but at a cost of low utilization of the resource (see for example, Figure 3 and Figure 5). For higher L values, with the increase in PAR, there would not only be a slight decrease in U but also a tremendous increase in the response time of all requests (see for example, Figure 4 and Figure 6). In order to prevent these situations resulting from potential starvation of on-demand requests a resource should guarantee a *reasonable* response time for the tasks submitted as ODs. To ensure this a resource can associate a virtual deadline with all ODs and during the scheduling process can make sure that *most* of ODs meet their deadlines. In order to study this, we associated a hard deadline with every OD and ensured that each OD accepted meets its deadline. The deadline chosen for ODs was equal to 6 times its execution time as this is equivalent to 500%L for ARs. This would provide us with a direct comparison of ODs and ARs for equal deadlines.

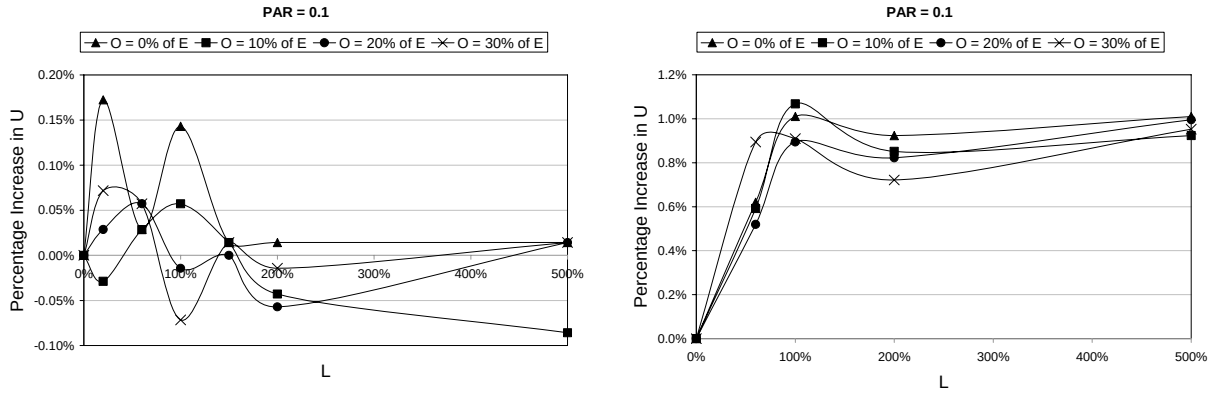

Figure 16: Effect of Task Preemption on Utilization for PAR = 0.1
(a) Effect on U for Uniformly Distributed Service Times
(b) Effect on U for Hyper-Exponentially Distributed Service Times

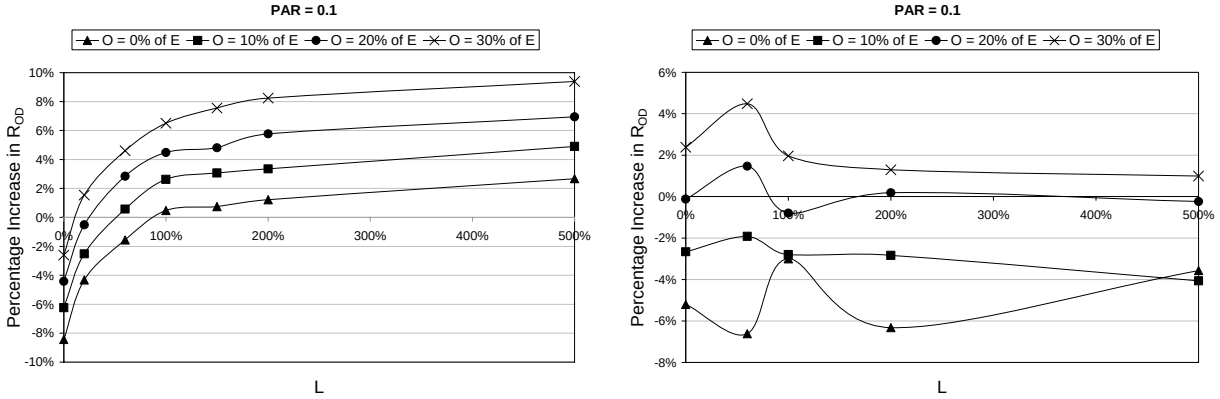

Figure 17: Effect of Task Preemption on Response Time of On-Demand Requests for PAR = 0.1
(a) Effect on R_{OD} for Uniformly Distributed Service Times
(b) Effect on R_{OD} for Hyper-Exponentially Distributed Service Times

