A Framework to Achieve Guaranteed QoS for Applications and High System Performance in Multi-Institutional Grid Computing

Umar Farooq, Shikharesh Majumdar, Eric W. Parsons
Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada
Email: {ufarooq, majumdar, eparsons} @ sce.carleton.ca

Abstract

Providing QoS guarantees to the applications in a multi-institutional Grid is a challenging task. Although advance reservations (ARs) can provide QoS guarantees for the applications, they often seriously degrade system performance for resource owners. In this paper, we present a framework for AR based resource sharing that not only provides QoS guarantees for the applications but also ensures high utilization of the resources. This paper focuses on the application-to-resource mapping component of the framework as our results show that traditional mapping algorithms used with best effort *jobs do not perform well for ARs. The paper proposes a set of algorithms for mapping ARs and investigates their performance in detail. The paper then presents a novel algorithm that outperforms a number of other algorithms in almost every respect for a wide range of workload parameters. Rigorous experimentation proves the efficacy of our algorithm and brings important insights into the dynamics of the system.*

1. Introduction

Grid computing provides dynamic, secure and coordinated sharing of heterogeneous resources that may be distributed geographically as well as organizationally. One of the major challenges for Grid computing is to ensure "non-trivial qualities of service" to the users [1]. Provision of guaranteed level of QoS in multi-institutional geographically distributed resource sharing is complicated by two factors. First, the unpredictable delays over Wide Area Networks and second, the requirements of the Grid applications for heterogeneous resources that are being independently administered or controlled. Since resources belonging to different administrative domains do not share their schedules, if an 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, advance reservations (ARs) were introduced as a part of *Globus Architecture for Reservation and Allocation* (GARA) [2]. 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. Since their introduction, ARs have been studied in numerous contexts such as architecture for ensuring end-to-end QoS for network applications [3], job scheduler for clusters and supercomputers [4] and Grid based architecture for dynamic optical networks [5]. As by reserving resources in advance the user can be assured of an upper bound on the response time, ARs can be used for ensuring end-to-end quality of service. Just like other resources, Grid-enabled network resources can also be reserved in advance for a more predictable network delay [3, 5, 6].

Although advance reservations are attractive for resource consumers, they may cause severe performance degradation for the resource owners. Sulistio [7] and Smith [8] have studied the performance of ARs. Their results show that with only 20% of the jobs arriving as ARs, the utilization can go as low as 66% of the case where none of the jobs is an advance reservation while mean wait times of the *best effort jobs*, commonly known as on-demand requests (ODs) in Grid literature, can increase by 71%. Since there are multiple stakeholders in the Grid – resource owners and resource consumers – each with different objectives, we need a framework for resource sharing with ARs that can meet the objectives of both: provide QoS guarantees to resource consumers and ensure high utilization of resources for resource owners.

In this paper, we present a framework for resource sharing with ARs that meets the objectives of both the resource consumers and the resource owners. The framework comprises several entities including resource brokers and schedulers. The goal of the resource broker is to efficiently map jobs to different

resources and the goal of the scheduler is to order jobs mapped to a particular resource to achieve specified time requirements. In earlier papers [6, 9, 10], we have shown how *laxity* in the reservation window can help improve resource utilizations of AR based scenarios by leaving the final scheduling decision with the scheduler. Laxity of an AR 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 earliest start time. Earlier, we have presented scalable algorithms for scheduling ARs mapped to a given resource [6, 9, 10]. In this paper, we focus on the mapping component as our results show that the algorithm for mapping ARs to resources, herein called matchmaking, can significantly affect performance. The contributions of the paper are listed.

- The paper presents a framework for resource sharing with ARs that can achieve QoS guarantees for the applications and high performance for the resources in multi-institutional Grid computing.
- To the best of our knowledge, this is one of the first papers that tackle the issue of mapping jobs to resources for AR based scenarios. With the help of simulation results, we show that traditional mapping algorithms used in Grid and distributed computing in general do not give desired performance with ARs. We present different algorithms for matchmaking and investigate their performance in detail.
- We present a novel algorithm, Minimum Laxity Impact (MLI), for matchmaking with ARs that outperforms a number of other algorithms in almost every respect for a wide range of workload parameters.
- Through simulation, we study the effect of workload and system parameters on matchmaking algorithms. The results provide important insights into system behavior and performance.
- The results of this paper are also applicable to Community Scheduler Framework (CSF) [11]. The meta-scheduler within CSF can use the matchmaking algorithms presented in this paper to co-schedule resources across different domains.

