A Framework for Quality of Service Aware Resource Management in Multi-Institutional Grids

by
Umar Farooq, M.A.Sc. (EE), B.Sc. (EE)

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
September 2007.

Your file Votre référence
ISBN: 978-0-494-33489-8
Our file Notre référence
ISBN: 978-0-494-33489-8

Abstract

A Grid is a collection of heterogeneous resources spread over multiple administrative and geographical domains. Many Grid applications require quality of service (QoS) guarantees in terms of guaranteed response time and guaranteed allocation of heterogeneous resources in multiple domains. Since QoS objectives of resource consumers in Grids are often in conflict with efficiency goals of resource providers and since Grid resources are shared by multiple applications that belong to different administrative domains, it is challenging to meet the QoS objectives of applications. Grid systems must also be robust enough to accommodate uncertainties such as those in user-estimated runtimes while meeting QoS and efficiency goals.

This thesis presents an advance reservations based middleware framework for Grids that achieves user satisfaction by providing QoS guarantees for Grid applications, cost effectiveness by efficiently utilizing resources and robustness by intelligently handling uncertain runtimes of applications. The framework provides components for each of the fabric, resource and collective layer of the Grid architecture. Within the framework, the thesis presents new application-to-resource mapping and scheduling strategies as experimental results show that traditional strategies do not give desired performance in multi-institutional environments. The thesis presents two scalable algorithms: Scaling through Subset Scheduling and Grid Scheduling with Deadlines for the NP-Complete problems of QoS constrained scheduling on non-shared and shared Grid resources, respectively. The thesis investigates different algorithms for mapping QoS constrained Grid requests to resources and presents a novel algorithm for mapping, Minimum Laxity Impact, which outperforms all other algorithms investigated in almost

every respect for a wide range of workload parameters. Co-Allocation of heterogeneous resources in multiple domains is challenging but is required to meet the complex demand patterns of Grid applications. The thesis presents a novel three phase co-allocation approach, along with effective co-allocation algorithms, that outperform traditional techniques. The thesis also studies in detail the impact of error in user-estimated runtimes on system performance and presents strategies and policies to avoid substandard performance resulting from such inaccuracies.

Finally, the thesis argues that some of the resource management strategies and conclusions presented have a broader appeal and are applicable for other multi-institutional resource sharing infrastructures.

Acknowledgements

First and foremost, I would like to express my sincere thanks and gratitude to my advisors Prof. Shikharesh Majumdar and Dr. Eric Parsons for their continuous support, encouragement and help throughout the thesis. Your guidance and time during each phase of the thesis was invaluable. I deeply appreciate your consistent enthusiasm and patience.

I am thankful to Nortel, Natural Sciences and Engineering Research Council of Canada, Government of Ontario and Carleton University for providing financial support for this research.

I am grateful to my local and global peers whose comments helped shape this thesis. I am thankful to Omair Muhi Kanwar who implemented a part of my design for the integration of some of the components of my framework with Globus Toolkit.

I am forever indebted to my father and my mother for their understanding, endless patience and encouragement. I am thankful to my sister and brother for their support. I owe special thanks to my mother and my grandfather who inspired me into research and critical thinking. Your confidence and faith in me have motivated me throughout my life and helped me achieve this first milestone of your aspirations about me.

I am thankful to my wife, Haida, who has always been my pillar, my joy and my source of strength. Without your unconditional love and unwavering support during many long days, this thesis would not have been possible. Thanks for always being there for me but never asking why it was taking so long. Thanks to my little bundle of joy – my daughter, Hannah. You make everything worthwhile.

Table of Contents

List of Tables

List of Figures

Glossary of Terms

APB	Abortion Policy Block
API	Application Programmers' Interface
AR	Advance Reservation Request
BF	Best Fit
CAA	Co-Allocation Agents
CB-DT-LE	Cost Based with Dynamic Timing – Low End Algorithm
CB-HE	Cost Based – High End Algorithm
CB-LE	Cost Based – Low End Algorithm
CIM	Common Information Model
CMP	Compression Policy
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRA	Co-Reservation Agents
CSF	Community Scheduler Framework
CTC	Cornell Theory Center
DCOM	Distributed Components Object Model
DP	Deadlines Policy
DRAC	Dynamic Resource Allocation Controller
DUROC	Dynamically Updatable Resource Online Co-Allocator
EAF	Earliest Arrival Time First
EDF	Earliest Deadline First
EP	End Point
ESF	Earliest Start Time First
FF	First Fit
FP	Feasibility Policy
ftp	File Transfer Protocol
GARA	Globus Architecture for Reservation and Allocation
GGF	Global Grid Forum
GIIS	Globus Index Information Service

GIS	Grid Information Service
GMA	Grid Monitoring Architecture
GRAM	Globus Resource Allocation Manager
GridFTP	Grid File Transfer Protocol
GRIS	Globus Resource Information Service
GSD	Grid Scheduling with Deadlines Algorithm
IMCT	Initial Minimum Completion Time Algorithm
IT	Information Technology
KTH	Swedish Royal Institute of Technology
LDAP	Lightweight Directory Access Protocol
LLF	Least Laxity First
LSF	Load Sharing Facility
MA	Match Advisor
MaxU	Maximum Utilization Algorithm
MDS	Globus Discovery and Monitoring System
MFP	Multiples Feedback Loops Policy
MI	Millions Instructions
MinU	Minimum Utilization Algorithm
MLI	Minimum Laxity Impact Algorithm
MLI-SA	Minimum Laxity Impact with Scaling A Algorithm
MLI-SB	Minimum Laxity Impact with Scaling B Algorithm
MLI-SDP	Minimum Laxity Impact with Self Dynamic Partitioning Algorithm
MM	Matchmaker
MMC	Matchmaker and Multi-Resource Coordinator
MRC	Multi-Resource Coordinator
NF	Next Fit
NP	Non-Preemptable Job
NP-Complete	Non Polynomial Complete
NSF	National Science Foundation
OD	On-Demand Request
ODIN	Optical Dynamic Intelligent Network Service

OGSA	Open Grid Service Architecture
OGSI	Open Grid Services Infrastructure
P	Preemptable Job
PAR	Proportion of Advance Reservations
PBS	Portable Batch System
PCR	Proportion of Co-Allocation Requests
pdf	Probability Density Function
PE	Pre-Scheduling Engine
QoS	Quality of Service
RAN	Random Algorithm
RL	Resource Liaison
RLC	Resource Liaison and Controller
SCH	Scheduler Component of Resource Liaison and Controller
scp	Secure Copy Protocol
SDSC	San Diego Supercomputer Center
SEM	Schedule Exceptions Manager
SND	Standard Normal Distribution
SP	Start Point
SPM	Starvation Prevention Manager
SSS	Scaling through Subset Scheduling Algorithm
WF	Worst Fit
WS	Web Services
WSDL	Web Services Description Language
WSRF	Web Services Resource Framework
XML	Extensible Markup Language

List of Symbols

ă	Translation Co-Efficient
a_i	Arrival Time of Job j_i
B	Band of Error
b_z	Resource z
C_v	Co-Efficient of Variation
d_i	Deadline of Job j_i
E_E	Mean Estimated Execution Time
$e_{E,ib}$	Estimated Execution Time of Job j_i on resource b
f	Transformation Factor in Compression Policy
f_Z	Transformation Factor for Class Z in Multiple Feedback Loops Policy
H	Heterogeneity of the System
h	One of the Four Parameters of a Two-Stage Uniform Distribution
$\hat{H}$	Total Number of Solution-Nodes Allowed
h_1	One of the Two Parameters of a Uniform Distribution
j_i	A Job with Index i
L	Mean Percentage Laxity
l	One of the Four Parameters of a Two-Stage Uniform Distribution
l_1	One of the Two Parameters of a Uniform Distribution
LX_i	Laxity of Job j_i
m	One of the Four Parameters of a Two-Stage Uniform Distribution
n	Number of Jobs
N	Total Number of Resources in the Candidate set
$\check{N}$	Total Number of Solution-Nodes Produced to Schedule Given Jobs
n_{ib}	Nodes of Resource b Required by Job j_i to Execute
$\check{N}_{max}$	Maximum Number of Solution Nodes Open in Memory at Any Point
ŋ	Percentage Error in Estimated Runtime
$ŋ_{mean}$	Mean Percentage Error in Estimated Runtimes
O	Mean Time Spent in Job Preemption and Its Resumption

o	Time Spent in a Job's Preemption and Its Resumption
p_1	Probability that Job is Serial
p_2	Probability that Job Size is a Power of 2
P_A	Probability of Aborting
P_b	Probability of Blocking
p_g	One of the Five Parameters of a Two Phase Hyper-Gamma Distribution
p_h	One of the Three Parameters of a Two Phase Hyper-Exponential Distribution
p_u	One of the Four Parameters of a Two-Stage Uniform Distribution
Q_R	Fairness
R	Mean Response Time of Jobs Accepted
R_{AR}	Mean Response Time of Advance Reservation Requests
r_j	Speed factor of a Resource b_j
RL_i	Remaining Laxity of Job j_i
$RL_{i(j)}$	Remaining Laxity of Job j_i After the Arrival of Job j_j
R_{OD}	Mean Response Time of On-Demand Requests
S	A Subset of Jobs Selected by the Scaling Through Subset Scheduling Algorithm for Scheduling
sch_i	Current Scheduled-Time of Job j_i
$sch_{i(j)}$	Scheduled-Time of Job j_i After the Arrival of Job j_j
scp	Secure Copy Protocol
s_{ib}	Size of Job j_i on resource b
$þ$	Threshold Size of Set of Jobs S
t_i	Earliest Start Time of Job j_i
T_S	Time between the Arrival of an Advance Reservation Requests and Its Earliest Start Time
U	Utilization
UU	Useful Utilization
W	Mean Wait Time of Jobs Accepted
W_A	Percentage of Work Aborted
WF	Worst Fit
W_R	Percentage of Work Rejected

x	A Tuning Parameter for Scaling Through Subset Scheduling Algorithm
y	A Tuning Parameter for Scaling Through Subset Scheduling Algorithm
α_1	One of the Five Parameters of a Two Phase Hyper-Gamma Distribution
α_2	One of the Five Parameters of a Two Phase Hyper-Gamma Distribution
β_1	One of the Five Parameters of a Two Phase Hyper-Gamma Distribution
β_2	One of the Five Parameters of a Two Phase Hyper-Gamma Distribution
ζ	Impact on Laxity
θ	Mean Error in Underestimated Runtimes as a Percentage of Mean Estimated Runtime
λ	Mean Arrival Rate
μ	Parameter of an Exponential Distribution
μ_1	One of the Three Parameters of a Two Phase Hyper-Exponential Distribution
μ_2	One of the Three Parameters of a Two Phase Hyper-Exponential Distribution
σ	Percentage of Mean Estimated Runtime that Equals τ in Time Units
τ	Quantum of Time in Feasibility Policy
υ_z	Speed of Resource b_z
χ	Tuning Parameter for Feasibility Policy and is Equal to θ / σ
ω	Total Nodes of a Shared Resource
r	Cutoff Point on x-axis
ϵ	A Set of Jobs in Resource Schedule

Chapter 1

Introduction

1.1. Introduction

We are experiencing a fundamental shift in the computing paradigm, away from the centralized monolithic mainframe systems towards a dynamic heterogeneous distributed approach. The scalability limitations of mainframe and file-sharing architectures gave rise to 2-tier client/server systems. Client/server systems performed well for locally distributed computing on a local area network with a limited number of users. But as the number of users and the complexity of interactions grew, performance of client/server systems started deteriorating and a third tier was added between the user system interface client environment and the database management server environment. The capabilities of the third tier, also known as the middleware, may range from resource management to load balancing, from database integration to clustering services, from security to providing interoperability in a heterogeneous environment. While the earlier well-known middleware technologies were successful in tightly integrated centrally managed settings such as within an enterprise, they have been ineffective in highly decomposed cross enterprise environments [FOS01]. With the rise of the Internet and success of e-business, the pressure for decentralization and distribution of software and hardware resources has increased tremendously. Not only Information Technology (IT) service decomposition is occurring within an enterprise but also there is a strong case for dynamic assembling of resources from cross enterprise and service provider systems [FOS01]. Moreover, there has been a growing awareness among companies that they can

achieve substantial cost savings by outsourcing non-essential components of their IT infrastructure to various service providers [FOS02a]. This has necessitated the need for new abstractions and architectures that can enable dynamic resource sharing in a geographically and administratively distributed setting.

Grid computing is emerging as a pinnacle of distributed computing evolution. A Grid is a system that "coordinates resources that are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver non-trivial qualities of services" [FOS02b]. Unlike its ancestors, Grid computing allows secure and coordinated sharing of heterogeneous resources across administrative and geographical boundaries. Through dynamic formation of *virtual organizations*, it enables the creation of virtual computing systems. Such systems consist of diverse distributed components which are sufficiently integrated to deliver a desired quality of service (QoS) [FOS01]. Virtually everything ranging from computers and networks to databases and scientific instruments can be viewed as potential Grid resources and can be shared among multiple Grid users in accordance with certain policies and service level agreements. Grid computing applications vary from distributed supercomputing [NOR96, NIE96] to high-throughput computing [LIT98], from data-intensive computing as in high energy physics [MAR96] to collaborative computing [ROU97], from business-to-business enterprise computing [FOS01] to pervasive and ubiquitous computing [HIN03], from remote medicine [CAM06] to digital sky surveying [FOS99a].

1.2. Motivation for the Thesis

Scalability, flexibility, quality of service provisioning, efficiency and robustness are the desired characteristics of most computing systems. Grids are scalable,

encompassing multiple organizations in multiple geographies, and flexible owing to their inherent dynamic nature that allows even runtime integration of heterogeneous resources from multiple domains. The need for Quality of Service (QoS) provisioning in Grids is well-established. Many Grid applications, including remote medicine [CAM06] and real-time control of sensitive sites and instruments [HOB05], require response time guarantees and often guaranteed allocation of resources in multiple domains. Quality of Service provisioning is so emphasized in Grids that it has been identified as one of the essential requirements for a distributed system to qualify as a Grid [FOS02b]. Efficiency of Grids depends on the effective utilization of Grid resources. Effective utilization of resources is not only necessary from the revenues standpoint in an economy-based Grid but it is also one of the essential requirements in non-business type sharing of resources, such as in scientific Grids. High efficiency results in cost-effectiveness and hence justifies the use of Grids. Since QoS objectives of resource consumers in Grids are often in conflict with efficiency goals of resource providers and since Grid resources are shared by multiple applications that belong to different administrative domains, it is challenging to meet the QoS objectives of applications while maintaining high system performance. On certain Grid systems, on Grids subjected to advance reservation requests for example, user-estimated execution times form an important parameter for the resource management. Such user estimates are often error prone (see [SMI03] for example) and execution times for applications may vary from one run to another. Thus, Grid systems must be robust enough to accommodate uncertainties such as those in user-estimated runtimes while meeting QoS and efficiency goals. This focus of this thesis is the development of a complete middleware framework for Grids that achieves user satisfaction by providing

QoS guarantees for Grid applications, cost effectiveness by efficiently utilizing resources and robustness by intelligently handling uncertain runtimes of applications.

For resource consumers in Grids, QoS generally relates to response time guarantees or guaranteed simultaneous availability of multiple resources. There are many examples of Grid applications that require response time guarantees. Specific examples include remote medicine [CAM06, KAR05a], air traffic flow management [WEI04], real-time control of sensitive sites and instruments [HOB05, KIM04], geo-processing [SHI02], particle physics [KAR05b, KAR05c] and multimedia [SAW03]. Various different types of Grid resources such as network, sensors, instruments, computational and storage resources may be involved in such scenarios. As Grid resources are being shared by multiple applications which are completely unaware of each other, it is difficult to provide response time guarantees to the applications in these environments. Unpredictable delays over Wide Area Networks further increase the difficulty in providing response time guarantees to the Grid applications. Moreover, Grid applications generally have rich requirements 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. As the interaction patterns of the applications grow more complex and as the number of applications and resources increases, one feels a pressing need for a scalable middleware level solution to the problem.

To meet the QoS objective of the resource consumers advance reservations of resources are required. Although the incorporation of advance reservations in the workload may lead to increased response times for *best-effort* jobs, it is essential in many situations. For example, to enable simultaneous availability of multiple resources in multiple domains, advance reservations of resources are necessary. Advance reservations of resources are also required for providing response time guarantees to the Grid applications. As advance reservations may result in poor system performance, performance engineering of advance reservations enabled resource management middleware and appropriate control of the proportion of advance reservations in the workload are necessary to meet the objectives of both the resource consumers and the resource owners.

The objective of the resource owners participating in a Grid is high utilizations of their resources. For service (resource) providers, high utilization of resources may mean high revenues and unless there is some guaranteed levels of expected utilization there may not be any incentive for them to participate in the Grid in the first place. As discussed earlier, effective utilization of resources is also one of the essential requirements in non-business type sharing of resources, such as in scientific Grids.

Provisioning a desired level of QoS to the consumers and effective utilization of resources in Grids is a challenging task but it is fundamental for the success of Grids. Little research that addresses both of these concerns exists in the area. This thesis attempts to fill this gap with the devising of a resource management framework that can meet the objectives of various stakeholders – resource consumers and resource providers – in the Grid. To the best of our knowledge, this is the first framework that

comprehensively tackles the problem of QoS provisioning in Grids and allows efficient use of resources even in the presence of uncertainties. A more detailed description of the limitations of the existing research and the challenges tackled by the thesis is presented in Section 2.6.1.

1.3. Contributions of the Thesis

The thesis has contributed to the state of art in resource management in Grids and various research papers have been published [FAR05a, FAR05b, FAR05c, FAR06a, FAR06b, FAR06c, FAR07] or submitted for publication [FARira, FARirb, FARirc]. The major contributions of the thesis are listed.

- The thesis presents a framework for resource sharing in multi-institutional geographically distributed Grids that can provide QoS guarantees to Grid applications while maintaining high system performance even in the presence of uncertain runtimes. The framework relies on the notion of under-constrained advance reservation requests (ARs) which have laxity (defined in Section 2.6) in their reservation windows. The framework presented by this thesis is compatible with Grid standards and current Grid technologies and practices and is applicable to a variety of scenarios. The thesis argues that some of the resource management strategies and conclusions have a broader appeal and are applicable for other multi-institutional resource sharing infrastructures.
- Scheduling ARs with laxity is an NP-Complete Problem [GAR79]. The thesis presents a comprehensive set of scalable algorithms for tackling the problem for different possible scenarios. Different algorithms are proposed to best suit the needs of a particular scenario.