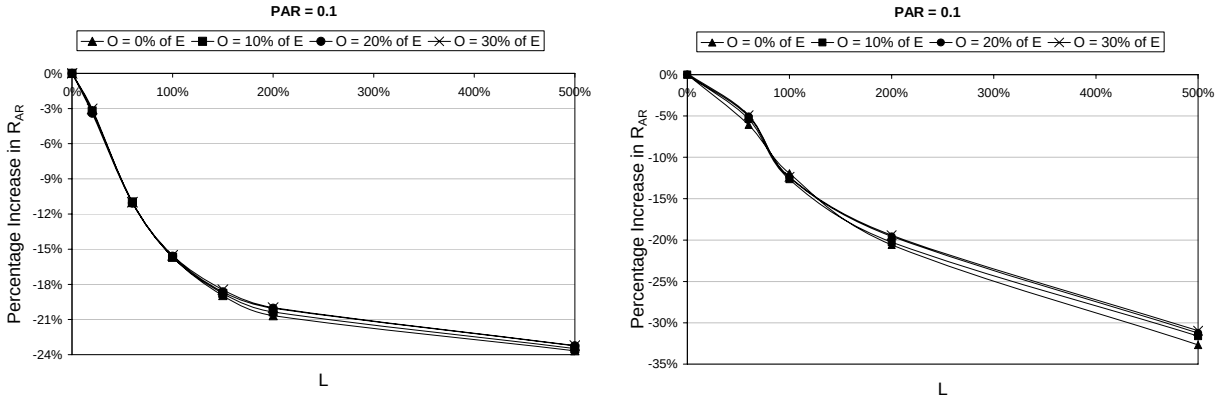

Figure 18: Effect of Task Preemption on Response Time of Advance Reservation Requests for PAR = 0.1
(a) Effect on R_{AR} for Uniformly Distributed Service Times
(b) Effect on R_{AR} for Hyper-Exponentially Distributed Service Times

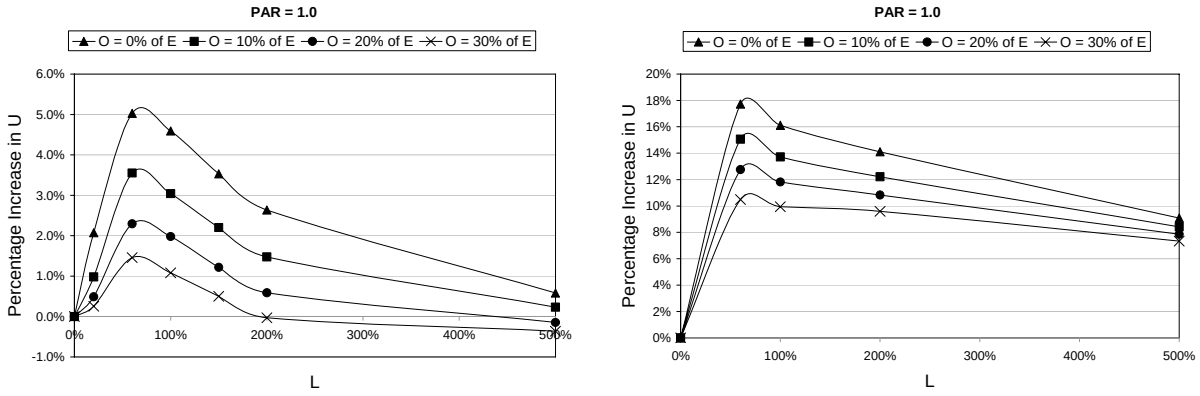

Figure 19: Effect of Task Preemption on Utilization for PAR = 1.0
(a) Effect on U for Uniformly Distributed Service Times
(b) Effect on U for Hyper-Exponentially Distributed Service Times