The rest of the paper is organized as follows. Section 2 briefly reviews the related work. In Section 3, we present a framework for resource management with ARs and present algorithms for matchmaking. We discuss in detail our experimental setup in Section 4 while in Section 5 we present the experimental results. We conclude the paper in Section 6.

2. Background and related work

An advance reservation reserves a Grid resource for the execution of a job at a specified time in future. A *job* is a Grid application consisting of a single or multiple *tasks*. A job completes when all its tasks are completed. A Grid resource may be a single CPU or a multiple CPU based computing device or a storage system. If a Grid resource is used in such a way that each job requests the exclusive use of the resource during its execution, we refer to the sharing model as the *non-shared resource model*. On the other hand, if a resource can be shared among multiple jobs running concurrently on the resource we refer to the sharing model as the *shared resource model*. An example of a shared resource model is one in which a cluster consisting of multiple nodes is shared among multiple jobs by running tasks of different jobs on different nodes. Only one task can run on a node at a time.

ARs with laxity are characterized by a *start time of the reservation* that specifies the time at which the job would be available for execution and a *deadline* by which the job must be completed by the resource. Thus, given a set of jobs assigned to a particular resource the goal of the scheduler is to schedule jobs in such a way that each job starts executing at or after its start time and finishes before its deadline. It has been shown that scheduling jobs with given start times, execution times and deadlines is an NP-Hard problem even if we consider a non-shared resource model [12]. In previous papers, we have presented scalable algorithms for scheduling ARs for both non-shared [6, 9] and shared [10] resource models.

Globus Architecture for Reservation and Allocation (GARA) [2], which introduced ARs, presents the concepts of co-reservation and co-allocation agents that an application can use to dynamically assemble resources to meet its QoS constraints. The co-reservation agents interact with the Information Service to discover resources that can satisfy application constraints. GARA however, does not specify how co-reservation agents select among the resources that can satisfy the application constraints or how reservations are scheduled by the underlying resource managers to achieve performance while meeting application's QoS constraints. Our framework specifically deals with the above-mentioned issues and in this context extends the functionality of GARA.

Among other notable application-to-resource mapping mechanisms in Grid are the Condor-G Class-Ad matchmaking mechanism [13] and Sun Grid Engine load averaging mechanism [14]. In Condor-G, an application can specify its request for resources in the form of a small advertisement while a resource can also advertise its capabilities. Based on these advertisements, the goal of the broker is to map application to resources that can satisfy the application constraints. Although Condor-G matchmaking takes into account user preferences for resources, it does not specify how broker should make dynamic mapping

decisions based on the system state. As the results presented in this paper show, such dynamic matchmaking can significantly affect the overall system performance especially for AR based scenarios. Sun Grid Engine on the other hand, has two primary policies for application-to-resource mapping. In one of the policies, the resources are sequenced and each time the list is iterated through to select the next node. In the second policy, the node with the minimum load is selected to run the next job. The results presented in Section 5 however show that when the proportion of advance reservations increases none of these policies results in high performance.

Community Scheduler Framework (CSF) [11] is an open source framework for implementing a Grid meta-scheduler that can dispatch jobs to underlying resource managers. The current implementation of CSF employs scheduling policies similar to those employed in Sun Grid Engine. It thus needs an efficient matchmaking algorithm for AR based scenarios.

3. A framework for resource management with advance reservations

We present a framework for AR based resource sharing that ensures high performance for the resource owners while meeting the QoS needs of the resource consumers. A high-level simplified view of the framework is presented in Figure 1. Due to space limitations, detailed architecture is not presented.

Figure 1 shows that the framework consists of two major components: MMC that lies at the collective layer of the Grid layered architecture [1] and RLC that lies at resource and fabric layers. Applications submit their resource requirements along with their QoS constraints to MMC. MMC finds a suitable match for the application and allocates the resource using RLC. For the applications that need QoS guarantees and/or those that need simultaneous co-allocations of resources, ARs are used.