- The Scaling through Subset Scheduling (SSS) algorithm is developed for scheduling advance reservations with laxities for non-shared resources (defined in Section 2.6) where each job requests the exclusive use of the resource. SSS is adapted from algorithms in real-time domain to cater to the needs of the Grid environment. It is a scalable algorithm for handling large number of jobs and it can efficiently schedule a mix of ARs and best-effort jobs (also known as On-Demand requests (ODs)) for achieving high resource utilization. It is also applicable for scenarios involving both non-preemptable and preemptable jobs including those in which overheads are associated with job preemption. The thesis studies with the aid of an extensive set of experiments, the effect of workload parameters such as proportion of advance reservations and laxity on performance. This thesis also investigates how much improvement in performance can be gained by preempting jobs (segmenting data in case of network resources) and up to what percentage of overheads is preemption (segmentation) justified in scheduling of on-demand and advance reservation requests in Grids.
- The Grid Scheduling with Deadlines (GSD) algorithm is developed for efficient scheduling of ARs with laxities and OD requests on homogeneous shared resources (defined in Section 3.1.2) where a number of Grid jobs can run in parallel with each other. Just like SSS, GSD is a scalable algorithm and can handle non-preemptable and preemptable jobs. GSD is adaptable to various workload conditions and it can gang schedule various tasks of the jobs to allow for inter-tasks communications. The effect of various workload and system parameters on the performance of GSD has been studied in detail. The thesis

analyses the worst case time complexity of GSD and demonstrates that GSD is a scalable algorithm.

- With the help of comparisons with theoretical bounds, the thesis demonstrates that the SSS and GSD scheduling algorithms obtain near-optimal schedules.

- In any form of distributed computing, effective application-to-resource mapping and load balancing are important to meet performance objectives. This process is generally referred to as matchmaking. With the help of simulation results, the thesis demonstrates that traditional matchmaking algorithms used in distributed computing result in poor performance when applied in multi-institutional settings particularly for workloads consisting of both advance reservations and best-effort jobs. The thesis presents different algorithms for mapping ARs and OD requests and through simulation investigates their performance in detail.

  - A novel algorithm for matchmaking, Minimum Laxity Impact (MLI), is developed which outperforms all other algorithms investigated in every respect for a wide range of workload parameters. MLI results in the lowest response time for the users while providing the highest resource utilizations. Another application of MLI is as an effective meta-scheduler to dispatch jobs to multiple resources in multiple domains. For example, MLI can be used as a meta-scheduler within Platform Computing's Community Scheduler Framework (CSF) [PLA05c]. Effect of various workload and system parameters such as proportion of advance reservations, laxity and size of candidate set on the performance of matchmaking algorithms has been studied in detail.

- The thesis presents extension of MLI for scenarios involving heterogeneous resources. Rigorous experimentation is conducted to investigate the performance of variants of MLI. A variant of MLI, called MLI-SDP, has been developed that accommodates a self-partitioning mechanism and outperforms all other algorithms investigated for a broad range of workload parameters experimented with.

- On certain Grid systems, on Grids subjected to advance reservation requests for example, user-estimated execution times form an important parameter for resource management. Such user estimates are often error prone (see [SMI03] for example) and execution times for applications may vary from one run to another. Providing QoS guarantees while maintaining high system performance in the presence of inaccurately estimated runtimes is a difficult task. The thesis studies in detail the impact of error in runtimes on system performance and presents novel strategies to deal with it.
  - Schedule Exceptions Manager (SEM) is developed as a part of the framework to deal with runtime exceptions and uncertain runtimes in the resource schedule. SEM adapts the resource schedule to the changing conditions and hence is able to obtain high useful utilizations (defined in Section 4.2.8) even for the scenarios involving abnormal terminations and inaccurately estimated runtimes. Performance of SEM has been investigated in detail with the help of a comprehensive set of experiments.
  - The thesis proposes novel pre-scheduling policies to avoid loss in performance resulting from extremely large errors in estimation.

- The thesis systematically investigates the merits and de-merits of overestimating/ underestimating runtimes so as to establish guidelines for runtimes-predicting-algorithms.

- Since different resources in Grids may be administered and scheduled independently, simultaneous co-allocation of multiple heterogeneous resources in multiple domains is a challenging problem. However, many Grid applications, such as those in data mining and computational-steering, and workflows may benefit from coordinated resource usage at multiple sites. The thesis presents a multi-resource coordinator that can effectively allocate multiple resources in multiple domains for the applications that need to access multiple resources simultaneously. Different algorithms for co-allocations have been investigated and their performance has been studied with the help of simulation experiments.
- The thesis proposes a resource-level policy to prevent starvation of OD requests and study its performance.

To the best of our knowledge, the framework presented by the thesis is the first comprehensive framework for resource management with ARs. Different components are being integrated into the well-known Globus Toolkit [GLO05c] and CSF [PLA05c] with the aim of making them available for users around the world. The integration with Globus Toolkit has already been used to build QoS enabled Grid compute clusters [FAR06c].

1.4. Thesis Outline

The thesis comprises ten chapters. Chapter 2 provides an overview of Grids and the current state of the art in resource management with Grids. Chapter 3 describes in

detail the framework proposed by this thesis while Chapter 4 presents the experimental setup. Chapter 5 and Chapter 6 discuss scheduling on non-shared and shared Grid resources, respectively. Chapter 7 discusses strategies to manage uncertain user-estimated runtimes. Chapter 8 and Chapter 9 present matchmaking and co-allocation strategies, respectively. Chapter 10 concludes the thesis and presents directions for future research.

Chapter 2

Overview of Grid Computing

2.1. Grid Computing

A Grid is a dynamic collection of heterogeneous resources belonging to multiple administrative domains and possibly spread over wide-area networks. These resources are coordinated through secure mechanisms to solve a common problem and are sufficiently integrated to deliver a certain level of quality of service.

Ian Foster, one of the pioneers of the Grid, provided a three point check-list that must be satisfied by a system for it to qualify as a Grid [FOS02b]. According to this check-list a Grid is a system that [FOS02b]:

"coordinates resources that are not subject to centralized control..."

"... uses standard, open, general-purpose protocols and interfaces..."

"... delivers non-trivial qualities of service".

The first point in the checklist emphasizes that a Grid, unlike a local management system, integrates resources belonging to different domains and handles issues associated with this integration such as security, billing and resource management. Cluster management systems such as Platform's Load Sharing Facility (LSF) [PLA05a] and Viridian's Portable Batch System (PBS) [ALT04] that like Grids bring resources together are examples of powerful Grid resources. However, they do not qualify as Grids themselves because of the centralized control of the resources they manage. On the other hand, peer-to-peer systems such as Gnutella [GNU04] and Kazaa [SHA04] that, like Grids, allow users in multiple domains to share files cannot qualify as Grids either. This

is because Grids are not limited to file sharing but they allow sharing of other resources as well.

The second point stresses that protocols and interfaces used for addressing issues such as authentication, authorization, resource discovery and resource access should not be application specific but rather should be standardized and generalized such that they allow dynamic resource sharing with *any* interested party. Distributed computing systems such as Platform's Multi-Cluster [PLA05b], Condor [LIT98] and Entropia [ENT06], like Grids, integrate distributed resources in the absence of a centralized control to deliver some level of qualities of service. However, since the protocols used by these systems are too specialized and are not open and standard, these systems can qualify as Grids only in a limited sense.

The third point makes delivery of some level of quality of service to resource consumers and resources owners in a Grid one of the essential requirements for a system to qualify as a Grid. As noted in Section 1.2, some level of QoS assurance for both resource consumers and resource owners is essential for the wide applicability of Grids. The World Wide Web that supports access to distributed resources using standard open protocols thus cannot qualify as a Grid as it does not provide for the coordinated use of resources to deliver some level of QoS.

2.2. Grid Applications

In recent years Grid computing has found immense acceptance from wide application circles and many national and continent levels Grids have emerged. A few significant Grid projects include European Union's EUROGrid [EUR05], United Kingdom's E-Science Grid [REA06], United States Department of Energy's Science

Grid [DOE05], European Union's Data Grid [DAT05], NSF's TeraGrid [TER04], Asia-Pacific Grid [APG06], Germany's Unicore Grid [UNI04], Dutch Grid [DUT06], Irish Computing Grid [GRI06] and many more.

Applications of the Grid can be categorized into the following broad categories [FOS99a].

2.2.1. Distributed Supercomputing

The applications in this category use Grids to assemble enough resources to solve a complex problem that cannot be tackled on a single system. Some of the examples in this category include distributed interactive simulation [DON06], stellar dynamics [NOR96], computational chemistry [NIE96] and climate modeling [MEC93].

2.2.2. High Throughput Computing

The applications in this category schedule independent or loosely coupled tasks on unused processor cycles in idle computers. Examples in this category include chip design [FOS99a] and search for extra-terrestrial intelligence [LIT98].

2.2.3. On-Demand Computing

The applications in this category use Grids to meet short-term requirements for resources that may not be available locally and it may not be cost effective to keep them locally. Examples include numerical solver systems [CZY96], remote imaging [POT96] and cloud detection [LEE96].

2.2.4. Data-Intensive Computing

Applications in this category use Grids to aggregate data distributed in data-repositories, databases and digital libraries at various geographical locations. Examples

are high energy physics project [ALL01a, MAR96], digital sky survey [FOS99a] and meteorological forecasting systems [FOS99a].

2.2.5. Collaborative Computing

Applications in this category use Grids to enable and enhance human-to-human interactions. Examples include education [ROU97] and collaborative design [WHE96].

2.2.6. Enterprise Computing

Applications in this category use Grids for business-to-business enterprise computing to enhance their capability and/or to form scalable, reliable and secure distributed systems. Outsourcing some of the non-essential IT components can result in significant costs savings. This has resulted in substantial growth of application service providers, cycle providers, storage providers, content distribution service providers and so on. Emerging applications in this category include financial forecasting and decision making [FOS01].

2.2.7. Pervasive Computing

With the evolution of Grid, more and more small smart devices ranging from personal digital assistants to laptops, mobile phones to tiny sensors are becoming a supportive force to Grid computing. Some of the important research for pervasive computing applications in Grid includes [GHO04, HIN03, PHA02].

2.2.8. Other Applications

Other novel applications are also emerging with the evolution of Grids. Examples include remote medicine, Grid gaming and interactive high definition television.

2.3. Grid Standards

As mentioned in Section 2.1, Grid protocol and interfaces for addressing issues such as authorization, resource discovery and resource access should be general and *standard*. For standardizing Grid specifications and architectures, Global Grid Forum (GGF) [GLO05a] was established. GGF is a public community forum for the discussion of Grid technology issues and currently more than 400 organizations from industrial and academic circles from around the world are involved with GGF. The goals of GGF include creation of an open process for the development of Grid specifications and agreements, and establish Grid architecture documents and best practices guidelines. Various research groups within GGF are currently working on issues such as application and programming model, security, architecture, and scheduling and resource management.

Another important forum for Grid developers is Globus Alliance [GLO05b]. Globus Alliance is an international collaboration conducting research for the development of fundamental Grid technologies. Globus Alliance has developed a basic infrastructure and high-level services for a computational Grid in the form of Globus toolkit [GLO05c]. Globus toolkit implements Grid standards and is the most widely used infrastructure for Grid computing in industry.

2.3.1. The Layered Grid Architecture

A standard high-level layered Grid architecture, shown in Figure 2.1, was specified in 2001 [FOS01] to identify requirements for general classes of components. The components within each layer in Figure 2.1 share common characteristics and builds upon capabilities provided by lower layers. The *fabric layer* in the Grid layered architecture consists of Grid resources to which shared access is mediated by Grid

protocols. A Grid resource may be a physical entity such as a CPU, a sensor or a storage component or it may be a logical entity such as a distributed file system, a lightpath or a compute cluster. Fabric layer components implement local resource-specific functions. The richer the functionality of fabric layer components, the greater the sophistication of sharing operations they can support. For example, fabric layer components that support advance reservations enable them to be co-scheduled with other components.

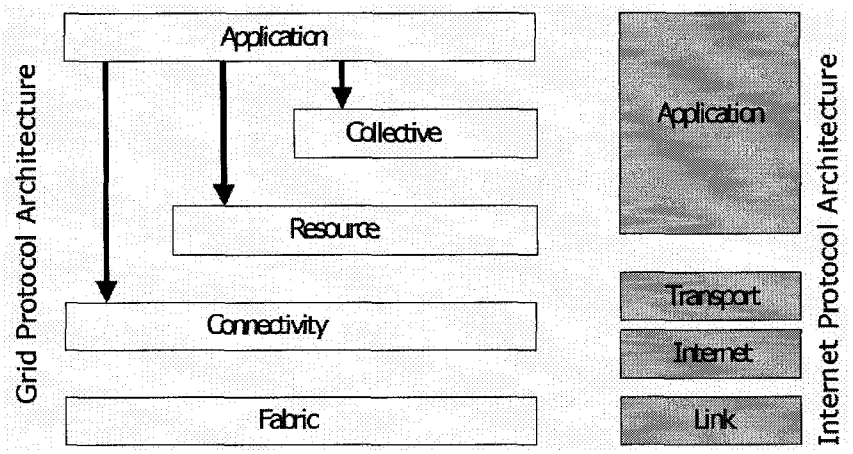


Figure 2.1: The Layered Grid Architecture and Its Relationship to the Internet Protocol Architecture [FOS01]

The *connectivity layer* defines communication and authentication protocols for Grid network transactions. The *resource layer* uses the functionality provided by the connectivity layer for the secure negotiation, initiation, monitoring and control of operations on individual fabric layer resources. While the resource layer is focused on interactions with a single resource, the *collective layer* coordinates multiple resources and provides application specific distributed services. Some of the examples of services at the collective layer include *directory services* that can discover the existence and properties of resources and *brokering services* that can request the allocation of one or more resources for a specific purpose. The *application layer* consists of Grid applications that use the services of lower layers to achieve a particular goal. Each layer has well-defined

protocols and provides useful services to the application such as resource discovery, resource management and data access.

2.3.2. Open Grid Service Architecture (OGSA)

The Grid infrastructure aims to provide for the creation, management and application of dynamically coordinated resources and services that may be single entities or consist of collection of component services. These resources and services may have a short or a long lifespan and they may be comprised of single or multiple administrative domains. The complexity involved in the creation and management of these dynamic and coordinated services prompted the introduction of Open Grid Service Architecture (OGSA) [FOS02a]. The goal of OGSA is to define resource models and resource profiles with interoperable solutions, to identify and to define core OGSA platform components and to define hosting and platform-specific service bindings [JOS03].

OGSA presents a *service-oriented view* of the physical resources being shared or the services supported by these resources. The OGSA architecture, shown in Figure 2.2, shows clear separation of functionality at each layer [JOS03]. In this figure, Open Grid Services Infrastructure (OGSI) forms the base infrastructure for the creation and management of services. OGSA platform services are a set of standard services such as policy, service level agreements, logging, accounting and security. High-level applications use these services to form more specialized domain specific services.

2.3.3. Open Grid Services Infrastructure (OGSI)

OGSA defines various aspects related to Grid services, such as interfaces that are needed. However, it does not define how the interfaces should be implemented. Open Grid Services Infrastructure OGSI [FOS05] defines mechanisms for creating and

managing *Grid services* and acts as a technical specification for implementing the Grid services defined in OGSA. A Grid service is a web services that follows a set of conventions (interfaces and behavior). These conventions define how a client interacts with the Grid service. OGSI extends Web Services Description Language (WSDL) [W3C05a] and XML schema to incorporate the concept of stateful web services, asynchronous notification of state changes, references to instances services, collections of service instances and service state data that augments the constraint capabilities of XML Schema definition [FOS05]. The interfaces and behaviors that define a Grid service are discussed in detail in [FOS05].

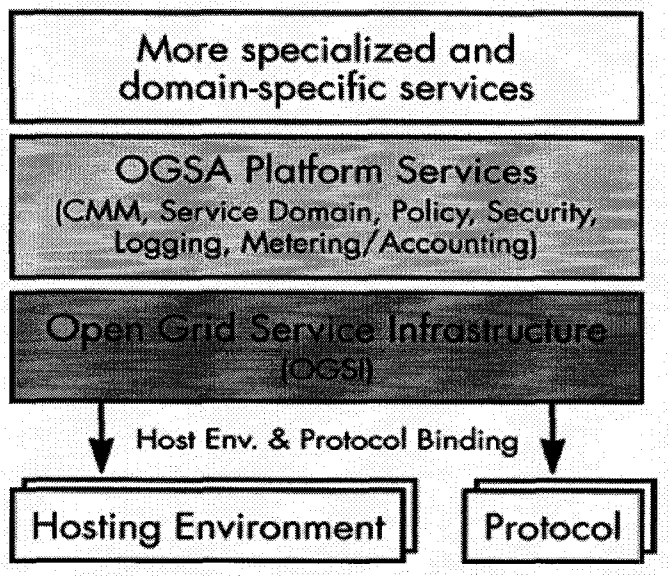


Figure 2.2: Open Grid Services Architecture [JOS03]

2.3.4. Web Services Resource Framework (WSRF)

Web Services Resource Framework (WSRF) [CZA04] was proposed as a refactoring and evolution of OGSI to exploit new web services standards such as WS-Addressing [W3C05b]. While WSRF retains all the functional capabilities of OGSI, it makes some changes in the syntax to exploit WS-Addressing and use different terminology in its presentation. It also partitions OGSI functionality into five distinct,

composable specifications. The relationship between OGSI and WSRF and the related WSRF specifications are discussed in detail in [CZA04].

2.4. Resource Access and Management in Grids

Resource access and management in Grids is a challenging task because of dynamic, heterogeneous resource sharing where the resources may belong to different administrative domains and may spread over wide area networks. This section briefly reviews the steps involved in using a Grid service and presents the current state of the art in the challenges involved in each step.

2.4.1. Resource Discovery

The first step in using a Grid resource is to discover the available resource(s) that can meet the requirements of the Grid application. Information about the available resources is gathered from Grid Information Services (GIS).

Globus Toolkit's implementation of GIS is called Globus Monitoring and Discovery System (MDS) [GLO05d]. A standard MDS architecture consists of the Globus Resource Information Service (GRIS) and the Globus Index Information Service (GIIS). GRIS runs on each resource and provides information about the resource. It is based on the Light Weight Directory Access (LDAP) protocol. GIIS caches information from GRIS and provides the collective level indexing and searching functions. Resources register with GIIS which then updates information from GRIS whenever a cache entry expires.

Another example of architecture for information gathering is the Grid Monitoring Architecture (GMA) [GLO05e], being developed by a research group at GGF. GMA concentrates on the provision of streaming data.

Grid monitoring systems must agree on the schema to describe the attributes of the system to enable different systems to understand the meaning of the information collected by GIS. Some of the significant research in this area includes the CIM-based Grid schema [GLO05f], the discovery and monitoring of event data [GLO05g] and the GLUE information model [GLU05].