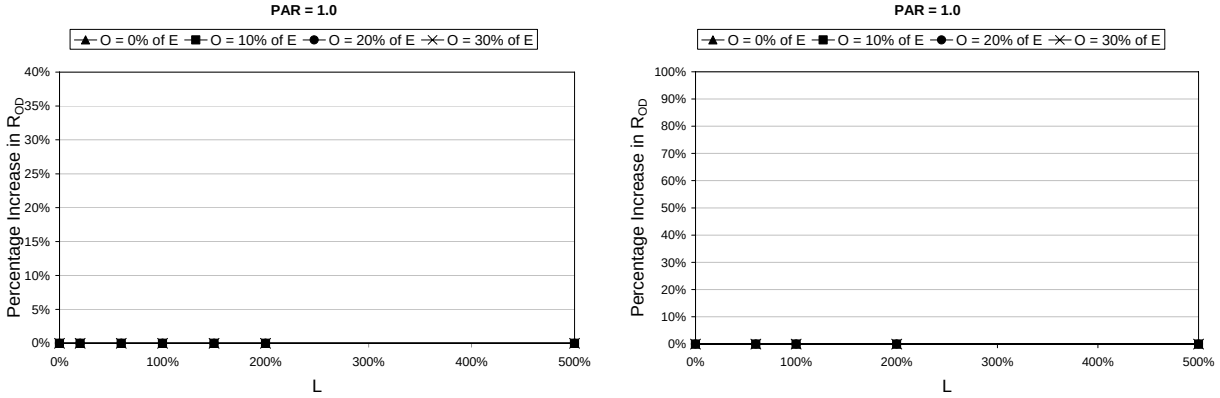

Figure 20: Effect of Task Preemption on Response Time of On-Demand Requests for PAR = 1.0
(a) Effect on R_{OD} for Uniformly Distributed Service Times
(b) Effect on R_{OD} for Hyper-Exponentially Distributed Service Times

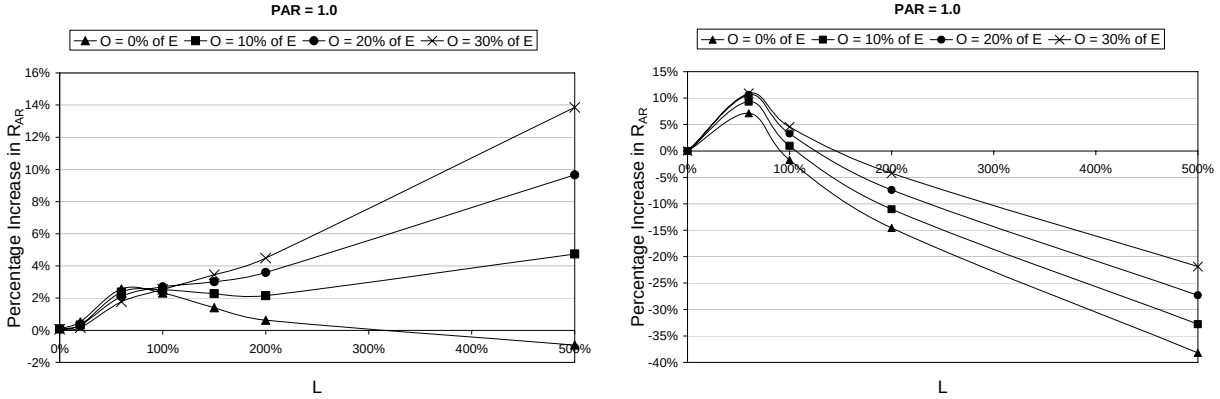

Figure 21: Effect of Task Preemption on Response Time of Advance Reservation Requests for PAR = 1.0
(a) Effect on R_{AR} for Uniformly Distributed Service Times
(b) Effect on R_{AR} for Hyper-Exponentially Distributed Service Times

In this section, we present the results obtained by associating virtual deadlines with on-demand requests. The results correspond to cases 6 and 7 in Table 1.

5.3.1 NUF Case: Non-preemptable Tasks with Uniformly Distributed Service Times and Bounds on the Waiting Times of On-Demand Requests

Utilization: Figure 22(a) shows that for the NUF case, PAR affects U in a similar fashion as it affects U in the NUI case (discussed in detail in Section 5.1.1). Figure 22(b) shows the percentage decrease in U in the NUF case compared to the NHI case. The figure shows that the utilization achieved with a given PAR and L in this case is slightly lower than that in the NUI case. This is the cost a resource incurs in reducing the response time of ODs. However, Figure 22(b) shows that the difference in U is for a given PAR and L is never more than 3.11% and it almost reduces to zero for higher L values.