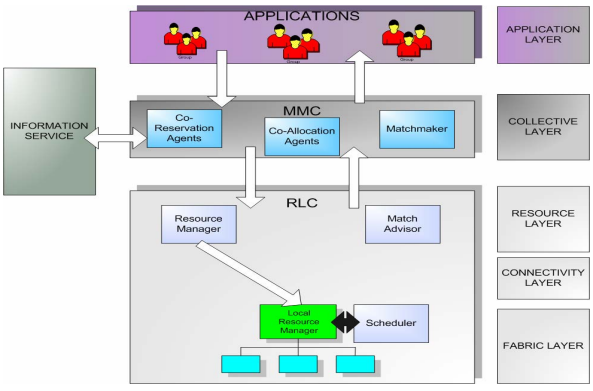

Figure 1. High-level view of the framework

The function of the RLC is to do admission control for each resource and ensure that each accepted AR meets its deadline. If the resource cannot meet the QoS needs of an AR, it rejects the new request. To prevent performance degradations due to ARs, laxity in the reservation window is required. Thus, while RLC provides response time guarantees to the application, it leaves final scheduling decisions with the local scheduler. The simplified view of RLC in Figure 2 shows three major components. The *Resource Manager* component provides the functionality of a remote resource manager and negotiates access to the resource through the local resource managers. The *Scheduler* component schedules jobs in such a way that QoS constraints of ARs are met while high utilizations of resources are maintained. Depending on the model of the resource used, the Scheduler employs either our algorithm for a non-shared resource model [6, 9] or our algorithm for a shared resource model [10]. The function of *Match Advisor* is to asses the degree of fit of a particular job given the QoS requirements of the job and current state (schedule) of the resource and report it to MMC.

MMC also consists of three components. Just like in GARA, the *Co-reservation Agents* discover available resources through the information service. They then use the services of the *Matchmaker* component to choose the resource to allocate the job. Matchmaker relies on the reports from the Match Advisors of each resource and employs our MLI algorithm for resource selection. Once the resource(s) is (are) selected, the reservation handle is provided to the application. At the execution time of the application, the application uses *Co-allocation Agent* to submit the tasks to the selected resource(s).

The framework can be used for efficient meta-scheduling within CSF. Intelligent matchmaking before dispatching ARs to be scheduled on resources can result in significant improvement in performance.

3.1. Algorithms for matchmaking

This section presents several algorithms that we have investigated for matchmaking. These algorithms assume that the resources that do not meet the job's functional requirements have already been eliminated during the resource discovery process. If the number of resources that meet a job's functional requirements is too high, only a subset of resources can be passed to the Matchmaker. We call the resources that are passed to the Matchmaker to do the selection from as the *selection window*. We compare the performance of these algorithms in Section 5.

3.1.1. Random (RAN). This algorithm randomly picks up a resource from the selection window. If the

resource meets the QoS needs of the job, the resource is selected. Otherwise, the process is repeated until either a match is found or all options are exhausted.

3.1.2. First fit (FF). With this algorithm, all resources in the selection window are sequenced in a particular order. Each time a new job is submitted, the first resource in sequence is tried. If the resource meets the QoS constraints of the job, the resource is selected. Otherwise, the next resource in sequence is tried. The process is repeated until either a match is found or all options are exhausted. This algorithm has a limitation as it assumes that either all jobs arrive at the same broker that has sequenced the resources or a particular ordering of the resources has been agreed upon by all brokers in the system.

3.1.3. Next fit (NF). In this algorithm, all resources in the selection window are sequenced in a particular order. When the jobs are submitted, the list of the resources is iterated through in a sequence, selecting the next resource in the list. If that resources does not meet QoS constraints of the job the next resource in sequence is tried. The process is repeated until either a match is found or all options are exhausted. In its simplest form, with no QoS constraints associated with the job, the algorithm is used in Sun Grid Engine [14]. This algorithm has the same limitations as FF.

3.1.4. Initial minimum completion time (IMCT). Whenever a new request arrives, this algorithm queries all the resources in the selection window and finds a subset of resources that can meet the QoS needs of the job. It then chooses from this subset the resource that reports minimum completion time for the job. Note that the Scheduler in RLC can re-schedule jobs to improve resource utilization as long as it meets the QoS needs of all accepted jobs. Hence, the time of completion reported to the Matchmaker at this time may not be the actual completion time of the job.