Assuming that a Grid application has access to a GIS, the next step for the application is to locate resources which it is authorized to acquire. Although there exists a number of mechanisms to provide security to Grid application once they have accounts to run a job on a resource [GLO05h], there is still a substantial effort needed to address the issues of account management [SCH02].

Once the Grid application has located the resources which it is authorized to acquire, it must reduce the set of candidate resources to those which can meet its minimum functional requirements. Currently, the application can specify its job requirements as a part of the command line or submission script as in systems such as PBS [ALT04] and LSF [PLA05a]. Another mechanism for specifying job requirements is in the form of a small advertisement called *ClassAd* [RAM98]. This approach is adopted my systems such as Cactus [ALL01b] and European Data Grid broker [GIA01].

2.4.2. Advance Reservations (ARs)

Once the set of resources that meet the minimum functional requirements of the resources has been located through the discovery process, the application can either submit the jobs to the queues of the resources or it can optionally make advance reservations on the resources for a particular time in future. An advance reservation is characterized by a *start time* of the reservation, representing the time at which the job

would be available for execution on the resource, and *duration* of the reservation, representing the period of time for which the job would need the resource. ARs were introduced as a part of Globus Architecture for Reservation and Allocation (GARA) [FOS99b] for two primary reasons. First, it allows different jobs of a Grid application to be co-scheduled on different resources at the same time. Without advance reservations there is no guarantee that jobs submitted to the queues of different resources possibly belonging to different administrative domains would be allocated at exactly the same time for execution. The second reason for the introduction of ARs is that it guarantees the resource allocation for Grid application and thus given an estimate of job execution time on the resource, an upper bound on the response time of the job can be calculated. ARs can hence be used to provide QoS guarantees to resource consumers.

Since their introduction, ARs have been studied in numerous contexts such as in architectures for ensuring end-to-end quality of service for network applications [FOS00], architectures for data-intensive collaboration [FOS03], scheduling of data placement activities [KOS04], job schedulers for clusters and supercomputers [MAU04] and Grid based architectures for dynamic optical networks [LAVir]. Smith [SMI00] and Sulistio [SUL04] have investigated the performance of advance reservations based scheduling. Figueira et al. [FIG04a] 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 lightpaths requested by ARs [FIG04a].

2.4.3. Matchmaking

Given a set of resources that meet the minimum functional requirements of the resources, the next step for the application is to select a single resource or a single

resource set. The thesis refers to this process as *matchmaking*. As an example, let us consider a Grid application that needs a certain number of nodes s on a cluster. Through the discovery process, the Grid application would locate a set of clusters that have s nodes available at the desired time of the application. Selection of a particular cluster, to run the application on, from the candidate clusters located through the discovery process is called matchmaking.

One of the notable works for this application-to-resource mapping is Sun Grid Engine's load averaging mechanism [SUN05] that has two primary policies. In one of the policies, the resources are sequenced and each time the list is iterated through to select the next resource. In the second policy, the resource with the minimum load is selected to run the next job. Other works in this category include [CAS00] where an application-level scheduling system for applications consisting of independent tasks has been presented. They use different heuristics to try to minimize the job completion time. Similarly, the Grid Harvest Service [SUN03] always assigns the job to the least loaded resource with the same goal as in [CAS00], namely minimizing job completion time.

Some of the research on resource selection in Grids focuses on using predictions to locate a suitable resource [SCH03, SMI03, WOL03] while research such as [BUY00, BUY02] uses computational economy in resource selection. Still other works include those that use a multi-criteria approach [KUR03] and those that use meta-heuristics approach [MIK03].

The Community Scheduler Framework (CSF) [PLA05c] is an open source framework for implementing a Grid *meta-scheduler* that can dispatch jobs to underlying resource managers. Currently CSF uses two mechanisms for resource selection. In one of

the mechanisms it finds a resource based on a Round Robin choice. In the other mechanism, it uses a job-throttle to limit the number of jobs submitted to the same resource.

There thus exists a wide body of research on matchmaking. However, almost all of the research in this area focuses on best-effort on-demand requests. As the results presented in Section 8.5.1 of this thesis show that commonly used mapping algorithms used in Grid and distributed computing in general do not give a desired performance with advance reservations. An efficient matchmaking algorithm for advance reservations based scenarios is highly desired. Such a matchmaking algorithm is one of the focuses of the thesis.

2.4.4. Job Submission and Monitoring

Once the application has selected the resources to submit the job to or has made advance reservations on selected resources, the next step is to submit the job to those resources. Job submission in Grids may involve just a simple execution of a single command or it may involve running a series of scripts. Globus Alliance has developed Globus Resource Allocation Manager (GRAM) [CZA98] that provides an API for submitting and canceling a job request on a resource. GRAM which has become an essential component of Globus Toolkit can process requests for resources for remote execution of a job, can allocate the required resources and manage active jobs. For data transfer requests on a dynamic optical network, Lavian et al. presents Grid network service architecture [LAVir]. They encapsulate the optical network into an OGSI-compliant network service to which a user can submit a job just like any other Grid service.

The job submitting process in Grids can be complicated by lack of standards for job submission. That is why there is substantial research being conducted on issues such as common APIs for job submission [GLO05i], common languages [GLO05j] and common protocols [GLO05k]. If the job submission process involves transferring of data files to the selected resources, an application can ftp or scp [SEC04] the required files. For large data transfers however, GridFTP [GLO05l] is most commonly used.

Once the job has been submitted to the selected resources, the application can monitor the progress of the job. GRAM provides an API for checking the status of the submitted job as well. If the job is not making sufficient progress on a resource, the job can be reallocated to another resource. However, this process might involve significant overheads especially if the job has to be restarted from beginning on the new resource. For this purpose systems such as Condor [LIT98] provides for preemptive resume scheduling where the jobs can be checkpointed, preempted and resumed elsewhere.

On the successful completion of a job on the resource, the application can be notified. This is often done by providing an email notification parameter in the submission script. The application then retrieves the files from the resources using ftp, scp or GridFTP and also reclaims the resource by removing, for example, temporary settings.

2.4.5. Co-Allocation

If different jobs of a Grid application need to access different resources at the same time, co-allocation of resources is required. For this purpose Dynamically Updatable Resource Online Co-allocator (DUROC) was introduced [FOS99b]. With DUROC, an application needing co-allocation submits its requirements to co-allocation

agents of DUROC architecture. The co-allocation agents interact with GIS to find the suitable resources for the applications. They then allocate each resource by redirecting the allocation request to either the local resource manager or GRAM. After successful authentication, the resources are allocated to the job and a *job handle* is passed back to the application. The application can use the job handle to monitor and control the job.

DUROC does not support advance reservations and hence cannot guarantee that a resource can provide the desired QoS. This limits the ability of DUROC to perform co-allocation as different resource may be in high demand. For this purpose, the Globus Architecture for Reservation and Allocation (GARA), supporting advance reservations, was introduced [FOS99b]. GARA consists of co-reservation agents in addition to co-allocation agents. The co-reservations agents interact with GIS to locate suitable resources for the application, reserve those resources for some interval of time in future and provide the *reservation handle* back to the application. The application can use the reservation handle to modify or cancel the reservation. At the execution time of the application, the application uses the services of the co-allocation agents to submit its jobs to different resources just as in DUROC.

2.5. Achieving Quality of Service in Grids and Its Challenges

As mentioned in Section 2.1, providing non-trivial QoS is one of the essential requirements for the system to qualify as a Grid. From the applications perspective, QoS generally relates to response time guarantees and/or guaranteed allocation (co-allocation) of (multiple) resources to meet complex demand patterns of the applications. The response time requirements are generally imposed by the nature of the Grid applications such as those in remote medicine and multimedia. Examples of Grid applications where

response time guarantees are needed include earth quake monitoring and real time control of sensitive sites [KIM04], particle physics projects such as KATRIN [KAR05c] and AUGER North [KAR05b], ultrasound computer tomography [KAR05a], air traffic flow management [WEI04], distributed immersive performance [SAW03], interactive control and management of sensitive instruments [HOB05], real-time distributed collaborative geo-processing [SHI02] and radiology image transfers across a metro-network [CAM06]. Other emerging Grid application such as interactive high definition television and Grid gaming would also require response time guarantees from Grids. As Grid resources are being shared by multiple applications which are completely unaware of each other, it is difficult to provide response time guarantees to the applications in Grids.

Guaranteed co-allocation of resources is often required for the application to function in the desired manner. As an example, consider an application that needs to transfer data stored in administrative domain A to another administrative domain B using the lightpath that belongs to yet another domain C. Until all the required resources in domain A, B and C are simultaneously available, data cannot be transferred correctly between the source and the destination. One way to arrange for the simultaneous availability of the resources in multiple domains is to manually allocate the resources through the domain administrators of all the concerned domains. However, as the interaction patterns of the applications grow more complex and as the number of applications and resources increases, the need for a scalable middleware level solution to the problem becomes increasingly important.

Provision of guaranteed level of QoS to Grid applications is thus complicated by two factors. First, the rich requirements of the Grid applications for heterogeneous

resources that are being independently administered or controlled and second, the unpredictable delays over Wide Area Networks. As discussed in Section 2.4.3, advance reservation of resources provides a way to co-allocate multiple resources in multiple domains as well as to ensure response time guarantees to the applications. Moreover, just like other resources, Grid-enabled network resources can also be reserved in advance for a more predictable network delay. The thesis thus uses advance reservations as a mechanism for meeting the QoS constraints of the applications.

2.5.1. Challenges for Resource Management with Advance Reservations

Resource management with advance reservations differs from traditional resource management. This is because in AR-based systems the start time of an application is not decided by the scheduler. Instead, the application tries to control its own behavior by specifying its start time. This selection of start time by the application may be motivated by the time constraints of the applications or it may be the result of an application's need to co-allocate multiple resources possibly in multiple administrative domains. While in traditional scheduling, a scheduler is deployed to effectively order and assign jobs to available resources to maximize system performance, in AR based scheduling each application is completely unaware of the other jobs in the system and their scheduling decisions. Thus as the proportion of ARs increases, one finds increasingly poorer scheduling decisions being made from the system perspective. This has been observed in previous research on ARs [SMI00, SUL04]. Their results show that with only 20% of the jobs arriving as advance reservations, the utilization can fall to less than 66% of that 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%. Hence, although advance reservations are attractive for resource consumers, they may cause severe performance degradation for the resource owners. A framework for resource sharing with ARs is thus needed that can meet the objectives of both, providing QoS guarantees to resource consumers and ensuring high utilization of resources for resource owners.

Another issue for resource management with advance reservations is to accurately estimate the execution time of the job on a resource so that the resource can be reserved for enough time to successfully complete the job. Most current job scheduling systems require users to specify maximum runtimes of their jobs. The investigation of traces resulting from such systems however shows that such a scheme results in extremely large errors in runtimes and consequently extremely poor utilization of the resources [FAR06a].

Error in estimation of runtimes can be reduced by using a suitable runtime-predicting-algorithm. There exists a wide body of research on predicting runtimes of the jobs that can be divided into two broad categories. In the first category are those that use statistical analysis of applications that have completed to predict runtimes. Examples of these categories include papers such as [DIN01, DOW97, SMI02]. In the second category, analytical analysis is done to construct equations describing application runtime. Some of the notable works in this category are [SCH98, TAY01]. Research however shows that the above-mentioned techniques can provide only rough estimate of the actual runtimes [SMI03]. Although techniques presented in [SMI03] reduce the error in estimation, the mean error for the kind of workload used by this thesis was still between 47.47 to 98.28%. This research shows that if not handled properly such an error

can significantly degrade system performance [FAR06a]. The framework for resource management with ARs thus should be capable of dealing with such uncertainties in runtimes of the jobs.

2.6. Approach of the Thesis for Achieving QoS for Applications and High System Performance in Multi-Institutional Grids

The thesis relies on the powerful mechanism of advance reservations that can help co-allocate multiple resources and provide response time guarantees for the application. However, to help prevent performance degradation for resource consumers, the thesis uses the idea of under-constraining advance reservation requests by introducing laxity in the reservation window. *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 [LAVir]. ARs with laxity allow the job to choose its own start time window as necessitated by its constraints but it leaves the final scheduling decision with the domain scheduler. The domain scheduler is free to re-order jobs to improve system performance as long as it meets the time constraints of all ARs accepted. Laxity in the reservation window can thus improve the performance of advance reservation based scenarios by providing more flexibility in scheduling.

The idea of laxity is not new to the scientific community where jobs are commonly specified with a ready time, an execution time and a deadline, the deadline being greater than the sum of the ready and execution times. Lavian et al. [LAVir] also follow the same approach for specifying data transfer requests in a Grid based dynamic optical network.

Advance reservations with laxity are characterized by a *start time of the reservation window* that specifies the earliest time at which the job would be available for

execution, an estimated execution time of the job on the resource and a *deadline* by which the job must be completed by the resource. In order to understand the challenges for resource management with advance reservations with a given start time, execution time and a deadline, let us consider the following two models of Grid resources.

Non-Shared Resource Model: If a Grid resource is used in such a way that each job requests the exclusive use of the resource during its execution, the thesis refers to the sharing model as the *non-shared resource model.* Examples of a non-shared resource model include a single CPU based computing device that can run only one job at a time.

Shared Resource Model: If a Grid resource can be shared among multiple jobs running concurrently on the resource, the thesis refers 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 may run on a node at a time. With the introduction of virtual machines in Grids [FIG03], it is possible to allocate certain proportions of even single CPU based computing devices to different jobs. In such cases, such devices act as shared resources and their shares of CPU cycles as their nodes (components).

2.6.1. Challenges

Some of the challenges for resource management with ARs having laxity are presented along with an indication of how they are addressed in the thesis.

2.6.1.1. Efficient Scheduling of ARs with Laxity

The efficiency of a system based on ARs with laxity depends on the effectiveness of the resource scheduler deployed. An efficient scheduler is thus needed that seeks to maintain high resource utilizations while ensuring that deadline constraints of Grid jobs are met. It has been shown that optimal scheduling of jobs with a given start time,

execution time and a deadline is an NP-Complete problem even if we consider a non-shared resource model [GAR79]. The scheduler thus must use heuristics in such a way that it can handle large number of requests while leading to near optimal schedules.

An additional constraint for the shared resource model based scenarios is that different tasks of a job may need to communicate while executing. The shared resource scheduler thus must be able to gang schedule different tasks of a job on the shared resource.

Algorithms presented in the real-time domain for similar problems generally have different goals – instead of maximizing utilization of the resource they try to minimize maximum lateness. For non-shared resource models, most of the algorithms [FOH95, LIU73] assume that all jobs are preemptable and resumable. This might not be true for Grid based scenarios where many jobs may not be preemptable or the overheads associated with preemption are too high that renders preemption impractical. Algorithms for non-shared resource model such as [MCM75, XU90] that can handle non-preemptable jobs do not scale for large number of jobs. It has been reported in [XU90] that for a problem size of 100 jobs, the algorithm presented in [MCM75] was unable to terminate after generating several tens of thousand of *solution-nodes* (temporary schedules) while the algorithm in [XU90] generated a few thousand solution-nodes. The number of solution-nodes grows exponentially as the problem size increases [XU90]. Moreover, 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. Such algorithms are thus not readily applicable for Grid based scenarios.

For the shared resource model as well, most research such as [FUN01, ZHA01] in the real-time literature assume completely preemptable and/or periodic tasks. Papers such as [MAN98, RAM90] that can support non-preemptable jobs on multiple resources do not support gang scheduling.

The thesis presents the Scaling through Subset Scheduling (SSS) and the Grid Scheduling with Deadlines (GSD) algorithms for efficient scheduling on non-shared and shared Grid resources respectively. Both SSS and GSD are scalable algorithms and they can handle both non-preemptable and preemptable Grid jobs. GSD supports gang-scheduling as well.

2.6.1.2. Support for Preemption

While for most of the applications in Grids it may not be feasible/desirable to preempt a job, in some scenarios preemption may make sense. For data transfer jobs on a network, preemption is equivalent to temporal data segmentation. The Grid scheduler thus should be capable of handling preemption (data segmentation) and for the cases where overheads are associated with job preemption (data segmentation), it should try to minimize preemption overheads. To the best of our knowledge, preemption has not been studied in detail in the context of ARs.

Both SSS and GSD supports preemption and they incorporate mechanisms to minimize preemption overheads.

2.6.1.3. Preventing Starvation of On-Demand Requests

As discussed in Section 2.5.1, advance reservations of resources may result in significant increase in response time of OD requests. Increase in response time of OD requests would encourage applications to submit their jobs as ARs which would increase the proportion of advance reservations. Increase in proportion of advance reservations

may significantly bring down system utilization and further increase the starvation of OD requests. Resource management frameworks should thus be capable of avoiding such situations that lead to starvation of OD requests. To the best of our knowledge, starvation prevention in the presence of deadline constrained ARs have not been studied. This thesis presents resource level strategies to prevent starvation of OD requests in the presence of ARs.

2.6.1.4. Efficient Matchmaking

A number of matchmaking techniques were reported in Section 2.4.2. However, almost of all of those techniques were designed for scenarios involving only OD requests and as the results in Section 5.3 would show those techniques perform poorly in terms of the high proportion of work rejected, low utilization and high response time when ARs are introduced. An efficient matchmaking algorithm for AR based scenarios is thus required. Meta-scheduling with ARs in CSF also requires an effective matchmaking mechanism.

The thesis presents the Minimum Laxity Impact (MLI) algorithm that outperforms all other algorithms investigated for AR based scenarios for a wide range of workload and system parameters. MLI can also act as an effective meta-scheduler and it has been extended to support scenarios involving heterogeneous resources.

2.6.1.5. Handling Uncertainties in User-Estimated Runtimes

As discussed in Section 2.5.1, error in user-estimated runtimes is unavoidable in general purpose systems. Since in theory, most of the feasibility analysis and fine-tuning of the scheduling and mapping algorithms are done assuming complete knowledge of the runtimes of the job, such analysis might fail or produce substandard performance in a real setting. Moreover, in a real setting some of the jobs terminate abnormally due to various

reasons. Such cases are usually not accounted for in theoretical analysis. Impact of such errors in estimation on performance is potentially large and one needs mechanisms for effectively dealing with it.

Some other papers have investigated the effect of estimated runtimes on scheduling in general [MUA01] [SOB04], but to the best of our knowledge, this is the first work that studies the problem in the contexts of ARs and QoS constrained admission control. The thesis presents SEM to effectively handle errors in user-estimated runtimes.