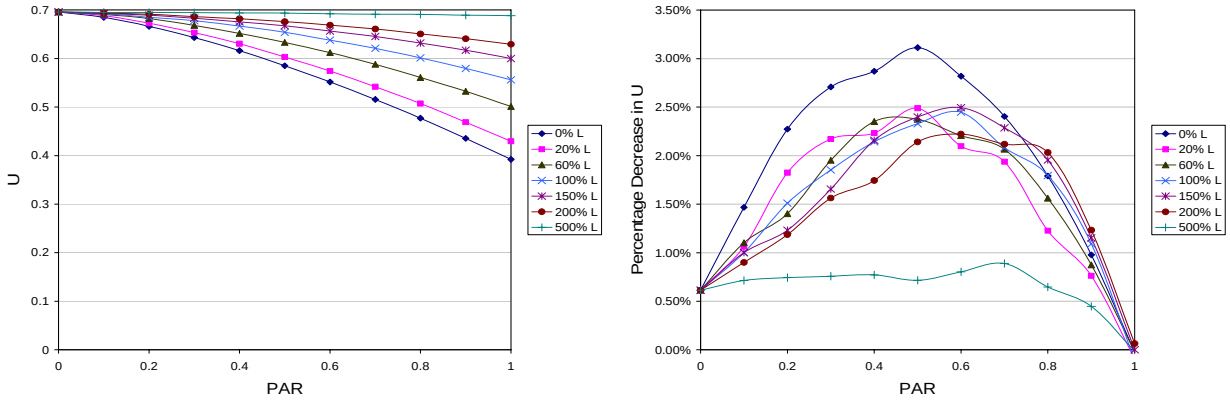

Figure 22: Effect of Associating a Virtual Deadline with ODs on U for Non-Preemptable Tasks
(a) Effect of PAR on U for the NUF Case
(b) Percentage Decrease in U in the NUF Case Compared to the NUI Case

Response Time of Advance Reservations and On-Demand Requests: Figure 23(a) shows the effect of PAR on R_{OD}. The figure shows that for L values less than 100% the behavior is similar to that obtained in the NUI case shown in Figure 4(a). However, R_{OD} in Figure 23(a) for higher L values is significantly lower than R_{OD} in Figure 4(a). This is shown in Figure 23(b) where the percentage decrease in R_{OD} in the NUF case compared to that in the NUI case is plotted. The difference in R_{OD} in Figure 23(b) increases with L for a given value of PAR. For L = 500% and PAR = 0.9, the difference in R_{OD} is as much as 60.22%. Unlike in Figure 4(a) where for L values above 150%, R_{OD} for a given value of PAR increases with L, in Figure 8(a) R_{OD} actually decreases with L for a given value of PAR. This is because as L increases, deadlines of ARs become comparable to that of ODs and hence ODs start getting as much priority as ARs by the SSS scheduler which gives priority to tasks with earlier deadlines when finding an initial solution.

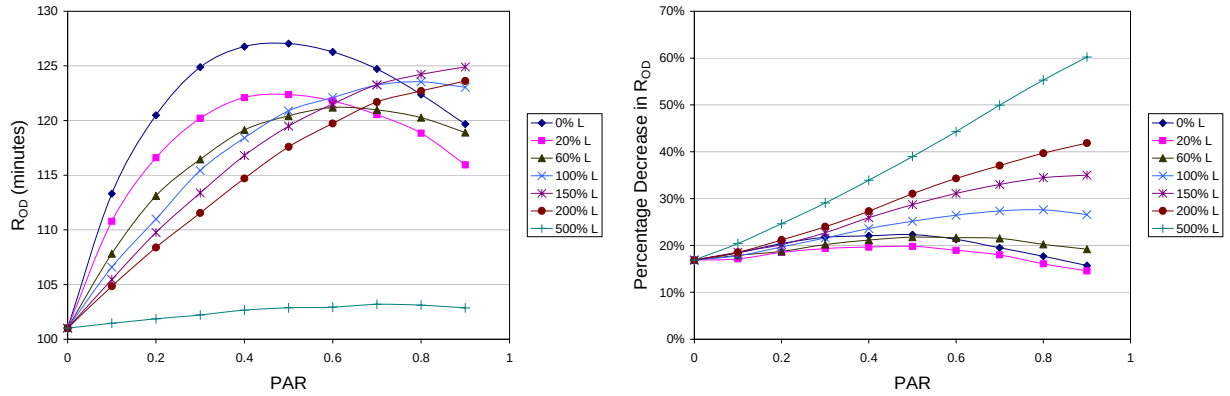

Figure 23: Effect of Associating a Virtual Deadline with ODs on R_{OD} for Non-Preemptable Tasks
(a) Effect of PAR on R_{OD} for the NUF Case
(b) Percentage Decrease in R_{OD} in the NUF Case Compared to the NUI Case

Effect of PAR on R_{AR} is given in Figure 24 which shows that for L values up to 200%, R_{AR} for a given PAR and L is approximately the same as in Figure 4(b). However, for L = 500% with low PAR values, R_{AR} is significantly higher (up to 50%) than that in the NUI case. For higher PAR values, the difference approaches zero. For L = 500%, R_{AR} in Figure 24 is almost equal to R_{OD} in Figure 23(a) for a given value of PAR. This shows that if tasks have equal deadlines there is no significant advantage in terms of reduction in response time by reserving the resources in advance. From this result, we can also deduce that changing the minimum time between the arrival of an advance reservation request and its start time does not change the overall performance.