3.1.5. Resource with minimum utilization (MinU). Whenever a new request arrives, this algorithm queries all the resources in the selection window and selects the one with minimum utilization among the resources that meet the QoS constraints of the job. This algorithm thus tries to balance the load on resources. It is used by Sun Grid engine [14].

3.1.6. Resource with maximum utilization (MaxU). Whenever a new request arrives, this algorithm queries all the resources in the selection window and selects the one with maximum utilization among the resources that meet the QoS constraints of the job. The rationale behind this algorithm is that it keeps enough spare capacity on some of the resources to accommodate jobs with large demands.

3.1.7. Minimum laxity impact (MLI). In this algorithm, whenever a new request arrives, all resources in the selection window are queried and the algorithm selects one from those that meet the QoS constraints of the job. This resource selection is done is such a way that the accommodation of this job produces the minimum impact on laxity of the jobs already in the schedule of the resource. If in order to accommodate the new job the resource has to re-schedule a number of jobs pushing them towards their deadlines, we say the impact on laxity was high. On the other hand, if the resource can accommodate the job without significant re-scheduling, we call the impact on laxity was low. Thus, if we always select the resource that has minimum impact on laxity to accommodate a job we would have a higher probability of finding out a feasible schedule for later arrivals by utilizing the spare laxity. Quantitatively, we measure the impact on laxity by subtracting from cumulative *remaining laxity* of the jobs in schedule before the new task was accommodated the cumulative remaining laxity of the jobs after the new job is accommodated.

If t_i represents the earliest start time of job i, e_i its runtime on the resource and d_i its deadline then the laxity LX_i of the job on the resource can be given as:

$$LX_i = d_i - e_i - t_i \qquad (3.1)$$

Since it is not always possible to schedule a job at its earliest start time due to the other jobs in schedule, the remaining laxity RL_i of a job i measures the "laxity" of the job at its scheduled-time s_i at which it is currently scheduled to start. Thus,

$$RL_i = d_i - e_i - s_i \qquad (3.2)$$

Let a_j represent the arrival time of the job j, ϵ represents the set of jobs in the resource schedule that are scheduled to start after a_j and $s_{i(j-1)}$ and $s_{i(j)}$ respectively represent the scheduled-time of a job i in ϵ before and after the job j is accommodated in the system. Then the impact on laxity, ζ, of the new job j is calculated as:

$$\zeta = \sum_{(\text{For all jobs i in } \epsilon)} [(d_i - e_i - s_{i(j-1)}) - (d_i - e_i - s_{i(j)})] \qquad (3.3)$$

$$= \sum_{(\text{For all jobs i in } \epsilon)} RL_{i(j-1)} - \sum_{(\text{For all jobs i in } \epsilon)} RL_{i(j)} \qquad (3.4)$$

If all resources are identical, the algorithm selects the resource that reports the smallest value of ζ. Otherwise, ζ is scaled with respect to the relative speeds of the resources and the algorithm selects the resource that results in smallest value of scaled ζ. Due to space limitations, discussion on scaling ζ is not presented in this paper. If more than one resource has same value of ζ, ties are broken by selecting the one that reports the minimum completion time.

4. Experimental setup

For the simulation experiments, we model 64-node IBM SP2 supercomputers as Grid resources. Shared resource model is simulated in which different applications can run on SP2 concurrently by running tasks of different jobs on different nodes. The Grid applications are modeled as parallel non-preemptable rigid jobs requesting any where from 1 to 64 nodes depending on their number of tasks. The applications submit their requirements along with their QoS constraints to MMC. The Matchmaker in MMC uses algorithms presented in Section 3.1 to find a suitable match. Since different tasks of the parallel job may need to communicate during execution, once a suitable cluster has been selected all tasks of the job are gang scheduled on the cluster. If the job needs to be re-scheduled to improve cluster utilization all tasks of the job are always re-scheduled as a gang.

For efficient gang scheduling with QoS constraints within a cluster, RLC employs our algorithm for AR scheduling with shared resource model [10]. This algorithm is capable of handling both non-preemptable and preemptable parallel rigid jobs. It can work for computing, network and storage resources and is capable of scheduling tasks of a job as a gang. The algorithm employs the earliest-deadline-first heuristic. If we consider the modeled resource to be a computing cluster, the algorithm selects nodes of the cluster to run the tasks of a particular job in such a way that fragmentation in the resource schedule is minimized. It is also capable of dealing with inaccuracies in user's estimations of the runtimes of the jobs. The algorithm has been described and analyzed in detail in [10].