Handling Highly Overestimated Runtimes: The analysis of the traces (presented in Section 4.3.3) collected from real systems shows that job runtimes are substantially overestimated by the users. For the cases where applications highly overestimate their runtimes, a large amount of work may be unnecessarily rejected by the resource if the feasibility analysis is done using user-estimated runtimes. Such rejections would result in low utilizations of the resources. Under these conditions a pre-scheduling policy is needed that can transform the runtimes of the jobs before passing them to the scheduler to avoid substandard performance. To the best of our knowledge, pre-scheduling policies have not been studied in the context of advance reservations. The thesis presents two novel pre-scheduling policies and demonstrates their efficacy through experimentation.

Guidelines for Runtimes-Predicting-Algorithms: Given a certain percentage error in user-estimated runtimes, the choice of overestimation/underestimation can substantially affect system performance. One needs to establish guidelines for runtimes-predicting-algorithms that can meet the goals of both the applications and resource

owners. To the best of our knowledge, no such guidelines exist for AR based scheduling. The thesis establishes a few useful guidelines for runtimes-predicting-algorithms.

2.6.1.6. Simultaneous Co-Allocation in Multiple Domains

As discussed in Section 2.4.5, many Grid applications, such as those in data mining and computational-steering, and workflows may require simultaneous allocation of multiple resources that may be independently administered and scheduled. Thus, one needs a mechanism for simultaneous co-allocation of multiple resources in multiple domains. Although GARA [FOS99b] presents the concept of co-reservation and co-allocation agents, it does not present any algorithms that will be used for the selection of resources for co-allocation and determination of a suitable time for applications at which all selected resources are available. The thesis develops algorithms that can effectively co-allocate multiple resources in multiple domains.

Chapter 3

A Framework to Achieve QoS for Applications and High System Performance in Multi-Institutional Grids

3.1. Background

As discussed in Section 2.6, the framework proposed by this thesis relies on advance reservations for achieving QoS guarantees for the applications. However, ARs are under-constrained by introducing laxity in the reservation window to provide flexibility in scheduling and hence to achieve high system performance. Section 2.6.1 discusses the challenges faced when developing a resource management framework based on ARs with laxity. This section describes the application and resource models for the framework.

3.1.1. Application Model

The thesis considers Grid applications as consisting of one or more *jobs*. Different jobs of an application may need to run on different possibly heterogeneous resources or on the same resource but at different times. Each job is allocated a single resource at a time. As an example, consider an application that first needs to perform some computing on a cluster and then transfer the results through the network to a storage facility. Such an application can be divided into three major jobs – compute job, network job and storage job. Each job consists of one or more *tasks*. Tasks represent closely related activities of a Grid application. For example, compute job in the above scenario may consist of multiple threads each of which represents one task. All tasks of the same job runs on the same

resource. However, for a shared resource model (defined in Section 2.6) different tasks are allocated different *nodes* of the resource. Nodes of a shared resource represent its different partitions. Each shared resource consists of one or more physical or logical partitions (nodes). For example, consider different workstations that belong to the same cluster. In this case, the cluster represents the shared resource and workstations its nodes. With virtual machines, even a single CPU based computing device may be allocated to more than one job each having a different share of the CPU cycles. In this case, nodes represent the smallest share of the CPU cycles.

Since different tasks of a job may need to communicate while executing, they are all scheduled for simultaneous execution on the nodes. Such a scheduling paradigm is generally referred to as *gang scheduling*. The need for gang scheduling has been widely recognized and for many multi-task applications the performance is severely affected when any part of the application is not running. Gang scheduling of tasks not only prevents communications overheads but also provides for efficient sharing of resources, responsiveness and ease of programming [SIL99].

The thesis uses an open model in which a stream of jobs arrives at the system. If a job of a Grid application needs response time guarantees or if different jobs of the Grid application are to be co-allocated, advance reservations are used. Otherwise, the job can be submitted to the resource as a best-effort on-demand request. A typical workload consists of a mix of ARs and OD requests.

Each job is specified by a number of parameters that describe the resource requirements of the job to function properly such as the kind of resource it needs, the amount of memory it requires and so on. Apart from the functional requirements, each

job is specified by three *QoS parameters* – earliest start time, estimated execution time and deadline – that describe the QoS constraints of the job. These are described next.

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

Estimated Execution Time or Estimated Runtime ($e_{E,ib}$): Estimated execution time of a job j_i on a resource b is the estimated time the job j_i takes to finish executing on the resource b. Consistent with previous work on scheduling systems involving ARs [FIG04a, SIM00, SUL04], the thesis assumes that properties of the job such as the estimated number of compute cycles in the case of a compute task or the estimated number of bytes to be transferred in the case of a network task are either known in advance or some statistics are available for their calculation. Note that most current job scheduling systems also require that the jobs specify their maximum runtime. The thesis only requires an estimated runtime and the framework presented by the thesis is tolerant of some errors in estimation. The framework has components to handle exceptions resulting from inaccuracies in estimated runtimes.

Since all tasks of a multi-task job need to be gang scheduled, for all practical purposes their runtimes can be considered equal. This is because even if one of the tasks is suspended for some reason the node it is running on is not allocated to a task of another job until all tasks of the job the task belongs to finish their execution. Hence, from the

system perspective the runtime of a multi-task job can be considered equal to the runtime of its longest task.

Note that the same job may have different estimated execution times on different resources depending on the relative *speed* of the resources (see Section 4.4.7).

Deadline (d_i): *Deadline* of a job is the absolute deadline before which the job must be finished. Once committed a resource ensures that a deadline associated with each AR is met provided the actual runtime of the job is equal to or less than its estimated runtime. If the actual runtime of a job is greater than its estimated runtime, the execution of that job may continue past its deadline.

In case of on-demand requests, the deadline to finish the job is assumed to be infinity. However, the thesis presents resource level policies that can prevent starvation of OD requests.

Size (s_{ib}): Each job j_i specifies its size s_{ib} on a resource b, as a part of its parameters. The size of the job is equal to the number of its tasks. Since for a non-shared resource model the job exclusively uses the resource, the size of the job becomes irrelevant. For shared resources, generally there is a one-to-one relationship between each task of the job and each node of the resource. Even if such a relationship does not exist, the number of tasks of a job can be translated into the required number of nodes of the resource.

3.1.2. Resource Model

The thesis considers any networked device that can be shared among other Grid users using Grid protocols to be a Grid resource. For example, a single CPU, a multiple CPU based computing device, a lightpath or a storage system. The shared and non-shared

models for using a Grid resource have been defined and discussed in Section 2.6. Section 3.1.1 explains different partitions of a shared resource known as nodes. If all the nodes of a shared resource are identical such that it does not make a difference which task of a job is allocated to which node, the shared resource is called *homogeneous shared resource*. If different tasks of a job can run only on particular nodes, the shared resource is called a *heterogeneous shared resource*. An example of a heterogeneous shared resource is a network consisting of a mesh of multiple segments. Even if all the segments of such a mesh network have identical physical characteristics, for an application that needs to transfer data from a certain point in the network to another point in the network, allocation of only a certain combination of segments can fulfill its requirements.

3.2. Framework for Resource Management with Advance Reservations

The thesis presents a framework for advance reservation based resource sharing that ensures high performance for the resource owners while meeting the QoS requirements of the resource consumers. A high-level view of the framework is presented in Figure 3.1.

Figure 3.1 shows that the framework consists of two major components: Matchmaker and Multi-Resource Coordinator (MMC) that lies at the collective layer of the Grid layered architecture and Resource Liaison and Controller (RLC) that lies at resource and fabric layers. In a typical scenario, applications submit their resource requirements along with their QoS constraints to MMC. MMC finds a suitable match for the application and RLC allocates the specific resource. Alternatively, applications can also interact directly with RLC. There is a many-to-many relationship between MMC and

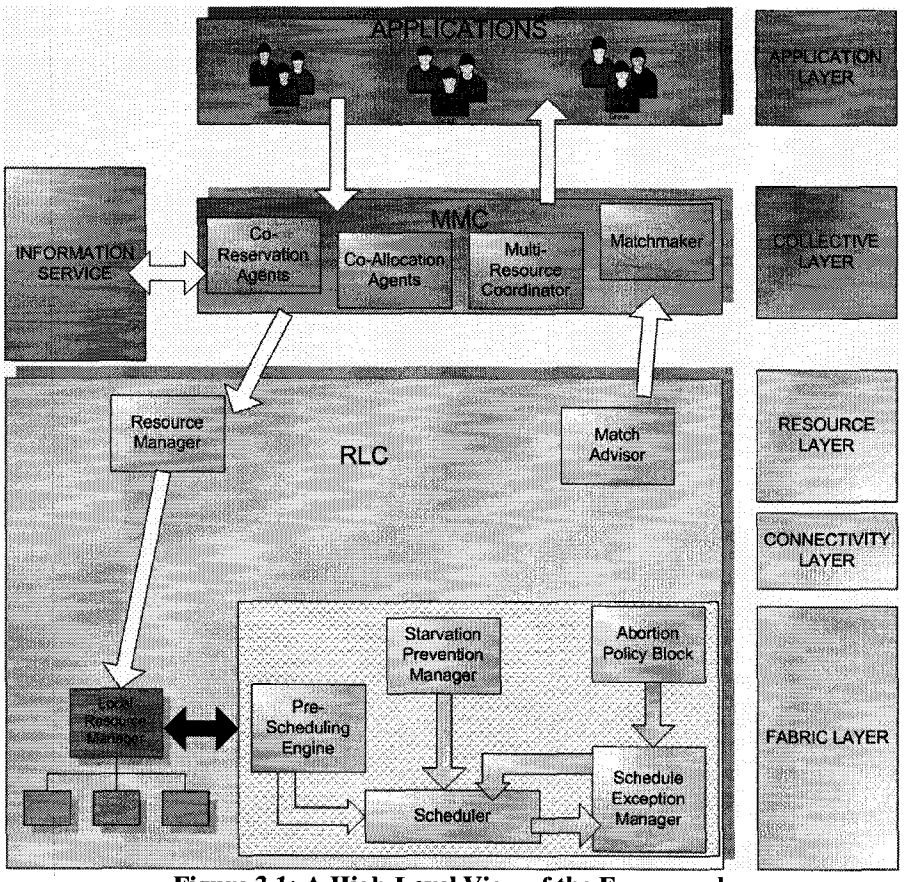


Figure 3.1: A High-Level View of the Framework

RLC showing that each MMC can act as a broker to many resources and each resource may be traded by many brokers. MMC and RLC are described in detail next.

3.3. Resource Liaison and Controller (RLC)

The function of RLC is to act as a liaison between the resource and the application or MMC, perform admission control for the resource and control the resource in such a way that each accepted AR meets its deadline. If the resource cannot meet the QoS constraints of an AR, it rejects the new request. While an application can specify the start time window of its job through its QoS constraints, RLC leaves the final scheduling decision with the local scheduler. As shown in the figure, RLC consists of seven major components which are described as follows.

3.3.1. Resource Liaison (RL)

The *Resource Liaison* component provides the functionality of resource managers such as GRAM [CZA98], and negotiates access to the resource through the local resource managers. It provides client API to remotely request resource allocation and process creation. MMC uses the services of RL to make reservations on and/or allocate the resource.

3.3.2 Scheduler (SCH)

The *Scheduler* (SCH) component schedules ARs and OD requests for a resource. At the arrival of every new job, SCH seeks to obtain a feasible schedule for the set of jobs already in schedule and the new job. If a feasible schedule is not found, the job is rejected. Otherwise, the Match Advisor component assesses the degree of fit of the job in the resource schedule and reports it to MMC. If the given resource is selected by MMC for that job, the job is added to the set of scheduled jobs. As each job finishes executing on the resource, it is removed from the set of scheduled jobs.

The performance of the resource depends largely on the efficacy of the scheduling algorithm employed by scheduler to schedule ARs and OD requests. SCH must try to achieve high utilization of the resource while ensuring that QoS constraints of each accepted AR is met. Since optimally scheduling jobs with given start times, estimated execution times and deadlines is an NP-complete problem and since large number of requests are possible in a Grid domain, scalability of the scheduler in terms of large number of requests is important to ensure reasonable scheduling times.

The problem of scheduling jobs with laxities can be divided into multiple problems corresponding to two different sharing models of the resources. The next sections formally define problem statements for each model.

3.3.2.1 Scheduling On-Demand and Advance Reservations Requests with Laxities on a Non-Shared Resource

The problem can be formally stated as:

Problem 1: *Given a non-shared resource b, a finite set of jobs* $\{j_1, j_2, ..., j_N\}$ *and sets of the earliest start times* $\{t_1, t_2, ..., t_N\}$, *estimated execution times* $\{e_{E,1b}, e_{E,2b}, ..., e_{E,Nb}\}$ *and deadlines* $\{d_1, d_2, ..., d_N\}$, *schedule jobs* $\{j_1, j_2, ..., j_N\}$ *on resource b such that each job* j_i *starts executing after its earliest time* t_i *and finishes before its deadline* d_i.

3.3.2.2. Scaling through Subset Scheduling (SSS) Algorithm

As discussed in Section 2.6.1.1, most of the algorithms presented in real-time domain for solving problems similar to Problem 1 do not scale for large number of jobs. In order to scale the algorithm for much larger number of jobs that are possible in the Grid domain, the thesis adapts the algorithms presented in the real-time domain and presents the Scaling through Subset Scheduling (SSS) algorithm. SSS is capable of finding a feasible schedule if one exits. It is also capable of studying scenarios involving both non-preemptable and preemptable jobs including the cases where overheads are associated with job preemption. SSS is presented and analyzed in detail in Chapter 5.

3.3.2.3. Scheduling On-Demand and Advance Reservations Requests with Laxities on a Homogeneous Shared Resource

For a shared resource, in addition to t_i, $e_{E,ib}$ and d_i, the size of the job s_{ib} needs to be taken into account for scheduling. The size of the job s_{ib} can be translated into the required number of nodes n_{ib} of the shared resource. The problem thus can be formally defined as follows:

Problem 2: *Given a homogeneous shared resource b with* ω *identical physical or logical nodes, a finite set of jobs* $\{j_1, j_2, ..., j_N\}$ *and sets of the earliest start times* $\{t_1, t_2, ..., t_N\}$, *estimated execution times* $\{e_{E,1b}, e_{E,2b}, ..., e_{E,Nb}\}$, *deadlines* $\{d_1, d_2, ..., d_N\}$ *and*

required number of nodes $\{n_{1b}, n_{2b}, ..., n_{Nb}\}$ *where* $n_{ib} \leq \omega$ *for all i, schedule jobs* $\{j_1, j_2, ..., j_N\}$ *on resource b such that each job* j_i *starts executing after its earliest time* t_i, *finishes before its deadline* d_i *and throughout its execution of* $e_{E,ib}$ *units of time,* n_{ib} *nodes are allocated to it.*

3.3.2.4. Grid Scheduling with Deadlines (GSD) Algorithm

Just like Problem 1, it can be shown that Problem 2 is NP-complete [GAR79]. Since it has an additional dimension compared to Problem 1, scalability of the algorithm for Problem 2 becomes even more important. The thesis presents a heuristics-based Grid Scheduling with Deadlines (GSD) algorithm for the efficient scheduling of jobs with laxities on homogeneous shared resources. GSD is a scalable algorithm and can handle non-preemptable and preemptable jobs. GSD is adaptable to various workload conditions and it can gang schedule all tasks of the jobs to allow for inter-tasks communication. It can also be adapted for independent scheduling of tasks where tasks of the jobs are not required to be gang scheduled. Mix of gang and independent scheduling is also possible with GSD. GSD has been described and analyzed in detail in Chapter 6.

3.3.2.5. Scheduling On-Demand and Advance Reservations Requests with Laxities on a Heterogeneous Shared Resource

GSD can be extended for heterogeneous shared resources. Such an extension of GSD is proposed as future research.

3.3.3. Starvation Prevention Manager (SPM)

Results in Section 5.3 will show that when the proportion of advance reservations requests is increased, the average response time of on-demand requests increases significantly. A very high average response time for OD requests will encourage all users to submit their tasks as ARs which would further increase the proportion of advance

reservations. A higher proportion of advance reservation would not only result in lower utilization of the resource but it would also ultimately increase the average response time of all requests in the system. The goal of the Starvation Prevention Manager (SPM) is to prevent these situations resulting from the potential starvation of on-demand requests. For this purpose, different policies can be implemented within SPM. The thesis presents one simple policy called Deadlines Policy which is discussed next. More sophisticated policies are beyond the scope of this thesis.

3.3.3.1. Deadlines Policy (DP)

Deadlines Policy associates a *large* deadline with every on-demand request before passing the job to the scheduler. This places a limit on the maximum response time for OD requests. The value of the deadline depends on number of system parameters such as the mean execution time of the jobs and proportion of advance reservations.

3.3.4. Schedule Exceptions Manager (SEM)

As discussed in Section 2.6.1.5, inaccuracies in user-estimated runtimes and abnormal terminations of jobs that are not usually accounted for in theoretical analysis are unavoidable in a real-world setting. Schedule Exceptions Manager monitors the resource schedule and deals with exceptions resulting from abnormal terminations of jobs as well as inaccuracies in user-estimated runtimes. Results in Section 7.5.1 will show that abnormal terminations of the jobs combined with inaccuracies in user-estimated runtimes can adversely affect the efficiency of the system. The goal of SEM is thus to prevent such performance degradations by adapting the resource schedule to the changing conditions. SEM is discussed and analyzed in detail in Chapter 7.

3.3.5. Abortion Policy Block (APB)

Abortion Policy Block (APB) governs policies to determine suitable actions when an underestimated job does not finish at its expected completion time. APB is consulted by SEM to decide the fate of the underestimated jobs. Different policies can be used with APB to suit the needs of the systems. APB is discussed and analyzed in detail in Chapter 7.

3.3.6. Pre-Scheduling Engine

The analysis (to be presented in Section 4.3.3) on inaccuracies in user-estimated runtimes using logs collected for IBM SP2 machines shows that runtimes of the jobs are substantially overestimated by the users. This combined with abnormal terminations of the jobs, results in an extremely large mean error in user-estimated runtimes with a positive sign showing over-estimation. Since the feasibility analysis at the time of admission of the job is performed using user-estimated runtimes, extremely large overestimation in runtimes results in a high degree of unnecessary rejections of jobs. The goal of Pre-Scheduling Engine (PE) is thus to prevent low utilizations of the resources resulting from unnecessary job rejections. For this purpose, user-estimated runtimes are transformed by PE before being passed to the Scheduler. PE is discussed in detail in Chapter 7 along with two pre-scheduling policies.

Note that PE is invoked at the arrival of every new job to suitably transform the runtime of the new job. Once the job has been admitted, it is the SEM component of the framework that manages the runtime exceptions and the inaccurately estimated job runtimes.

3.3.7. Match Advisor (MA)

The function of *Match Advisor* is to asses the degree of fitness of a particular job in the resource schedule given the QoS parameters of the job and the current state (schedule) of the resource and report it to MMC. The assessment of fitness depends on the matchmaking algorithm used by the system. MA is invoked at the arrival of a new job, to decide whether the new job is a good fit for a given resource.