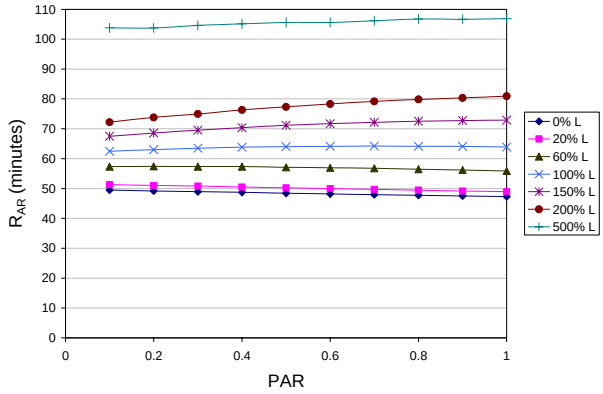

Figure 24: Effect of PAR on R_{AR} for the NUF Case

5.3.2. PUF Case: Preemptable Tasks with Uniformly Distributed Service Times and Bounds on the Waiting Times of On-Demand Requests

The results obtained by associating virtual deadlines with ODs for preemptable tasks are shown in Figure 25, Figure 26 and Figure 27. The results are similar to those discussed in Section 5.3.1 for non-preemptable tasks.

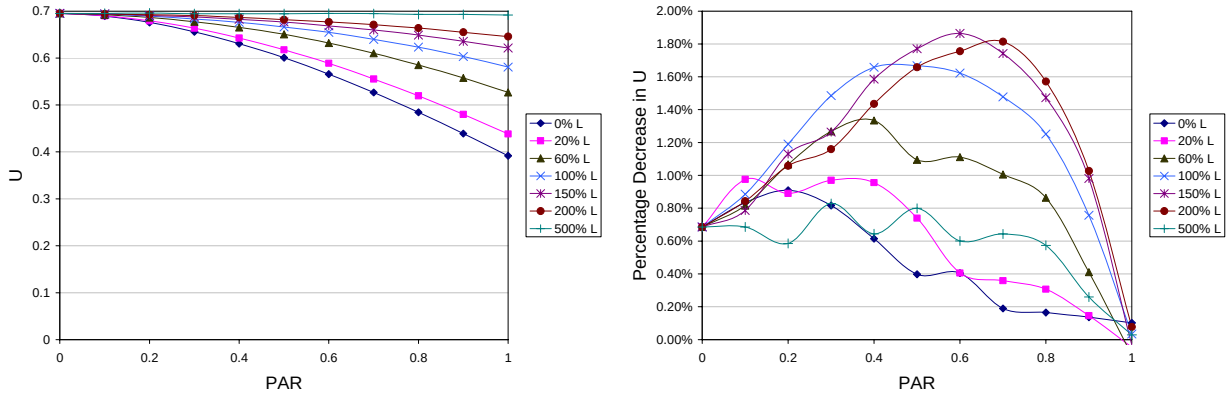

Figure 25: Effect of Associating a Virtual Deadline with ODs on U for Preemptable Tasks
(a) Effect of PAR on U for the PUF Case
(b) Percentage Decrease in U in the PUF Case Compared to the PUI Case

The results presented in this section thus show that associating a virtual deadline with ODs can effectively decrease the response time of ODs without significantly decreasing U or substantially increasing R_{AR}.

5.4 Performance of the SSS Algorithm

The SSS algorithm for scheduling advance reservation requests with laxity successfully scheduled hundreds of thousands of tasks. The results show that the total number of nodes produced to schedule given number of tasks depends on PAR and L in addition to the properties of the tasks such as the variability in service times of the tasks. For the NUI case in Table 1, the total numbers of nodes N produced to schedule 100,000 tasks are shown in Figure

28. The figure shows that comparatively, large number of nodes N is produced for PAR values between 0.4 to 0.6. However, even for these values of PAR, with the workload parameters in our experiments, average number of nodes produced was less than 1 node per task. Maximum numbers of nodes open at one time N_{max} for 100,000 tasks with uniformly distributed service times never exceeded 21.