4.1. Performance metrics

The paper uses the following performance metrics.

4.1.1. Percentage of work rejected (W_R). W_R measures the percentage of the work rejected by the system because it cannot be accommodated without violating QoS constraints of some of the jobs. Total work of a job is defined as the sum of the runtimes of all of its tasks. Since in our model the tasks are always gang-scheduled, runtimes of all the tasks can be considered equal. Hence, total work of a job can also be computed by multiplying the size of the job (number of tasks) by its runtime. W_R is the ratio of the sum of the total work of all the jobs rejected and the sum of the total work of all the jobs submitted.

4.1.2. Probability of blocking (P_b). P_b is the probability that a system would reject a new job because it cannot be accommodated without violating QoS constraints of some of the jobs. It can be calculated as:

P_b = Total Number of Requests Rejected / Total Number of Requests (4.1)

4.1.3. Utilization (U). U of a resource is the fraction of the total time the resource is busy executing jobs. It is calculated as follows:

U = Sum of the Busy Units of Time of All the Nodes in the Cluster / (Total Duration of Simulation * Total Number of Nodes in the Cluster) (4.2)

4.1.4. Fairness (Q). Q measures the ability of the matchmaking algorithm to treat small and large jobs equally. It is defined as the average total work of all jobs rejected divided by the average total work of all jobs submitted. A value of 1 for Q means that the matchmaking algorithm treats small and large jobs equally, a value greater than 1 means more large jobs are rejected than small jobs while a values less than 1 means comparatively more small jobs are rejected.

4.1.5. 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.

4.1.6. Mean response time of advance reservation requests (R_{AR}). R_{AR} is the mean time between the earliest start time of an AR request and the time of completion of the job associated with that request.

4.2. Workload model

We generated jobs using a synthetic workload model for rigid jobs proposed in [15]. The workload model is based on the traces collected from San Diego Supercomputer Center's Intel Paragon Machine, a CM-5 machine at Las Alamos National Lab and IBM SP2 machine at the Swedish Royal Institute of Technology. Since the Grid resources modeled in this paper are IBM SP2 machines, we additionally analyzed traces collected from SP2 at Cornell Theory Center and SP2 at San Diego Supercomputer Center to calculate the parameters for the workload model. The workload model models the runtime of the jobs using hyper-gamma distribution with a mean of approximately 2200 seconds and the size of the jobs using two-phase uniform distribution with a mean of approximately 8.4 nodes. The correlation between the size of the job and its runtime is also modeled and described in detail in [15]. Since it is not always possible to accurately predict the runtimes of the jobs, in Section 5.3 we discuss how our framework deals with inaccuracies in user-estimated runtimes.

We used an open model in which a stream of jobs arrives on the system. As in [15], the arrival process is modeled with a Gamma distribution. We multiplied the arrival time of the jobs with a constant number to generate different load conditions. The details of the parameters are discussed in [10].

4.3. Workload parameters

Following are the workload parameters of interest.

4.3.1. Proportion of advance reservations (PAR). PAR is the proportion of AR requests in the total number of requests. We study the performance of the matchmaking algorithms at different values of PAR. The time between the arrival of an AR and its start time is modeled as a uniform distribution. ARs can request to reserve the resource for any time between the current system time and the next 12 hours.

4.3.2. Mean percentage laxity (L). We define *percentage laxity* of an AR as the ratio of its laxity and its runtime. A uniform distribution for the laxity of ARs is used with the lowest value of the distribution fixed at 0. Mean percentage laxity L is varied between 0% and 1000%.

Theoretically, ODs have no deadlines. However, to prevent starvation of ODs, we associate a large deadline with ODs equivalent to 2 days.

4.3.3. Total number of clusters (N). In order to study the effect of the size of the selection window, we vary the total number of clusters in the system. However, only the results obtained with N = 4 are presented in this paper.

4.4. Accuracy of results

Tests were run long enough and repeated multiple times, to obtain reasonably small confidence intervals for the performance metrics. We obtained confidence intervals of ±5% for W_R and Q, ±3 % for U, R_{AR} and R_{OD}, and ±10.0% for P_b at a confidence level of 95%. For all graphs to be presented in Section 5, the mean of the performance metric is plotted.