3.4. Matchmaker and Multi-Resource Coordinator (MMC)

MMC acts as a broker between the Grid applications and Grid resources. Typically, a Grid application submits its requirements to MMC and MMC finds matching resource(s) for the application. It takes into account interests of both the applications and the resources during this matchmaking. If MMC cannot find any resource that can meet the application's QoS constraints, the request is rejected. For the applications that require more than one resource simultaneously, MMC can help coordinate multiple resources. MMC itself consists of four components which are described next.

3.4.1 Co-Reservation Agents (CRA)

Just as in GARA [FOS99b], the *Co-reservation Agents* discover available resources that can satisfy an application's functional requirements with the help of the Grid Information Service. The agents then invoke the services of the *Matchmaker* or the *Multiple-Resource Coordinator* components to choose the resource(s) to allocate the job. If the number of resources that meet an application's functional requirements is too high, only a subset of resources can be passed to the Matchmaker or Multiple-Resource Coordinator. The thesis calls the resources that are passed to the Matchmaker or Multiple-Resource Coordinator to do the selection from as the *candidate set.*

3.4.2 Matchmaker (MM)

The function of the matchmaker is to select a suitable resource from the candidate set in such a way that not only application's constraints are met but also overall system performance is maximized. Overall system performance can be measured in terms of either the amount of work rejected or system utilization. In order to meet its goal, the matchmaker relies on a matchmaking algorithm. The efficiency of the system depends on the efficacy of the matchmaking algorithm. The thesis investigates several matchmaking algorithms and presents a novel algorithm Minimum Laxity Impact (MLI) for matchmaking that outperforms all other algorithms investigated in almost every respect for a wide range of workload parameters.

Several matchmaking algorithms are presented in Chapter 8 along with a detailed discussion and analysis of MLI. Most of these algorithms depend on the reports from the *Match Advisors* within RLC of each resource for doing the selection.

3.4.3. Multiple-Resource Coordinator (MRC)

Matchmaker can be effectively used for scenarios where an application requires a single resource or where different jobs of an application are independent to each other. For the latter case, each job can be treated as a separate application and the matchmaker can find a suitable resource for each. In situations where an application requires simultaneous allocation of multiple resources from possibly different domains, Multiple-Resource Coordinator is required. MRC ensures that different jobs of the Grid application are scheduled to execute at exactly the same time on different resources in possibly different domains. MRC relies on co-allocation algorithms that can a) determine a suitable scheduled-time for the application and b) suitably select the resources that can start the application at the determined scheduled-time. The efficiency of the system

depends on the co-allocation algorithm used. In Chapter 9, the thesis presents and analyzes co-allocation algorithms that seek to minimize work rejected by the system while meeting the QoS and co-allocation constraints of the applications.

3.4.4. Co-Allocation Agents (CAA)

At the execution time of the application, the application uses *Co-allocation Agents* to submit the jobs to the selected resources. CAA also provides means to monitor the progress of the jobs on the resources.

3.5. Comparison of the Framework with Other Grid Technologies

To the best of our knowledge, the framework presented by this thesis provides the first complete framework for resource sharing with advance reservations. The most appealing feature of the framework is its ability to take into account interests of both the Grid applications and Grid resources. This is important in a multi-institutional setting with multiple stakeholders. Unlike most of other works that focus on only one aspect of resource sharing in Grids, the framework presents components for each of the fabric, resource and collective layer of the Grid layered architecture. The framework aligns well with other Grid technologies. For example, as long as the local resource managers have been bound to scheduling algorithms presented in this thesis for advance reservations, any remote resource manager can be used as a Resource Liaison. For a computational application, GRAM can be used as an RL while for data transfer on a dynamic optical network, DRAC [NOR04] can be used as an RL. Similarly, GARA can be interfaced with the Matchmaker or MRC to act as an MMC.

The framework is not limited to any particular kind of Grid applications or resources. For example, it can be applied for a computational job requesting resources on

a cluster, a data transfer application requesting lightpaths or a job that requires both computational and network resources.

Different components of the framework such as SEM and PE can by dynamically plugged in and plugged out as needed by the system. Similarly, RLC can be adapted to cater to the needs of the resource through the adaptable GSD scheduling algorithm and dynamic selection of polices for SEM and PE. The thesis argues that this dynamic configurability of the modern systems, which may comprise thousands of resources and consumers, is extremely important for achieving high efficiency.

Comparison of the framework by the thesis with other Grid technologies shows that different components of the framework enhance the functionality of different Grid technologies. For example, GARA that introduced advance reservations presents the concepts of co-reservation and co-allocation agents. 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. The framework presented by this thesis specifically deals with the above-mentioned issues and in this context extends the functionality of GARA. Similarly, CSF [PLA05c] presents an open source framework for implementing a Grid meta-scheduler that can dispatch jobs to underlying resource managers. It however, does not present an efficient meta-scheduling algorithm for AR based scenarios. The MLI algorithm presented in this thesis can be used as an efficient meta-scheduling algorithm within CSF.

Applicability of algorithms such as SSS, GSD and MLI and of components such as SEM and APB is not limited to Grids. The thesis argues that with proper adaptations they can be very well used in other multi-institutional resource sharing settings.

Different components of the framework are being integrated into the well-known Globus Toolkit [GLO05c] and CSF [PLA05c] with the aim of making the framework available for users around the world. The integration with Globus Toolkit has already been used to build QoS enabled Grid compute clusters [FAR06c].

3.6. Integration with Globus Toolkit

This section discusses the integration of the GSD scheduler with the WS-GRAM component of Globus Toolkit [FAR06c]. The system design for this integration was undertaken by the author of the thesis while part of the implementation was performed by an undergraduate summer scholar. The prototype resulting from this integration is tested on the epsilon cluster at the Department of Systems and Computer Engineering in Carleton University. The epsilon cluster consists of twenty 3.0 GHz Pentium 4 machines each having 1 GB RAM and running Fedora Linux.

WS-GRAM component of the Globus Toolkit has a simple built in scheduler that immediately dispatches a time-sharing job on arrival. A process corresponding to the job is spawned using the `fork()` system call in Unix. However, to manage large compute clusters and/or to meet more specialized scheduling needs (such as a need to schedule ARs), a specialized local scheduler needs to be interfaced with WS-GRAM. WS-GRAM provides the concept of *Scheduler Adapters* and *Schedule Event Generator (SEG)* that can be used for interfacing local schedulers with it. The schematics of running a specialized local scheduler with WS-GRAM are shown in Figure 3.2 which is adapted

from [GLO05m]. In this figure, a client submits its job request to WS-GRAM and communicates its requirements in Resource Specification Language (RSL) [GLO05c]. RSL is an XML schema based standard for specifying job requests. WS-GRAM services shown in Figure 3.2 invoke the Reliable File Transfer (RFT) service of Globus Toolkit to aid in the staging of job related files before and after job computations. GridFTP is used to access data elements in a remote storage. RFT acts as a client to GridFTP servers to manage data transfer between a remote storage and the file system of the local computing resource. Delegation is a process by which a client can authorize Grid services to act in its name. The Delegation service of Globus Toolkit shown in Figure 3.2 allows the clients to delegate their credentials for use by WS-GRAM and RFT.

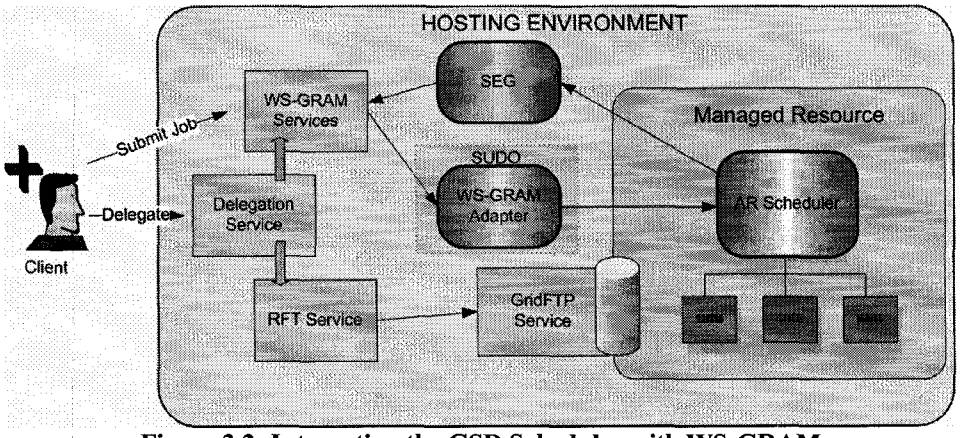


Figure 3.2: Integrating the GSD Scheduler with WS-GRAM

As can be seen in Figure 3.2, the interface between WS-GRAM Services and an AR scheduler consists of two modules – Schedule Event Generator (SEG) and Scheduler Adapter. A high level description of the modules is presented first. Details of how the modules are incorporated in the prototype are provided in the following paragraph. The SEG module is to be written in C while Scheduler Adapter is to be implemented in Perl. Schedule Event Generator is responsible for monitoring a job's state and generating job state change events. On the other hand, Scheduler Adapter is the component that takes the

job description and converts it into a form that the local scheduler can understand. Note that Scheduler Adapter runs under a standard Unix utility called `sudo` [SUD06]. The `sudo` utility allows Scheduler Adapter to perform system specific operations, required for initializing and running jobs, in the user account context without requiring Scheduler Adapter to have super-user privileges. Scheduler Adapter usually implements a number of methods such as those for job submission, job cancellation and for polling job state. However, all Scheduler Adapters must implement a *submit* method which is called whenever a new job is submitted. This method was implemented in this prototype.

A high-level view of the integration of the GSD scheduler with Globus Toolkit for scheduling on the epsilon cluster is shown in Figure 3.3. In the figure, Scheduler Adapter is implemented as a Perl module, `ar_sched.pm`. This module is integrated with the GSD scheduler with the help of a component called *AR Server.* AR Server is responsible for dispatching jobs to the individual nodes in the epsilon cluster, monitoring job states and freeing up resources on the nodes after either the job has successfully completed or it has executed on the node for its estimated runtime. Since it is virtually impossible to accurately predict the runtime of a job, Schedule Exceptions Manager (SEM), effectively handles exceptions resulting from inaccurately estimated runtimes and abnormal terminations of jobs. SEM is currently being integrated with AR Server.

Whenever a new job is submitted, the GRAM services extract the original job specifications in RSL and make them available to `ar_sched.pm` in the form of a *Job Description Object.* For jobs requiring advance reservations, parameters of jobs such as earliest start time, job size, estimated execution time and deadline are passed using the `arguments` element defined in RSL. The Perl module extracts all the job parameters

from Job Description Object and passes it to AR Server. AR server is implemented as a multi-threaded process which upon receiving a job request, spawns a new thread which consults the GSD scheduler to determine whether the new job can be accommodated without violating the deadline constraints of the new job and the jobs previously committed. If the job can not be accommodated, it is rejected. Otherwise, the job is accepted and the resource schedule is updated. At the scheduled time of a job as determined by the resource schedule, AR Server dispatches the job to the nodes on which it has been scheduled to run. Similarly, at the expected completion time of the job, AR Server frees up resources on the nodes running the job. After the integration of SEM with AR Server, AR Server will be able to adjust the resource schedule whenever a job finishes earlier than its expected completion time or a job terminates abnormally. SEM also governs policies (presented in Section 7.3) regarding jobs that do not complete within their estimated runtime.

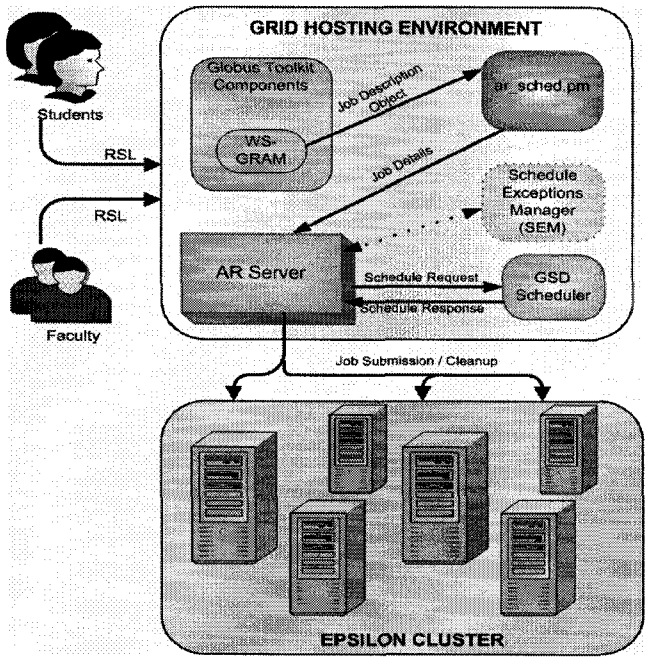


Figure 3.3: High-Level Architecture for Interfacing the GSD Scheduler with Globus Toolkit

The architecture described in this section, for the integration of the GSD scheduler with Globus Toolkit, is not specific for the epsilon cluster. Any compute cluster can be made advance reservations enabled by suitably setting the parameters (such as the number of nodes) in the AR Server's configuration file.

Chapter 4

Simulation Models

4.1. Experimental Setup

Since the research models expensive and not easily accessible Grid resources such as supercomputers and lightpaths, a prototype implementation for experimental study was not possible. The research is thus based on simulation. A detailed simulator was developed to model different aspects of Grid resources, Grid applications and their interactions. The experimental setup used for a performance study of Resource Liaison and Controller (RLC), the SSS algorithm and the GSD algorithm is described in Section 4.1.1 while that for a performance study of Matchmaker and Multi-Resource Coordinator and matchmaking algorithms including MLI is discussed in Section 4.1.2. Note that in general matchmaking is performed first, but for this section we study RLC first.

4.1.1. Experimental Setup for Performance Study of Resource Liaison and Controller

To perform the performance study at the RLC level, the experimental setup assumes that an application-to-resource mapping has already been performed using a suitable matchmaking algorithm. An open model is used where a stream of jobs arrives at RLC. A new job is accommodated in the resource only if the QoS constraints of all jobs previously accepted as well as that of the new job can be met. Otherwise, the job is rejected.

Since the research is concerned with two different types of resources, non-shared and shared, two different models of resources and applications are used to study the

performance behavior of each. For the non-shared model, since each job requests the exclusive use of the resource, the number of tasks of a job is irrelevant. Any resource such as a single CPU or a multiple CPU based computing device, a storage system or a system consisting of both CPU and storage devices that can satisfy the job requirements and can provide exclusive access to a job can be considered as a non-shared resource. SSS is used for scheduling jobs on non-shared resources.

For the shared resource model, models of 64-node IBM SP2 supercomputers are used as the Grid resources. IBM SP2 is one of the most widely used parallel supercomputer and has been deployed in many established research institutes such as Cornell Theory Center, Swedish Royal Institute of Technology and San Diego Supercomputer Center. Another reason for modeling SP2 as the Grid resource is that detailed high quality traces of real parallel workloads for SP2 machines deployed at such institutes are publicly available. With the help of those traces, performance of various strategies and algorithms on the modeled resource can be studied by subjecting the modeled resource to a workload based on the traces. This increases the confidence in the results obtained.

Different jobs can run on an SP2 concurrently by running tasks of different jobs on different nodes. The Grid applications are modeled as one or more parallel non-preemptable rigid jobs requesting any where from 1 to 64 nodes depending on their number of tasks. For rigid jobs, the size of a job is specified by the user and it remains the same for the entire execution of the job [DAN03]. As discussed in Section 3.1.1, different tasks of the parallel jobs may need to communicate during execution. Thus, all tasks of a

job are always gang scheduled on the SP2 cluster. GSD is used as a scheduling algorithm on the SP2 cluster.

SEM described in Section 3.3.4 monitors and adapts the resource schedule in the presence of uncertain runtimes and PE (described in Section 3.3.6) transforms the runtimes of the jobs before passing them to the scheduler. SEM and PE, however, are used in only those set of experiments where their use has been mentioned explicitly. SPM tries to prevent starvation of OD requests. For the non-shared resource model, some experiments are conducted without SPM to study the system behavior in its absence. For all other experiments SPM uses the DP policy discussed in Section 3.3.3.1 to prevent starvation of OD requests.

4.1.2. Experimental Setup for Performance Study of Matchmaker and Multi-Resource Coordinator

The simulated setup for the performance study at the MMC level is shown in Figure 4.1. In the figure, jobs arrive at the MMC and submit their requirements along with their QoS constraints to the MMC. For applications that do not need simultaneous co-allocation in multiple domains, the Matchmaker in MMC uses matchmaking algorithms presented in Chapter 8 to find a suitable match. To study simultaneous co-allocation of different resources in multiple domains, the Multi-Resource Coordinator (MRC) uses co-allocation algorithms presented in Chapter 9 to map the incoming requests to the underlying resources.

The underlying Grid resources may have different relative speeds. This heterogeneity in the system is defined and discussed in detail in Section 4.4.7.

The underlying resources are considered to be homogenous shared resources. Hence, the application and resource models for shared resources discussed in Section

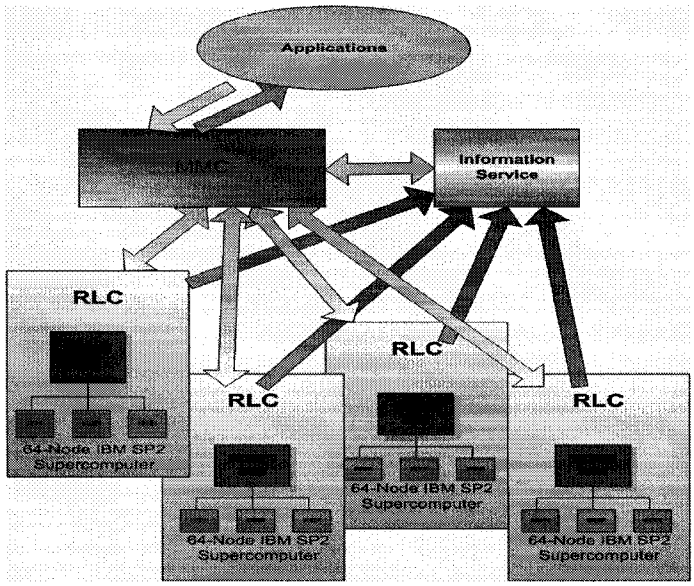


Figure 4.1: Simulated Setup for the Performance Study at the MMC level

4.1.1 are used. Note that since different resources might have different relative speeds, a given job may have different execution times on different resources as discussed in Section 4.4.7.

4.2. Performance Metrics

The performance metrics used in this thesis are described in this section.