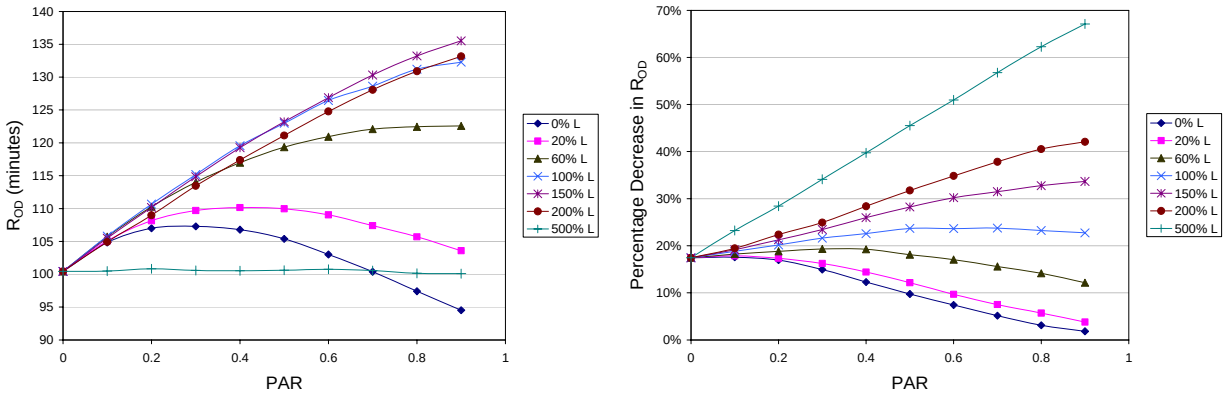

Figure 26: Effect of Associating a Virtual Deadline with ODs on R_{OD} for Preemptable Tasks
(a) Effect of PAR on R_{OD} for the PUF Case
(b) Percentage Decrease in R_{OD} in the PUF Case Compared to the PUI Case

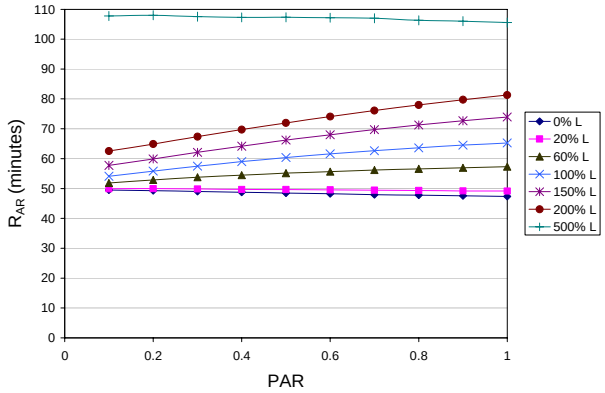

Figure 27: Effect of PAR on R_{AR} for the PUF Case

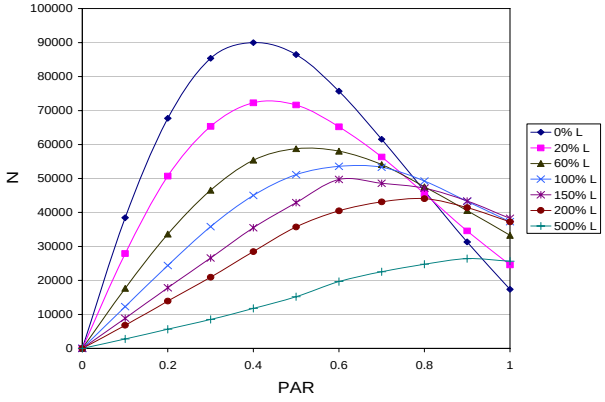

Figure 28: Effect of PAR on the Total Number of Nodes N Produced to Schedule 100,000 Tasks in the NUI Case

For uniformly distributed service times of the tasks, it took our simulator, written in JAVA (JDK 1.4.2) and running on a RedHat Linux 7.2 machine with 2.4 GHz Pentium IV processor and 512 MB of RAM, on average 24 seconds to schedule 100,000 tasks. This converts to per task average of 0.24 milliseconds. Note that in addition to the SSS algorithm, several other modules, such as the workload generator, for the simulator were also running on the machine at the same time. Therefore, the actual time taken by the SSS algorithm to schedule a task is even smaller. Since in a Grid environment each task on average is of the order of at least few minutes, we can conclude that dynamic scheduling of tasks with the SSS algorithm is feasible in Grids even for a fairly high arrival rate.

Comparatively, larger number of nodes N was produced for the hyper-exponentially distributed service time. Nevertheless, even for such a distribution, maximum numbers of nodes open at one time N_{max} for 100,000 tasks never exceeded 120. Thus, N_{max} and N produced by the SSS algorithm are much smaller than those of other algorithms where at least thousands of nodes are produced to schedule just 100 tasks [11, 12].