5. Experimental results

5.1. Impact of laxity

This section presents the impact of laxity on the performance of the system. The results shown in Figure 2 were obtained with PAR = 1 and N = 4.

As expected, Figure 2(a) shows that with the increase in laxity, percentage of work rejected W_R decreases significantly. The figure also shows that the matchmaking algorithm can significantly affect system performance justifying the use of an intelligent Matchmaker at MMC. At L = 1000%, the spread in W_R is over 25% of W_R of RAN. The comparison of the algorithms shows that when L is increased beyond a certain point, MLI performs better than all other algorithms. At L = 1000%, it results in W_R which is only 89.2% of the next best algorithm (IMCT). Although not shown in the figure due to space limitations, the results also show that MLI increases U of each cluster by 2% over that of the next best algorithm. The performance of MLI can be attributed to the economical use of jobs laxity.

Surprisingly, MaxU and FF perform better than MinU, resulting in lower W_R and higher U. This is because these algorithms always concentrate the load on a subset of clusters and hence leave enough capacity on some of the other clusters to accommodate large jobs when they arrive. This shows that the techniques that are known to perform well in ordinary application-to-resource mapping do not perform well for AR based scenarios. Although not shown in figure to avoid cluttering, the performance of NF is close to that of RAN.

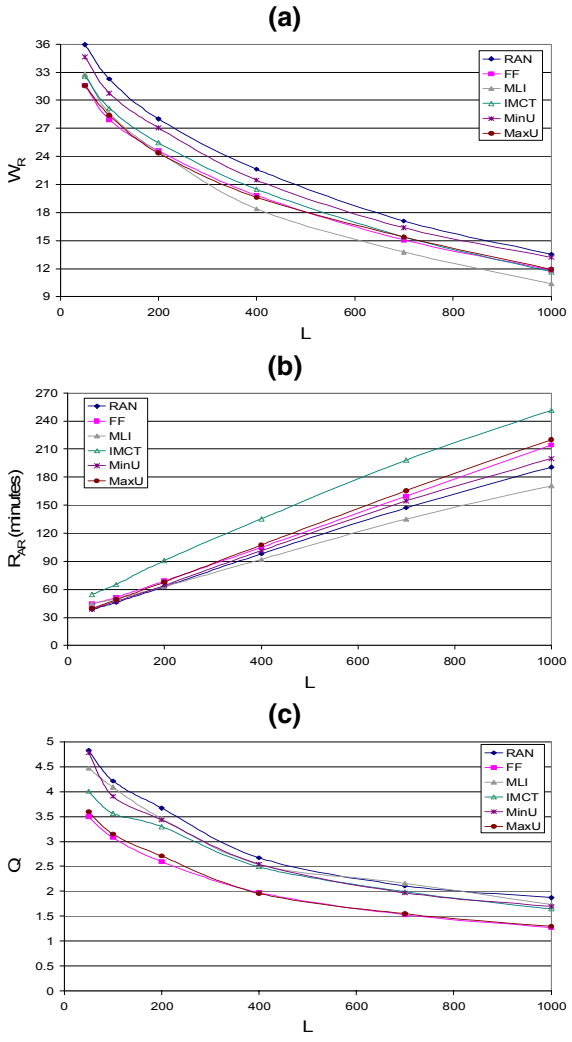

Figure 2. Impact of laxity on performance

Figure 2(b) shows that MLI results in the lowest R_{AR} values. IMCT results in a significantly higher R_{AR} than all other algorithms. This is because that often minimum completion time for the new job is reported by that cluster in which, in order to accommodate the new job in the schedule, the shared resource scheduler

ends up pushing jobs already in the schedule towards their deadlines. This tends to increase the response time of the previously scheduled jobs. As this greedy algorithm often selects such a cluster the overall response time of the jobs in the system increases.

Figure 2(c) shows that when L is small, Q is many times higher than 1 showing that many of the jobs with large total works are rejected as they cannot be accommodated in the system. As L increases, there is more flexibility in scheduling and Q approaches 1. As FF and MaxU keep enough spare capacity on some of the clusters to accommodate large jobs, their Q is lower and they are fairer to large jobs than the other algorithms. Since with MLI the load is more balanced among the clusters, it results in values of Q higher than those achieved by FF and MaxU.