4.2.1. Percentage of Work Rejected (W_R)

W_R measures the percentage of the work rejected by RLC (for experiments conducted to study the performance behavior of RLC) or MMC (for experiments conducted to study the performance behavior of MMC) because it cannot be accommodated without violating the QoS constraints of some of the jobs. Total work of a job is defined as the sum of the actual runtimes of all of its tasks. Since for the shared resource 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 actual runtime. W_R is the proportion of the sum of

the total work of all the jobs rejected and the sum of the total work of all the jobs. W_R can be calculated as:

W_R = *100 * (Sum of the Total Work of the Jobs Rejected by the System/ Sum of the Total Work of All the Jobs Submitted)* **(4.1)**

This metric is not used for non-shared resource models for which only the probability of blocking is considered.

4.2.2. Probability of Blocking (P_b)

P_b is the probability that an RLC (for experiments conducted to study the performance behavior of RLC) or an MMC (for experiments conducted to study the performance behavior of MMC) would reject a new job because it cannot be accommodated without violating the QoS constraints of some of the jobs. It can be calculated as:

P_b = *Total Number of Requests Rejected / Total Number of Requests* **(4.2)**

4.2.3. 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 Resource / (Total Observation Time * Total Number of Nodes in the Resource)* **(4.3)**

For the non-shared resource model, the number of nodes is considered to be equal to 1. For experiments with MMC involving resources of the same relative speeds, U represents average utilization of all the resources in the system. For experiments with MMC involving resources having different relative speeds, utilization of individual resources is reported.

4.2.4. Mean Response Time of Advance Reservation Requests (R_{AR})

R_{AR} is the mean response time for advance reservations. The response time of an AR is the difference between the time of completion of the job associated with the request and the earliest start time of the job.

4.2.5. Mean Response Time of On-Demand Requests (R_{OD})

R_{OD} is the mean response time for on-demand requests. The response time of an OD is the difference between the time of completion of the job and the time the job was submitted.

4.2.6. Percentage of Work Aborted (W_A)

For the experiments involving uncertain runtimes, jobs that do not complete within their estimated runtime may be aborted by the resource. W_A measures the proportion of the work executed on the resource but aborted before its successful completion. It is calculated as:

W_A = *100 *(Sum of the Units of Time of All the Nodes in the Resource When They were Busy Executing The Jobs Which were Ultimately Aborted / Total Work of All the Jobs Submitted to the Resource)* **(4.4)**

For experiments with MMC, W_A represents average percentage of work aborted by all the resources in the system.

4.2.7. Probability of Aborting (P_A)

For the experiments involving uncertain runtimes, P_A measures the probability that a job submitted to the resource would be aborted before its successful completion. It is calculated as follows:

P_A = *Total Number of Jobs Aborted / Total Number of Jobs Submitted* **(4.5)**

4.2.8. Useful Utilization (UU)

For the experiments involving uncertain runtimes, UU of a resource is the fraction of the total time the resource is busy executing those jobs that are completed successfully. As the jobs that are aborted prematurely are not considered to contribute to the useful work done by the resource, for the scenarios involving inaccuracies in user-estimated runtimes, UU is the most important measure of the efficiency of the system. UU can be calculated as:

$$UU = \frac{\textit{Sum of the Units of Time of All the Nodes in the Resource When They were Busy Executing The Jobs Which were Completed Successfully}}{\textit{(Total Observation Time * Total Number of Nodes in the Resource)}} \quad \textbf{(4.6)}$$

For experiments with MMC involving resources having the same relative speed, UU represents average useful utilization of all the resources in the system. For other experiments with MMC, UU is not used as a performance metric.

4.2.9. Fairness (Q_R)

Fairness measures the ability of the system to treat small and large jobs equally. Q_R, a measure of fairness, is defined as the ratio of average total work of all jobs rejected and the average total work of all jobs submitted. A value of 1 for Q_R means that the algorithm treats small and large jobs equally at the time of admission. A value greater than 1 means that the system is discriminating against large jobs because the average total work for rejected jobs is higher than the average total work for all jobs submitted. Similarly, a value less than 1 implies that the system is discriminating against small jobs. Q_R can be shown to be equal to $W_R/(100*P_b)$.

4.2.10. Parameters for the Performance of the SSS Algorithm

For evaluating the performance of the SSS algorithm, the total number of child solution-nodes $\check{N}$ produced to schedule a given number of jobs is determined. The

maximum number of solution-nodes in memory at any point $\check{N}_{max}$ is also determined, as this is one of the numbers that grow with the increase in number of jobs that need to be scheduled.

Table 4.1 shows the summary of performance metrics used in the thesis.

Table 4.1: Summary of Performance Metrics

Performance Metric	Symbol
Percentage of Work Rejected	W_R
Probability of Blocking	P_b
Utilization	U
Mean Response Time of Advance Reservation Requests	R_{AR}
Mean Response Time of On-Demand Requests	R_{OD}
Percentage of Work Aborted	W_A
Probability of Aborting	P_A
Useful Utilization	UU
Fairness	Q_R
Total Number of Solution-Nodes Produced by SSS to Schedule a Given Number of Jobs	$\check{N}$
Maximum Number of Solution-Nodes in Memory	$\check{N}_{max}$

4.3. Workload Models

Two different workload models are used by this research – one for each of the two resource usage models. They are described next.

4.3.1. Workload Model for Non-Shared Resources

For the non-shared resource model, this research has used a uniform distribution for the runtimes of the jobs. A uniform distribution for modeling runtimes of jobs has been used in other researches on advance reservation based scheduling (see [SUL04], for example). The preliminary experiments demonstrated that for uniformly distributed runtimes of the jobs, the characteristics of the results (to be presented in Chapter 5) obtained is independent of the mean as long as the ratio of the largest and the smallest runtime in the distribution remains the same. This research models runtimes of the jobs 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 as the OMNInet

optical network such a distribution of runtimes corresponds to data transfer requests of sizes between 10GByte and 100GByte [LAVir, FIG04b]. For compute jobs, such a distribution of execution times corresponds to compute jobs 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 runtimes of the jobs, the research has also used a bi-phase hyper-exponential distribution with a co-efficient of variation of 2. The mean of the hyper-exponentially distributed execution time E_E is chosen to be equal to 50 minutes which is equal to the mean E_E of the uniform distribution.

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, an intermediate mean arrival rate λ of 0.014 requests per minute is used. The arrival process followed a Poisson distribution which has been used by other researchers as well (See [KAP07], for example). 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 [SMI00].

Note that for non-shared resources, the size of the job, which is defined as the number of tasks of a job, is not important in the context of scheduling and matchmaking.

4.3.2. Workload Model for Shared Resources

For the shared resources, the research generated jobs using a synthetic workload model for rigid jobs proposed by Lublin and Feitelson [LUB03]. The workload model is

based on the traces collected from San Diego Supercomputer Center Intel Paragon Machine, a CM-5 machine at Las Alamos National Lab and IBM SP2 machine at the Swedish Royal Institute of Technology (KTH) for long period of times. Since the shared Grid resources modeled are IBM SP2 machines, this research additionally analyzed traces collected from SP2 at San Diego Supercomputer Center (SDSC) and SP2 at Cornell Theory Center (CTC) to calculate the parameters for the workload model. The logs were cleaned before analysis by removing, for example, jobs that terminated abnormally.

The workload model in [LUB03] models the size of the jobs according to the diagram shown in Figure 4.2. In this figure, with probability p_1, the job is a serial job. Otherwise, the size of the job is selected from a two-stage uniform distribution. A two-stage uniform distribution is the generalization of a uniform distribution with four parameters l, p_u, m and h. The cumulative distribution function for the two-stage uniform distribution consists of two straight lines which pass through three points (l, 0), (m, p_u) and (h, 1). If the job is not a serial job, then with probability p_2 the size of the job is a power of 2. Since the algorithm calculates the size of the job in base-2 logarithm, the *logsize* of the job is converted to actual size of the job and rounded up to the nearest integer. As in [LUB03], the thesis uses p_1 equal to 0.24 and p_2 equal to 0.75. Table 4.2 shows the parameter values for the two-stage uniform distribution for the SP2 logs and model of job sizes.

The workload model in [LUB03] models the natural logarithm of the runtimes of the jobs using a two-phase hyper-gamma distribution. A two-phase hyper-gamma distribution is a combination of two gamma distributions and has a total of five parameters α_1, β_1, α_2, β_2 and p_g. Here α_1 and β_1 are the parameters for the first gamma

distribution, α_2 and β_2 are the parameters for the second gamma distribution and p_g is the probability of selecting the first gamma distribution during runtime generation. As in [LUB03], this research takes the natural logarithm of the runtimes of the jobs and then calculates the parameters of the resulting hyper-gamma distribution. The natural logarithm of the runtime of a job is called *log-runtime*. The log-runtime of the job is converted to the actual runtime during the simulation. Table 4.3 shows the parameter values for the hyper-gamma distribution for the model of runtimes of the jobs.

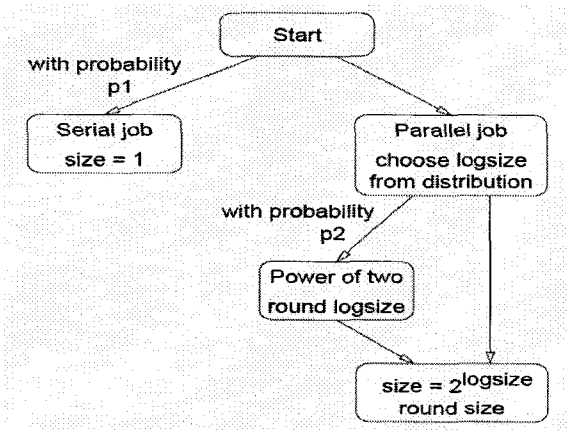


Figure 4.2: Algorithm for Modeling the Size of the Job [LUB03]

Table 4.2: Parameter Values for the SP2 Logs and Model of Job Sizes

	Total Number of Nodes	Maximum Number of Nodes Requested by a Job (in base-2 log) (max)	l (in base-2 log)	m (in base-2 log)	h (in base-2 log)	p_u (in base-2 log)
KTH	100	6.64	0.70	4.00	6.64	0.84
CTC	512	8.39	0.75	5.5	8.39	0.94
SDSC	128	7.00	0.80	4.5	7.00	0.86
Model	--	--	0.75	(max-2.5)	(max)	0.88

Table 4.3: Parameter Values for the Runtimes of the Jobs for the SP2 Logs

	Minimum Runtime (seconds)	Maximum Runtime (seconds)	Mean Runtime (seconds)	Co-Efficient of Variation C_V	α_1	β_1	α_2	β_2	p_g
KTH	1	213994	6146.7	2.36	5.76	0.82	159.5	0.05	0.61
CTC	38	64834	9906.4	1.67	4.78	2.8	114.1	0.04	0.85
SDSC	23	118561	6879.7	2.08	2.79	5	122.1	0.03	0.87

The analysis of the traces shows that there is a correlation between the size of a job and its runtime. The workload model in [LUB03] models this correlation by correlating parameter p_g of the hyper-gamma distribution to the size of a job. In their model, the size of a job is calculated first and then p_g is selected depending on the calculated size of the job. The details of this correlation are available in [LUB03]. The same technique is used in this research.

Using the workload model just described, the research generated jobs for 64-node SP2 machines. The resulting mean of the runtimes of the jobs turned out to be approximately 2200 seconds and the mean of the size of the jobs turned out to be approximately 8.4 nodes.

The thesis has used an open model in which a stream of jobs arrives on the system. As in [LUB03], the arrival process is modeled with a Gamma distribution with the same parameters as given in [LUB03]. If for a given arrival rate all the jobs submitted to the system are accepted, the utilization of the system is called *the maximum possible utilization* at that arrival rate. Note that maximum possible utilization may not be achievable due to the deadline constraints of the jobs.

The arrival time of the jobs is multiplied with a constant number to generate different maximum possible utilizations that range from medium (50%) to high (125%) values. For most of the experiments, arrival rate is selected in such a way that the maximum possible utilization of each of the clusters is 100%.

4.3.3. Models for Error in User-Estimated Runtimes

For studying the impact of inaccurate user-estimated runtimes on system performance, this research uses two different models of error in user-estimated runtimes.

The first model uses a standard normal distribution while the second model is based on traces collected from SP2 machines. The first model refers to the case where the runtimes are estimated using a runtime-prediction-algorithm [SMI03] and hence the error is relatively small. For the second model, traces collected from SP2 systems at Cornell Theory Center, SP2 systems at San Diego Supercomputer Center and SP2 systems at Swedish Royal Institute of Technology are analyzed. The analysis of these traces shows that the mean error in user-estimated runtimes is quite large.

For the first model, a standard normal distribution (SND) is used. Normal distribution has been the central assumption of the mathematical theory of errors and is involved in statistical model fitting for finding out the goodness of fit [WIK05]. It is hence a good choice for modeling the error in runtimes when the runtimes are estimated by a runtime-prediction-algorithm. It can aptly model both underestimation and overestimation while providing the ability to efficiently control the mean percentage error. The graph of the probability density function (pdf) of the SND is truncated on x-axis at two points r and –r such that 99.9% of the area of the pdf curve lied between these two points. The thesis defines percentage error in runtime ŋ for a given job as

$$\eta = 100 * (\text{Estimated Runtime} - \text{Actual Runtime}) / \text{Actual Runtime} \quad \textbf{(4.7)}$$

Thus, a positive sign of ŋ shows that the runtime was overestimated while a negative sign shows underestimation.

For a given set of experiments, a *band of error B* is defined. *B is defined as the difference between the maximum and minimum percentage error in user-estimated runtimes.* For each job, the research generates percentage error using the following formula:

$$\text{Generated } \eta \text{ for a given job} = [\text{Random Number from SND} * (B / 2r)] + \check{a} \quad \textbf{(4.8)}$$

where ă is the translation co-efficient. Note that the first term in Equation 4.8 scales the random numbers from SND within the band B. ă is used to translate the resultant curve towards over or under estimation. For example, if the value of ă is greater than B/2, then all the jobs in the system overestimate their runtimes.

For the second model, the thesis analyzes traces collected for SP2 machines. The analysis of these traces shows that for almost 98% of the jobs, ŋ has a positive sign showing high percentage of jobs with overestimated runtimes and jobs that terminate abnormally. The degree of overestimation depends on the actual runtime of the job with small jobs having higher percentage error ŋ. For some of the jobs ŋ was as high as 25000% [FAR06a].

For modeling error in overestimated jobs, the research divides the jobs into different categories depending on the value of 10-base logarithm of their actual runtimes. Thus, all jobs for which the logarithms of their actual runtimes are between 1 and 2 are in one category, those with that between 2 and 3 are in the second category and so on. Model for percentage error in each category was determined separately by using curve fitting [FAR06a]. The analysis shows that for most of these categories, the percentage error can be modeled using a two-phase hyper-exponential distribution. A two phase hyper-exponential distribution is a combination of two exponential distributions and can be described with parameters μ_1, μ_2 and p_h. Here μ_1 and μ_2 represents the parameters for the first and the second exponential distribution respectively and p_h represents the probability of selecting the first exponential distribution. The parameters of the two-phase hyper-exponential distribution varied from one category to another. The values of the parameters for errors in overestimated jobs are shown in Table 4.4. Additionally, each

category has a certain percentage of jobs whose actual runtimes are the result of abnormal terminations of some larger jobs. The analysis of the traces shows that for such jobs, the percentage error can be modeled with a uniform distribution. The parameters l_1 and h_1 for the uniform distribution for each category are also shown in Table 4.4.

The analysis of the traces shows that almost 2% of jobs underestimate their runtimes. The analysis shows that for such jobs, the percentage error can be modeled using a uniform distribution with minimum value of η at approximately -82% and the maximum value at 0%.

4.4. Additional Workload and System Parameters

In addition to the workload parameters discussed in Section 4.3, the following workload and system parameters are considered in this research.

Table 4.4: Parameter Values for the Model of Percentage Error in Overestimation

Range of Base-10 Logarithm of Runtime	Distribution (D) for Percentage Error	Parameters for Distribution D	Probability of Abnormal Termination	Parameters for Uniform Distribution
0 – 2	Hyper-Exponential	$\mu_1 = 0.00082$ $\mu_2 = 0.00025$ $p_h = 0.9$	0.21	$l_1 = 10000\%$ $h_1 = 25000\%$
2 – 3	Hyper-Exponential	$\mu_1 = 0.00128$ $\mu_2 = 0.00024$ $p_h = 0.9$	0.09	$l_1 = 10000\%$ $h_1 = 20000\%$
3 – 4	Hyper-Exponential	$\mu_1 = 0.00300$ $\mu_2 = 0.00137$ $p_h = 0.9$	0.03	$l_1 = 3000\%$ $h_1 = 10000\%$
4 – 5	Exponential	$\mu = 0.00550$	0.01	$l_1 = 400\%$ $h_1 = 600\%$

4.4.1. Proportion of Advance Reservations (PAR)

PAR is the proportion of advance reservation requests in the total number of requests. System performance has been studied for different values of PAR by varying it between 0, where all requests are on-demand, and 1, where all requests are advance reservations.

ARs reserve the resource for some time in future. In the experiments, ARs can request to reserve the resource for any time between the current system time and the next 12 hours. This mean time T_S between the arrival of an AR and its start time is modeled with a uniform distribution. Such a distribution is also used in other studies such as [SMI00]. The results show that if mean T_S is changed, it has a negligible effect on the overall system performance.

4.4.2. Mean Percentage Laxity (L)

The thesis defines *percentage laxity* of an AR as 100 multiplied by the ratio of the laxity of the AR and its actual runtime. A uniform distribution for the laxity of ARs is used with the lowest value of the distribution fixed at zero. Such a distribution is found be effective for investigating the relative performance of the resource management strategies. To the best of our knowledge, no data for the distribution of laxities in Grid workloads is available in literature. This research varies mean percentage laxity L for a wide range of values between 0% and 1000%.

Theoretically, OD requests have no deadlines and hence have infinite laxities. However, as discussed in Section 3.3.3, to prevent starvation of OD requests, SPM associates a large deadline with OD requests. For the non-shared resource model, deadline for an OD j_i is set to $t_i + 6*e_{E,ib}$. For the shared resource model, a deadline equivalent to 2 days from the time of submission is associated with each OD. Such a deadline assures that AR requests are given priority over OD requests by the scheduler and yet OD requests are not starved.

4.4.3. Job Preemption

A range of proportion of preemptable jobs 0 to 1 has been experimented with. This corresponds to workloads having only non-preemptable jobs (NP), only preemptable jobs (P) and both non-preemptable and preemptable jobs. Cases where overheads are associated with job preemption and its resumption are also studied.