6. CONCLUSIONS

In this paper, we argued that the notion of laxity in the reservation window can help improve system performance when scheduling with advance reservation and on-demand requests. The paper presented the SSS algorithm for an NP-Hard problem of scheduling on-demand and advance reservation request with laxities. The results show that the SSS algorithm is scalable and can successfully schedule hundreds of thousands of tasks even for a fairly high arrival rate. The number of nodes produced to schedule large number of tasks in the SSS algorithm is many times smaller than that produced by other algorithms in the real-time domain for solving similar problems.

The performance results obtained using the algorithm show that laxity in the reservation window can significantly improve system performance by reducing the probability of blocking and increasing utilization. The effect is more pronounced for the cases where proportion of advance reservations is high. When P_b is plotted against PAR, there exists a knee on the curve for a given value of laxity after which P_b increases more rapidly. Given the mean laxity of the tasks, the network can limit the ratio of requests it accepts as ARs equal to the value of PAR at the knee of the curve to keep P_b at reasonable levels. When P_b is plotted against L, the curves are characterized by a knee that can act as a suitable operating point. The knee is reached for much smaller value of L compared to the one required to make P_b exactly equal to 0.

The paper also investigated the effect of task preemption on system performance. The results in Section 5.2 show that choice of preemption depends largely on the distribution of the task service times and the proportion of advance reservations. For low PAR values, when the variance between the service times is small, there is no significant advantage in task preemption. For PAR = 0.4, with the values of the parameters in the experiments , maximum increase in utilization for uniformly distributed task service times is 1.05% and it diminishes as overheads of preemption are considered. For hyper-exponentially distributed task service times with a co-efficient of variation of 2, the effect of task preemption is much more pronounced. In our experiments, for hyper-exponentially distributed service times, for PAR = 0.4, task preemption resulted in an increase in utilization of up to 3.15% and substantial decrease of up to 60.7% in R_{OD} and up to 43.38% in R_{AR}. For higher values of PAR, the improvement in utilization and response times is much more significant.

The results in Section 5.2 also show that the improvement in performance with preemption is sensitive to L. With the workload parameters in the experiments, maximum improvement in utilization is achieved at L = 70%. At higher L values, difference in utilization diminishes. This suggests that laxity can be exchanged for preemption to achieve high system utilization.

As proportion of advance reservations increases, response time of the on-demand requests increases substantially. The effect is more pronounced for higher L values. A very high response time for ODs will encourage all users to submit their tasks as ARs which would increase PAR. For lower L values, this would decrease their response time but at a cost of low utilization of the resource (see for example, Figure 3 and Figure 5). For higher L values, with the increase in PAR, there would not only be a slight decrease in U but also a tremendous increase in the response time of all requests (see for example, Figure 4 and Figure 6). In order to prevent these situations resulting from potential starvation of on-demand requests a resource, we presented a very simple policy of associating a virtual deadline with ODs. The results in Section 5.3 show that this policy can effectively decrease the response time of ODs without significantly decreasing U or substantially increasing R_{AR}. For higher L values, percentage decrease in utilization

diminishes while the percentage decrease in R_{OD} increases substantially. The policy is thus especially suitable for high L values.

The results in Section 5.3 also show that if tasks have equal deadlines there is no significant advantage in terms of reduction in response time by reserving the resources in advance. From this result, we can also deduce that changing the minimum time between the arrival of an advance reservation request and its start time does not change the overall performance.

7. REFERENCES

[1] M. Norman, P. Beckman, G. Bryan, J. Dubinski, D. Gannon, L. Hernquist, K. Keahey, J. Ostriker, J. Shalf, J. Welling, S. Yang, "Galaxies Collide on the I-WAY: An Example of Heterogeneous Wide-Area Collaborative Supercomputing," in *The International Journal of Supercomputer Applications*, Volume 10, No. 2, pp.131-140, 1996.

[2] J. Nieplocha, R. Harrison, "Shared Memory NUMA Programming on the I-WAY," in the *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, Syracuse, New York, USA, August 1996, pp. 432-441.

[3] Michael J. Litzkow, Miron Livny, Matt W. Mutka, "Condor: A Hunter of Idle Workstations," in the *Proceedings of the 8th IEEE International Conference of Distributed Computing Systems*, San Jose, CA, USA, June 1988, pp. 104-111.

[4] K. Marzullo, M. Ogg, A. Ricciardi, A. Amoroso, F. Calkins, E. Rothfus, "Nile: Wide-area Computing for High Energy Physics," in the *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[5] Glen H. Wheless, Cathy M. Lascara, Arnoldo Valle-Levinson, Donald P. Brutzman, William Sherman, William L. Hibbard, Brian E. Paul, "Virtual Chesapeake Bay: Interacting with a Coupled Physical/Biological Mode," in *IEEE Computer Graphics and Applications*, Volume 16, No. 4, pp. 42-43, July 1996.