P_b curves (not presented in this paper due to space limitations) show that P_b does not change significantly with the increase in L. It can be shown that $P_b = W_R / Q$ and hence as both W_R and Q decreases at almost the same rate with the increase in L, P_b tends to remain constant. MLI results in the lowest P_b values.

5.2. Effect of mixed workload

This section presents the results obtained by using a mix of advance reservation and on-demand (OD) requests. The results shown in Figure 3 were obtained with PAR = 0.5 and N = 4.

Figure 3 shows that trends of the curves for W_R, Q and R_{AR} are the same as in Figure 2 with PAR = 1. However, with 50% ODs in Figure 3(a) W_R is significantly lower (by up to 66.74%). This is because ODs have less QoS constraints and hence provide more flexibility in scheduling. The graphs show that when ODs are introduced along with ARs the choice of the matchmaking algorithm becomes even more important. For L = 1000%, the spread of W_R in Figure 3(a) is 46.5% of W_R obtained with the worst algorithm which in this case is MaxU. The figure shows that FF and MaxU that perform well for PAR = 1, perform very poorly when there are a significant number of ODs in the system. However, MLI still performs the best among all the algorithms. IMCT gives comparable performance as far as W_R, U and R_{OD} are concerned; however, IMCT gives worst R_{AR} for the reasons described in Section 5.1.

5.3. Impact of inaccuracies in user-estimated runtimes

It is not always possible to accurately predict the runtimes of the jobs but our framework is capable of dealing with inaccuracies in user-estimated runtimes. The inaccuracies in runtimes are dealt with at the Scheduler level in the RLC component of our framework. In our framework, the schedulability analysis at the time of admission is always done using the user-estimated runtimes. However, if a job finishes earlier than its estimated runtime the scheduling process is triggered to utilize the extra time left in the schedule by the finishing job. If a job takes longer to finish than its estimated runtime, the schedulability analysis is triggered to see if a small quantum of time can be allocated to the job without violating the QoS constraints of the other jobs in schedule. If the job still does not finish in the extra time, the job can be allocated more than one quanta if doing so does not result in violation of the QoS constraints of other jobs. Otherwise, the job is aborted. The detailed discussion on dealing with inaccuracies and the corresponding performance results are currently under investigation.

6. Conclusions

Providing QoS guarantees to the applications in a multi-institutional Grid is a challenging task because there are multiple stakeholders – resource owners and resource consumers – each with different objectives. Although advance reservations can provide QoS guarantees for the applications, they may result in poor system performance for resource owners if resource management is not done carefully. This paper presents a framework for resource management with ARs that not only provides QoS guarantees for the applications but also ensures high utilization of the resources. The paper focused on the Matchmaker component of the framework, presented several algorithms for matchmaking and investigated their performance in detail.

The results show that the choice of the matchmaking algorithm can significantly affect system performance. For the parameters used in the experiments, using a suitable matchmaking algorithm can reduce W_R by as much as 46.5%. This justifies the use of intelligence in matching. The results also show that traditional mapping algorithms used in Grid technologies such as MinU and NF do not perform well when ARs are introduced in the system. The MLI algorithm proposed by this paper exploits the current system state and performs better than every other algorithm in terms of lower W_R, R_{AR} and R_{OD} and higher U for a wide range of workload parameters. MLI gives the best performance even when a significant number of ODs are introduced in the system. The high performance of MLI can be attributed to the intelligent use of laxity of jobs.

Among other algorithms that give good performance are FF and MaxU when PAR is high and IMCT when PAR is low. MaxU and FF are most fair to small and large jobs and for large L result in values of Q close to 1.