4.4.4. Total number of Resources in the Candidate set (N)

For the experiments with MMC, the effect of the size of the candidate set (N) on system performance is studied by varying the total number of resources in the system. To keep the comparison fair, the amount of total work is adjusted in such a way that maximum possible utilization of resources remains the same for all values of N. This is achieved by changing the arrival rate of the jobs. The value of N is varied by a factor of 3: from 2 to 6. The relative performance of the matchmaking strategies is observed to be insensitive to the exact value of N used.

4.4.5. Error in User-Estimated Runtimes

For the normally distributed error in user-estimated runtimes, the band of error B (defined in Section 4.3.3) is varied as a parameter to study its impact on system performance. B is varied for a wide range of values between 0% and 100%. To study the effect of over/underestimation, the translation co-efficient ă in Equation 4.8 is changed such that mean percentage error η_{mean} is varied while B remains constant. For a distribution with a symmetrical pdf such as a standard normal distribution, η_{mean} can be shown to be equal to ă. η_{mean} is varied for a wide range of values between -50% to +50% that include both underestimation and overestimation of runtimes.

For the model of the error in user-estimated runtimes based on traces, the values of the parameters calculated from the analysis of the traces are used in the experiments.

4.4.6. Proportion of Co-Allocation Requests (PCR)

PCR is the proportion of co-allocation requests in the total number of requests. System performance has been studied for a range of values of PCR between 0.0 and 0.4. Such a range of PCR gives a wide variation in W_R. Increasing PCR beyond 0.4 is likely to give rise to unacceptably high values of W_R. A co-allocation request can request any where from 2 to N resources and for the experiments the number of resources requested by co-allocation requests are uniformly distributed between 2 and N.

4.4.7. Heterogeneity of the System (H)

Since a Grid may consist of multiple resources of the same type (e.g. compute resources) but different physical characteristics (e.g. processor speeds), a given job may have different execution times on different resources. The relative speed of the resources in the context of this thesis always refers to the relative execution times of a given job on different resources. The thesis makes no assumption about the relationship between for instance, the processor speed measured in clock speed and the execution time of the job.

The thesis assumes that execution times of different jobs are affected in the same proportion by the different relative speed of the resources. Thus, the heterogeneity H of the system is defined as the actual execution time of a given job on the slowest resource in the candidate set divided by the actual execution time of that job on the fastest resource in the candidate set. Note that in between the maximum execution time of a job (on the slowest resource) and the minimum execution time of that job (on the fastest resource), there may be other execution times of the job on other resources in the candidate set. For scenarios involving resources with different speeds, the thesis defines a standard resource as the resource that achieves the execution time produced by the model presented in Section 4.3.2. The execution times of the job on other resources in the candidate set can

be computed by either dividing or multiplying the execution time of the job on the standard resource by a certain factor depending on how fast or slow the resource is with respect to the standard resource.

If all resources in the candidate set have the same relative speeds, the value of H is equal to 1. The thesis varies H for a wide range between 1 and 8. Different speed of resources in between the slowest and the fastest resource in the candidate set have been used to investigate the performance of the matchmaking and the co-allocation algorithms.

Note that the actual execution time of a job is used in computation of a number of performance metrics such as W_R. In scenarios where H is greater than 1, the actual execution time of the job on the standard resource is used to compute the value of such performance metrics.

Table 4.5 and Table 4.6 summarize the workload characteristics and parameters for the non-shared and the shared resource models, respectively. Where appropriate both the distribution (Column 2) and the key parameter (Column 3) for each characteristic are included in the tables. Some of the characteristics do not change value during a single simulation experiment. For such characteristics a "constant" label is used in Column 2 while Column 1 and Column 3 have the same entries.

A factor at a time approach is used in the simulation experiments. One of the factors (parameter) is varied while other factors are held at fixed values. The levels of the variable factor are displayed in the last column of Table 4.5 and Table 4.6. Unless specified otherwise, all other factors are held at default values listed in Table 4.5 and Table 4.6. Justification for the levels for each variable factor is provided in the section in

which the parameter was first introduced. In most cases, one of the intermediate levels is chosen as the default value.

4.5. Accuracy of Results

Simulations were run long enough and repeated multiple times, to obtain sufficiently small confidence intervals for the performance metrics. For the experiments, confidence intervals of ±5% (or less) for W_R, P_b, W_A, P_A, and Q_R and ±3 % (or less) for U, R_{AR}, R_{OD} and UU were obtained at a confidence level of 95%. For all the results presented in the thesis, the mean of the performance metric is plotted.

Table 4.5: Summary of the Workload Parameters for the Non-Shared Resource Model

Workload Characteristic	Distribution	Key Parameter	Key Parameter Values	
			Default Level	Variable Levels
Job Runtime ($e_{E,ib}$)	Uniform, Hyper Exponential	Mean Job Runtime	50 minutes	--
Arrival Process	Poisson	Mean Arrival Rate	0.014 jobs/minute	--
Proportion of Advance Reservations (PAR)	Constant	Proportion of Advance Reservations (PAR)	0.4	0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
Percentage Laxity	Uniform	Mean Percentage Laxity (L)	100%	0%, 20%, 60%, 100%, 150%, 200%, 500%
Time Between the Arrival of an AR and Its Earliest Start Time	Uniform	Mean Time Between the Arrival of an AR and Its Earliest Start Time (T_S)	12 Hours	--
Proportion of Preemptable Jobs	Constant	Proportion of Preemptable Jobs	0.0	0.0, 0.5, 1.0
Overheads of Preemption (O)	Constant	Overheads of Preemption (O)	0% of E_E	0%, 10%, 20%, 30% of E_E

Table 4.6: Summary of the Workload Parameters for the Shared Resource Model

Workload Characteristic		Distribution	Key Parameter	Key Parameter Values	
				Default Level	Variable Levels
Job Runtime ($e_{E,ib}$)		Two-Phase Hyper Gamma (see Section 4.3.2)	Mean Job Runtime	2200 seconds	--
Arrival Process		Gamma	Mean Arrival Rate	0.0032 jobs/sec	0.0016, 0.0024, 0.0032,0.004 jobs/sec
Job Size (s_{ib})		Two-Stage Uniform, Other (see Figure 4.2)	Mean Job Size	8.4 nodes	--
Error in User Estimated runtimes – First Model	Band of Error (B)	Constant	Band of Error (B)	20%	0%, 20%, 40%, 60%, 80%, 100%
	Percentage Error (η)	Normal	Mean Percentage Error (η_{mean})	0%	-50%, -40%, -30%, -20%, -10%, 0%, 10%, 20%, 30%, 40%, 50%
Error In User Estimated Runtimes – Second Model		Hyper-Exponential, Exponential, Uniform, (see Table 4.4 and Section 4.3.3)	See Section 4.3.3	Based on SP2 Logs (see Table 4.4 and Section 4.3.3)	
Proportion of Advance Reservations (PAR)		Constant	Proportion of Advance Reservations (PAR)	1	0.5, 1.0
Percentage Laxity		Uniform	Mean Percentage Laxity (L)	400%	50%, 100%, 200%, 400%, 700%, 1000%
Time Between the Arrival of an AR and Its Earliest Start Time		Uniform	Mean Time Between the Arrival of an AR and Its Earliest Start Time (T_S)	12 Hours	--
Size of the Candidate Set		Constant	Size of the Candidate Set	4	2, 4, 6
Proportion of Co-Allocation Requests		Constant	Proportion of Co-Allocation Requests	0.0	0.0, 0.2, 0.4
Heterogeneity of the System (H)		Constant	Heterogeneity of the System (H)	1	1, 4, 8

Chapter 5

QoS Aware Scheduling on Non-Shared Grid Resources

5.1. Scheduling Problem on Non-Shared Resources

As discussed in Section 3.3.2, the performance of individual resources and hence that of a Grid system as a whole depends largely on the effectiveness of the scheduler deployed to schedule on-demand and advance reservation requests. The scheduler should seek to maximize resource utilization while meeting the QoS constraints of all jobs accepted. The problem of scheduling on-demand and advance reservation requests with laxity on non-shared resources has been formally defined in Section 3.3.2.1. Since the problem is an NP-Complete problem and since a large number of jobs are possible in a Grid environment, scalability in terms of being able to handle a large number of requests is one of the desired characteristics of the scheduling algorithm.

As discussed in Section 2.6.1.1 most of the algorithms presented in the real-time domain for solving similar problems do not scale and make certain assumptions such as all jobs are known a priori, jobs are periodic and/or completely preemptable. These assumptions are not always true for the Grid domain. This thesis presents the Scaling through Subset Scheduling (SSS) algorithm for scheduling OD requests and ARs with laxities on non-shared resources. SSS is adapted from the algorithms in the real-time domain. SSS can study scenarios involving both non-preemptable and preemptable jobs including the cases where overheads are associated with job preemption. SSS is observed

to be a scalable algorithm. For example, for the experiments presented in Section 5.3, SSS scheduled thousands of jobs in a reasonable amount of time. SSS is discussed next.

5.2. Scaling through Subset Scheduling (SSS) Algorithm

At the arrival of every new job, SSS seeks to obtain a feasible schedule for the set of jobs already in schedule and the new job. If a feasible schedule is not found, the job is rejected. Otherwise, if the given resource is selected by MMC for that job, the job is added to the set of scheduled jobs. As each job finishes executing on the resource, it is removed from the set of scheduled jobs. The thesis defines *scheduled-time* of a certain job as the time at which it is scheduled to start its execution under the current schedule of the resource. Different segments of a preemptable job may be scheduled to be executed at different times.

Whenever a new request arrives, the SSS algorithm first identifies all those jobs in the resource schedule that can affect the feasibility of the new schedule with the new request and then seeks to obtain a feasible schedule for only that subset of jobs S. This helps in significantly reducing the completion of the algorithm. Given a set of jobs S, the algorithm first finds an initial solution using a well-known heuristic. If the initial solution is infeasible, it is improved using *pruned branch-and-bound technique* until either a feasible schedule is found or the algorithm determines that none exists. Since Problem 1 (defined in Section 3.3.2.1) is an NP-Complete problem, it is possible to come up with pathological situations (when the size of set S is very large) in which the completion time of the SSS algorithm is very large. However, this research argues that such situations are not likely to arise in the Grid domain (see Section 5.4.1). Nevertheless, if such a situation occurs, SSS limits its search space to prevent extremely large scheduling times.

Modification of SSS for such scenarios is discussed in Appendix A.1. There is a potential tradeoff between the optimality of the solution and the completion time of the algorithm.

By definition, *S contains at least all those jobs that if not included in S, the resulting new schedule is infeasible although a feasible schedule exists.* For example, consider a scenario shown in Figure 5.1. At the arrival of job j_4, if S does not include at least j_1, j_2 and j_4, a feasible solution can not be obtained. Let us consider a scenario in which S contains only j_2 and j_4. As j_1, which is not included in S for scheduling, completes its execution at 7, any of the jobs in S can not start its execution before 7. Under these conditions, if j_2 is scheduled before j_4, j_4 will miss its deadline. Similarly, if j_4 is scheduled before j_2, j_2 will miss its deadline. As shown in the figure when S includes j_1, j_2 and j_4 a feasible solution can be obtained. Thus if a feasible solution exists, a subset that must include at least those jobs that are required to obtain a feasible solution is called the set S. The discussion on finding that subset S is explained later in this section. Once the exact subset of jobs 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. The thesis modifies the SCHRAGE heuristic [SCH71] used in [MCM75], for scenarios involving both non-preemptable and preemptable jobs for finding an initial solution. The modified heuristic supports preemptable jobs and 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 jobs with $t_i \leq t$. If no such job exists, set t equal to the earliest t_i among all jobs in S.

Step 3: Among all jobs in S with $t_i \leq t$, select a job with the earliest d_i. Break ties by selecting a job with the highest $e_{E,ib}$. If the job selected is preemptable, go to Step 5.

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

Step 5: If $e_{E,ib}$ is the estimated execution time of the selected job and d_i its deadline, then among all jobs in S with $t_k \leq t + e_{E,ib}$ find all jobs with deadline less than d_i. If no such job exists, go to Step 4. Otherwise, among that set of jobs find a job with the earliest t_k and set pT equal to its earliest start time.

Step 6: Schedule the job selected in Step 3 from t to pT. Preempt that job at time pT. Update the estimated execution time of that job in S as $e_{E,ib} = e_{E,ib} - (pT - t)$. 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 shows that for non-preemptable jobs the algorithm does not remove the job until it finishes its execution on the resource even if during its execution a new job with a lower deadline becomes available for execution. This is because the job is non-preemptable. For preemptable jobs, the algorithm strictly uses the earliest-deadline-first strategy.

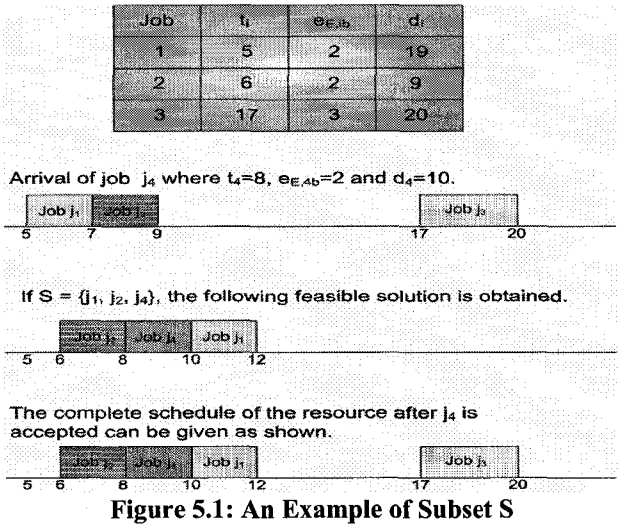


Figure 5.1: An Example of Subset S

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, the algorithm accepts the new request and updates the overall

schedule of the resource. If it is infeasible, the algorithm checks by calculating lower bounds on the lateness of jobs using equations given in [MCM75], whether the lateness of the job that misses its deadline by more time than any other job, known as *critical job*, can be improved. If the lateness of the critical job cannot be improved, the solution is called *optimal* and the algorithm rejects the new request. If the solution is infeasible but not optimal, the algorithm finds a subset of jobs that if scheduled after the critical job can improve its lateness. Such a subset is known as *generating set*. It can be shown that if the initial solution is obtained using SCHRAGE heuristic or any of its derivatives, generating set consists of the set of non-preemptable jobs that belongs to the same block as the critical job and have deadlines greater than that of the critical job [MCM75]. For each job in the generating set, a new solution can be obtained by scheduling that job after the critical job. This is done by setting the earliest start time of that job equal to the earliest start time of the critical job and then obtaining a new solution using the modified SCHRAGE heuristic. The resulting set of solutions where each solution corresponds to each job in the generating set, if infeasible, can further be improved by finding their critical jobs and corresponding generating sets. This process is continued and a tree of solutions is obtained with the initial solution at its root. Each solution in a tree is known as a *solution-node*. This process of solution-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 jobs are known in advance. However, in a Grid domain jobs arrive one by one. Hence, if the initial solution is infeasible but there exists more than one feasible solutions corresponding to more than one jobs in the generating set, it is impossible to identify *a priori*, selection of which job

in generating set to produce the child node can get the maximum benefit later. This is because some of the jobs might have already executed when a certain new request arrives. Selecting a job 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 jobs as seen at the start time of the critical job. If j_i is the critical job and j_j one of the jobs in the generating set, effective laxity of j_j at t_i can be given as $d_j - e_{E,jb} - t_i$. The algorithm then selects the job with the highest effective laxity to produce the child solution-node. Note that the SSS algorithm selects a job with the highest flexibility and hence minimizes the chance of an infeasible schedule. At the same time, if two jobs have equal deadlines, a job with a lower execution time will be selected to produce a child solution-node. This increases the current utilization of the resource as the job with the higher execution time is scheduled to execute.

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

Initially,

S = {New Job}

SP = max {current system time, min {t_i for all jobs j_i in S}}

EP = max {d_i for all jobs j_i in S}

The jobs are then added in S during each iteration according to the following rules:

- If any job j_i not in S has $t_i \geq$ SP and its scheduled-time < EP, then add j_i to S.
- If any job j_i not in S has scheduled-time < EP and it finishes its execution after SP, then add j_i to S.

When S is obtained using the given methodology, all jobs that can affect the feasibility of the new schedule are included in S and S is said to be complete. The thesis further optimizes the algorithm to reduce its completion time and the optimized algorithm is presented 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, $e_{E,ib}$ and d_i be the earliest start time, execution time and deadline of the new job request.

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

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

A. If EP < d_i and the new job is a preemptable job and TIU equals $e_{E,ib}$.

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

C. TIU = $x*e_{E,ib}$.

D. TIUS = $y*e_{E,ib}$.

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 job 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 job scheduled at EP, job j_j. Increment EP by 1 recursively as long as in the resource schedule job j_j is scheduled at EP. If job j_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_j has a finite deadline. If it does not have a finite deadline, set SP equal to scheduled-time of j_j and go to Step 2.

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

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

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

If S is obtained using the above algorithm, all jobs that are scheduled before start point but after the current system time are those that finish their execution before the earliest start time of any of the jobs in S. Hence, they can not affect the scheduled-time of any of the jobs in S and hence of the new request. Note that, in Step 4 and Step 6 if the job that has an earliest start time less than start point is an OD request the algorithm does not set SP equal to its earliest start time. The reason is that an OD can be delayed indefinitely and hence 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 job 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 job cannot be accommodated in the current schedule even if we include all the jobs that are scheduled after the end point. The reason is that the jobs that are scheduled after the end point can not improve the lateness

of the critical job. 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 runtimes of jobs in the schedule. None of the jobs that are scheduled after the end point can delay the execution of any of the jobs in S. However, the jobs in S may affect the scheduled-time of the jobs 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, the algorithm re-obtains 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 solution-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 solution-node of the last solution-node in v. This is done by setting the earliest start time of the job with the highest effective laxity in the generating set equal to t_i. Remove that job from generating set of last solution-node in v. If the resulting generating set size of that solution-node is zero, remove that solution-node from v.

Step 4: Obtain a solution for the new child solution-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 job 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 the end point equal to the current end point and go to step 2. Otherwise, accept the job and break out of the algorithm.

Step 6: Reject the new job.

Note that the SSS algorithm limit the growth of solution-nodes with the number of jobs by, a) running the algorithm for a subset of jobs, b) by logically selecting and producing only one solution-node at a time. Selecting and producing only one solution-node at a time and removing the solution-nodes that consists of optimal solutions reduces the number of open solution-nodes in memory by a large factor.

5.2.1. Handling Scenarios Involving Preemption Overheads with SSS

In order to handle the specific scenarios in which overheads are associated with job preemption and its resumption, the SSS algorithm is modified in two ways. First, the SCHRAGE heuristic [SCH71] for obtaining the initial solution is modified, to account for overheads in job preemption, in the following manner.

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

Step 2: Find among S, all jobs with $t_i \leq t$. If no such job exists, set t equal to the earliest t_i among all jobs in S.