[6] Maria Roussos, Andrew Johnson, Jason Leigh, Christina Valsilakis, Craig Barnes, Thomas Moher, "NICE: Combining Constructionism, Narrative and Collaboration in a Virtual Learning Environment," in *Computer Graphics*, Volume 31, No. 3, pp. 62-63, August 1997.

[7] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," in the *Proceedings of the 7th International Workshop on Quality of Service*, London, UK, May 1999.

[8] A. Sulistio, R. Buyya, "A Grid Simulation Infrastructure Supporting Advance Reservation," in *the Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems,* Cambridge, Boston, USA, November 2004.

[9] W. Smith, I. Foster, V. Taylor, "Scheduling with Advanced Reservations," in the *Proceedings of the IEEE/ACM 14th International Parallel and Distributed Processing Symposium,* Cancun, Mexico, May 2000.

[10] T. Lavian, S. Merrill, H. Cohen, D. Hoang, J. Mambretti, S. Figueira, D. Cutrell, S. Naiksatam, F. Travostino, "A Grid Network Service Architecture for Dynamic Optical Networks," submitted to the *Journal of Grid Computing, special issue on High Performance Networking.*

[11] G. McMahon, M. Florian, "On Scheduling with Ready Times and Due Dates To Minimize Maximum Lateness", in *Operations Research,* Volume 23, No. 3, pp. 475-482, May-June, 1975.

[12] J. Xu, D. Parnas, "Scheduling Processes With Release Times, Deadlines, Precedence And Exclusion Relations," in *IEEE Transactions on Software Engineering*, Volume 16, No. 3, pp. 360-369, 1990.

[13] The Globus Toolkit. http://www.globus.org.

[14] The Legion Project. http://legion.virginia.edu

[15] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A Resource Management Architecture for Metacomputing Systems," in the *Proceedings of the 4th IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, Orlando, FL, USA, March 1998, pp. 62-82.

[16] I. Foster, A. Roy, V. Sander, "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation," in the *Proceedings of the 8th International Workshop on Quality of Service*, Pittsburgh, PA, USA, June 2000, pp. 181-188.

[17] I. Foster, J. Vockler, M. Wilde, Y. Zhao, "The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration," in the *Proceedings of the First CIDR - Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, January 2003.

[18] T. Kosar and M. Livny, "Stork: Making Data Placement a First Class Citizen in the Grid," in the *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.

[19] The Maui Scheduling System. http://www.mhpcc.edu/maui.

[20] S. Figueira, N. Kaushik, S. Naiksatam, S. A. Chiappari, N. Bhatnagar, "Advance Reservation of Light-paths in Optical-Network Based Grids," in the *Proceedings of the 1st International Workshop on Networks for Grid*, San Jose, CA, USA, October 2004.

[21] D. A. Menasce, E. Casalicchio, "A Framework for Resource Allocation in Grid Computing", in the *Proceedings of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Volendam, The Netherlands, October 2004, pp. 259-267.

[22] H. Casanova, G. Obertelli, F. Berman, R. Wolski, "The Apples Parameters Sweep Template: User-Level Middleware for The Grid," in the *Proceedings of the ACM/IEEE Conference on Super Computing,* Washington D.C., USA, 2000.

[23] X. H. Sun, M. Wu, "Grid Harvest Service: A System for Long-Term Application-Level Task Scheduling," in the *Proceedings of International Parallel and Distributed Processing Symposium,* Nice, France, April 2003.

[24] L. Schrage, "Obtaining Optimal Solutions to Resource Constrained Network Scheduling Problems," *Unpublished Manuscript*, March 1971.

[25] S. Figueira, S. Naiksatam, H. Cohen, D. Cutrell, D. Gutierrez, D. B. Hoang, T. Lavian, J. Mambretti, S. Merrill, F. Travostino, "DWDM-RAM: Enabling Grid Services with Dynamic Optical Networks," in the *Proceedings. of the 4th IEEE International Symposium on Cluster Computing and the Grid*, Chicago, IL, USA, April 2004.

[26] J. Mambretti, J. Weinberger, J. Chen, E. Bacon, F. Yeh, D. Lillethun, B. Grossman, Y. Gu, M. Mazzuco, "The Photonic TeraStream: Enabling Next Generation Applications Through Intelligent Optical Networking at iGrid 2002," in *Journal of Future Computer Systems*, pp.897-908, August 2003.