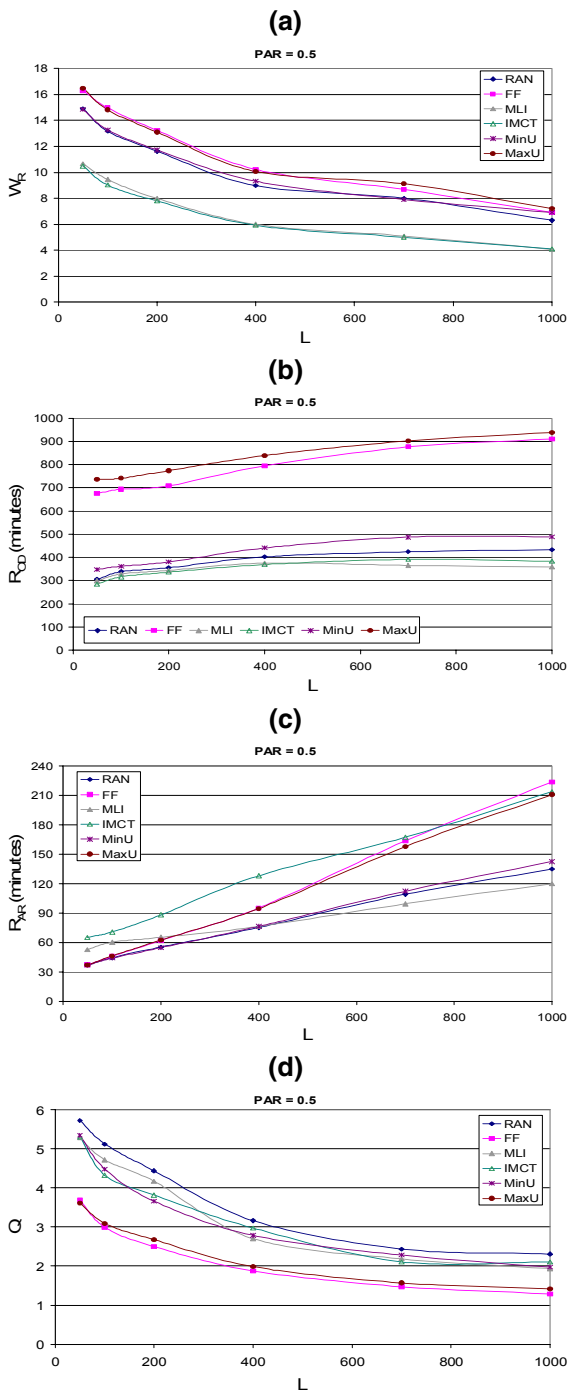

Figure 3. Effect of mixed workload on performance

6. References

[1] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," in *Int'l Journal of Supercomputer Applications*, 15(3), 2001.

[2] 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 *Proc. of the 7th Int'l Workshop on QoS*, May 1999.

[3] I. Foster, A. Roy, V. Sander, "A QoS Architecture that Combines Resource Reservation and Application Adaptation," in the *Proc. of the 8th Int'l Workshop on QoS*, pp. 181-188, June 2000.

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

[5] 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.*

[6] U. Farooq, S. Majumdar, E. Parsons, "Dynamic Scheduling of Lightpaths in Lambda Grids," in the *Proc. of the 2nd Int'l Workshop on Networks for Grid Applications*, pp. 540-549, Oct. 2005.

[7] A. Sulistio, R. Buyya, "A Grid Simulation Infrastructure Supporting Advance Reservation," in the *Proc. of the 16th Int'l Conf. on Parallel and Distributed Computing and Systems,* Nov. 2004.

[8] W. Smith, I. Foster, V. Taylor, "On Sched. with Advanced Reservations," in the *Proc. of the 14th Int'l Parallel and Dist. Proc. Symp.,* May 2000.

[9] U. Farooq, S. Majumdar, E. Parsons, "Efficiently Scheduling Advance Reservations in Grids," *Technical Report SCE-05-14*, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, August 2005.

[10] U. Farooq, S. Majumdar, E. Parsons, "Scheduling Advance Reservations on Shared Grid Resources", *Technical Report*, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, July 2006.

[11] Platform Computing Inc., "Open Source Meta-Scheduling for Virtual Organizations with the Community Scheduler Framework (CSF)," *Technical Whitepaper*. http://www.platform.com.

[12] G. McMahon, M. Florian, "On Scheduling with Ready Times and Due Dates To Minimize Maximum Lateness," in *Operations Research,* 23(3), pp. 475-482, May-June, 1975.

[13] R. Raman, M. Livny, M. Solomon, "Matchmaking: Dist. Resource Management for High Throughput Computing," in the *Proc. of the 7th IEEE Int'l Symposium on High Performance Distributed Computing,* July 1998.

[14] The Sun Grid Engine, http://www.sun.com/software/gridware/index.xml

[15] U. Lublin, D. G. Feitelson, "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs," in *Journal of Parallel & Distributed Computing,* 63(11), pp. 1105-1122, Nov. 2003.