Step 3: Among all jobs in S with $t_i \leq t$, select a job with the earliest d_i. Break ties by selecting a job with the highest $e_{E,ib}$. If the job selected is preemptable, go to Step 5.

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

Step 5: If $e_{E,ib}$ is the estimated execution time of the selected job and d_i its deadline, then among all jobs in S with $t_k <= t + e_{E,ib}$ find all jobs with deadline less than $d_i - o$, where o is the overhead in terms of time in preempting and resuming the selected

job. If no such job exists, go to Step 4. Otherwise, among that set of jobs find a job with the earliest t_k and set pT equal to its earliest start time.

Step 6: If pT – t > o, go to Step 7. Otherwise, among all jobs in S with $t_j \leq t$ find all jobs with $e_{E,jb} \leq (pT - t)$. If no such job 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: Schedule the job selected in Step 3 from t to pT. Preempt that job at time pT. Update the estimated execution time of that job in S as $e_{E,ib} = e_{E,ib} - (pT - t)$. 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 is similar to the modified SCHRAGE heuristic that was used in the SSS algorithm in Section 5.2. However, in Step 5 the algorithm does not preempt a selected job for another job whose deadline is earlier than that of the selected job by a difference less than the overheads involved in preempting and resuming the job. This prevents unnecessary missing of deadlines as depicted in Figure 5.2. In step 6, the algorithm ensures that the resource is not allocated to a job j_i for a duration that is smaller than the time spent in overheads in preempting that job later. Instead, the algorithm tries to allocate the resource to another job j_j that can finish its execution before another job j_k with $d_k < d_i < d_j$ becomes available for execution.

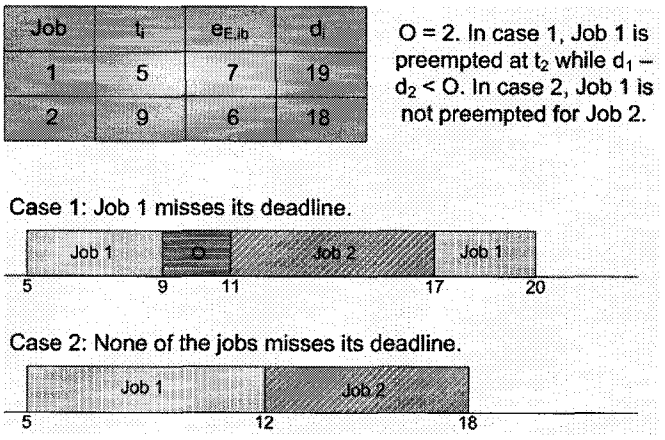


Figure 5.2: Scheduling Preemptable Jobs with Overheads

The second modification in SSS for handling scenarios involving preemption overheads involves a change in condition A of Step 2 of the procedure that is used to determine subset S in the SSS algorithm, presented in Section 5.2. Condition A is modified in the following way.

A. If EP < d_i and the new job is a preemptable job and TIU equals $e_{E,ib} + \kappa * o$, 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 job and κ is the number of times the new job would be preempted if placed in the idle time units after t_i.

The modified condition A ensures that idle segments in resource schedule which are smaller than o are not counted towards the execution time required to run the job. It also takes into account the time spent in overheads in preempting and resuming a preemptable job. The rest of the algorithm remains unchanged.

5.3. Performance Analysis of Resource Liaison and Controller with Non-Shared Resources

Table 5.1 summarizes the sets of experiments conducted with RLC with non-shared resource models. The setup for these experiments has been discussed in detail in Section 4.1.1 while the workload model has been discussed in Section 4.3.1. The experiments in this section assume that estimated runtimes of the jobs are equal to their actual runtimes. The impact of error and strategies related to error handling are the focus of Chapter 7. The values of the parameters for each of the experiment sets are given in Table 5.1. SSS is used as the scheduling algorithm in these experiments.

Case NS.1 described in Table 5.1 studies the impact of PAR and L on performance for non-preemptable jobs. Cases NS.2 and NS.3 study the impact of preemption and that of overheads associated with preemption. Case NS.4 investigates a scenario where both non-preemptable and preemptable jobs are present in the workload.

Cases NS.5, NS.6 and NS.7 study the impact of high variability in service time of jobs on system performance. Cases NS.8 and NS.9 investigate the performance of the Starvation Prevention Manager. The results are presented next.

Table 5.1: Cases Studied with RLC with Non-Shared Resources

Case No.	Dist. of Jobs Service Times	Job Preemption		PAR	L	Starvation Prevention	
		Type of Jobs	Preemption Overheads			SPM Active	Deadline of OD Jobs
NS.1	Uniform	NP	--	0 to 1	0 to 500%	No	Infinity
NS.2	Uniform	P	No	0 to 1	0 to 500%	No	Infinity
NS.3	Uniform	P	0 to 30% of E_E	0 to 1	0 to 500%	No	Infinity
NS.4	Uniform	50% NP & 50%P	No	0 to 1	0 to 500%	No	Infinity
NS.5	Hyper-Exp.	NP	--	0 to 1	0 to 500%	No	Infinity
NS.6	Hyper-Exp.	P	No	0 to 1	0 to 500%	No	Infinity
NS.7	Hyper-Exp.	P	0 to 30% of E_E	0 to 1	0 to 500%	No	Infinity
NS.8	Uniform	NP	--	0 to 1	0 to 500%	Yes	$t_i + 6*e_{E,ib}$
NS.9	Uniform	P	No	0 to 1	0 to 500%	Yes	$t_i + 6*e_{E,ib}$

5.3.1. 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 requests on non-shared Grid resources.

5.3.1.1. Case NS.1: Non-Preemptable Jobs with Uniformly Distributed Service Times and No Starvation Prevention for On-Demand Requests

Probability of Blocking: Figure 5.3(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 OD requests 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

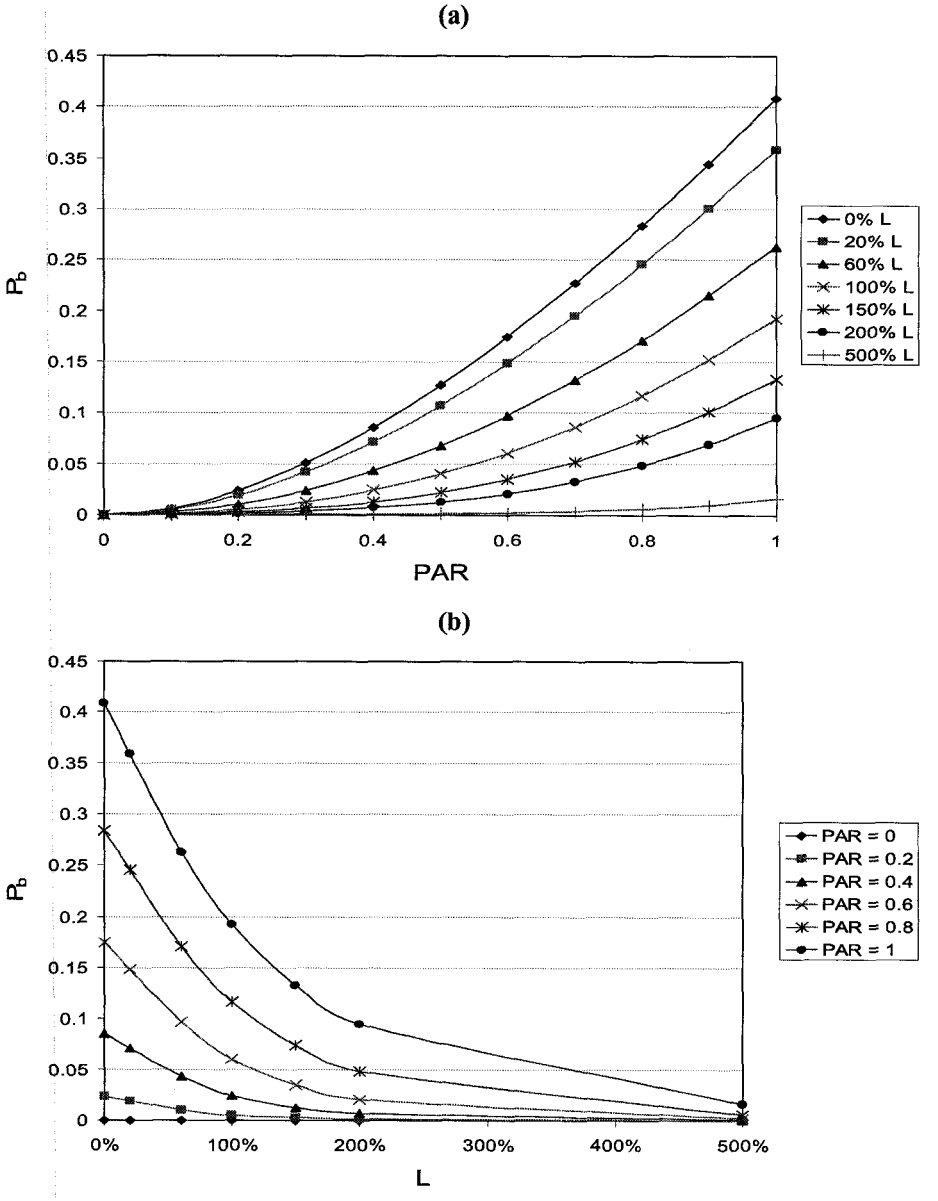


Figure 5.3: Probability of Blocking for Case NS.1
(a) Effect of PAR on P_b (b) Effect of L on P_b

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 L after which P_b increases more rapidly. Given the mean laxity of the jobs, a resource can limit the proportion of requests it accepts as ARs. The results in Figure 5.3 suggest that the PAR corresponding to the knee of the curve can be used as this limiting proportion to keep P_b at reasonable levels.

Figure 5.3(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 job scheduling. The effect becomes more pronounced with the increase in PAR. Thus for 80% requests arriving as ARs, using L = 200% can decrease P_b by 82.93% compared to the case in which ARs have no laxity. The curves shown in Figure 5.3(b) are characterized by a knee. The knee of the curve, that brings P_b to a reasonably small value, is reached for a much smaller value of L compared to the one required to make P_b exactly equal to 0.

Utilization: Figure 5.4(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 jobs in the schedule. Figure 5.4(b) shows that by having some laxity in the jobs 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:

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

Where R is the mean response time of the jobs accepted and W their mean wait time. Note that R – W is not always equal to E_E since jobs with higher runtimes have a higher probability of being rejected.

Response Time of Advance Reservations and On-Demand Requests: Figure 5.5(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 because ARs have deadlines they can arbitrarily delay OD requests that are not associated with any deadlines. Thus, as PAR increases, more and more OD requests are delayed for longer periods. This increases R_{OD} with the increase in PAR until it reaches its maximum

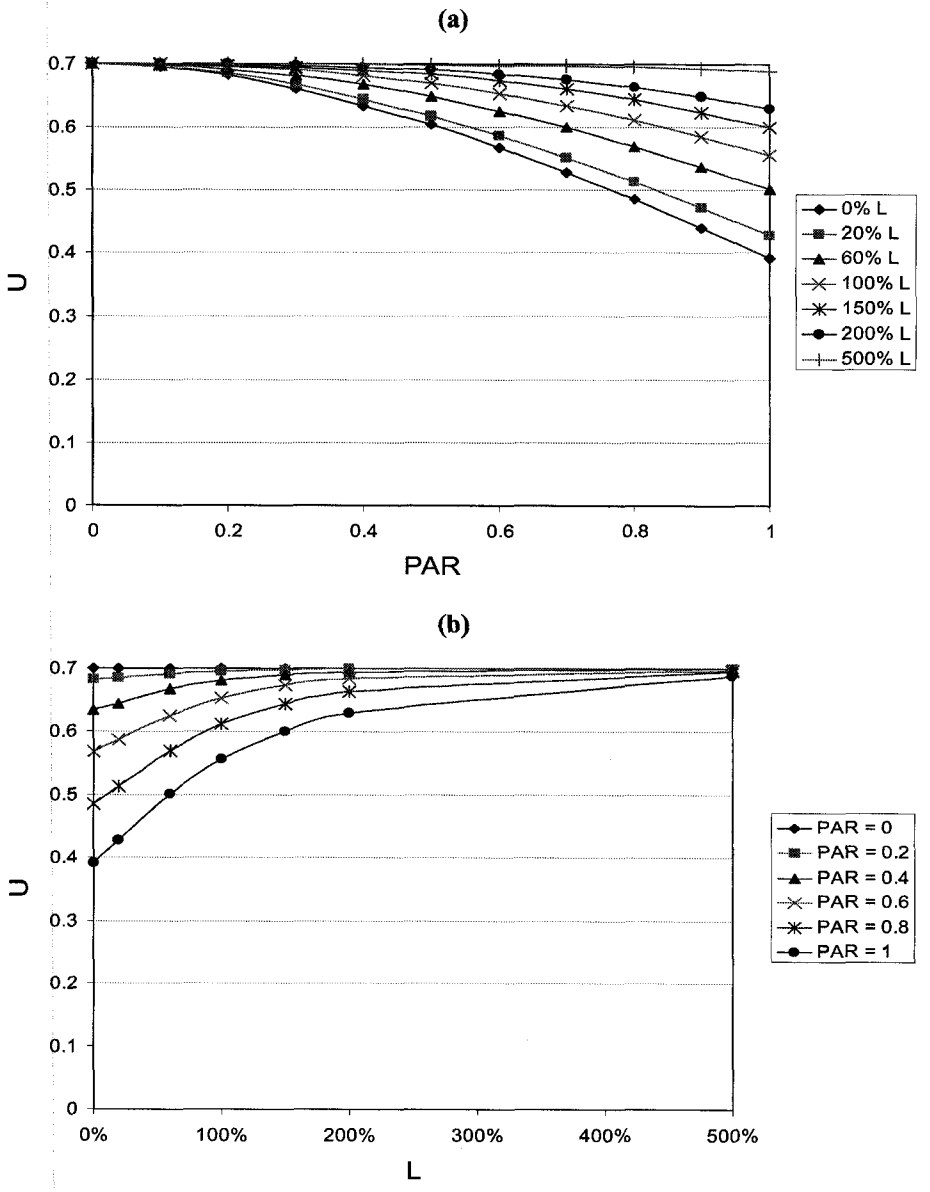
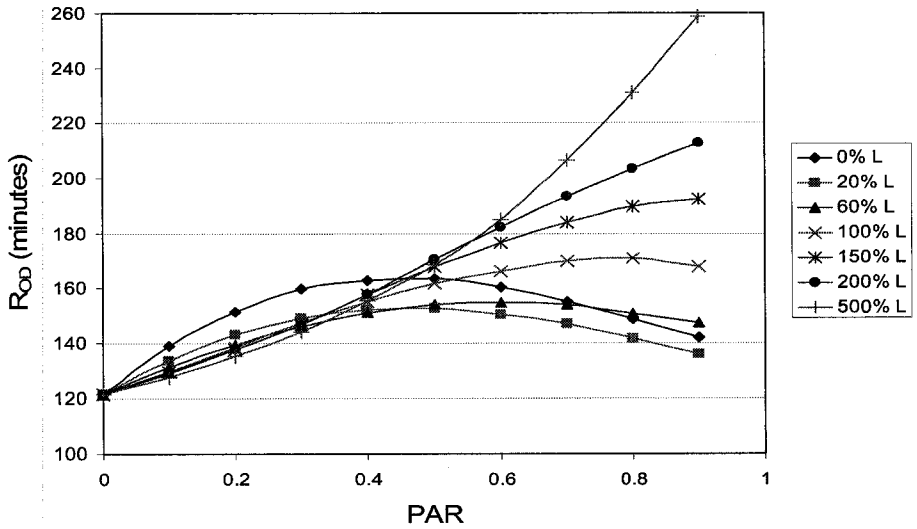


Figure 5.4: Utilization for Case NS.1
(a) Effect of PAR on U (b) Effect of L on U

value. As PAR increases further, due to an increase in P_b, U decreases significantly and this decreases the mean wait time of OD requests. 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 OD requests. Hence, for large L values, R_{OD} is observed to increase with PAR. The PAR

(a)



(b)

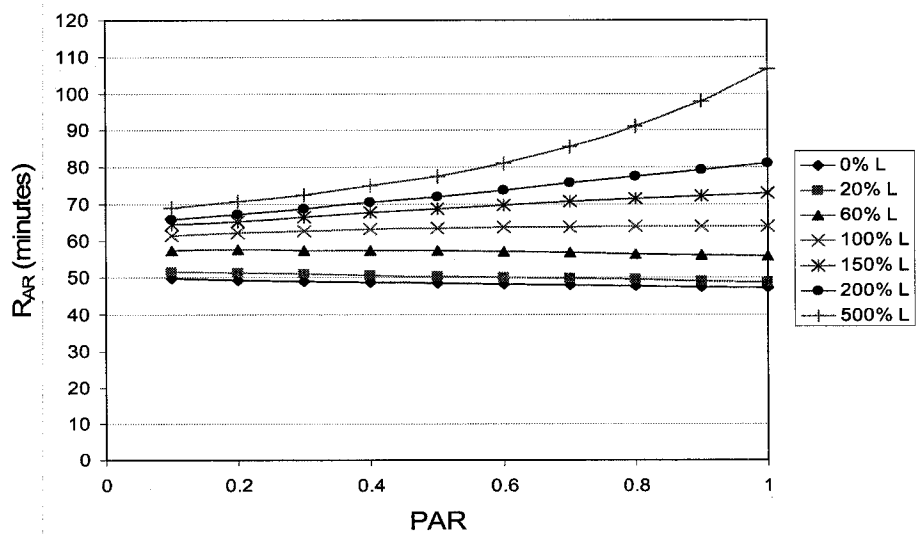


Figure 5.5: Response Time for Case NS.1
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

value can be effectively controlled by the system administrator to achieve the desired value of R_{OD}.

Figure 5.5(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 runtimes have a higher probability of being rejected. As more and more ARs with higher runtimes

are rejected, the mean runtime 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_E. As L increases, mean wait times of ARs tend to increase which increases R_{AR} with L for a 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 seems to be balanced by the decrease in the mean runtime 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.

As there was no error in user-estimated runtimes in this set of experiment, no jobs are aborted by the resource and hence $W_A = 0$, $P_A = 0$ and U = UU for all values of L.

5.3.1.2. Case NS.2 and Case NS.3: Preemptable Jobs with Uniformly Distributed Service Times and No Starvation Prevention for On-Demand Requests

The results for Case NS.2 show that the behavior of preemptable jobs is similar to the one obtained for non-preemptable jobs presented in Section 5.3.1.1. The results for Case NS.2 are included in Appendix A.2. The results for Case NS.2 show that for L > 0, preemptable jobs result in a slightly higher utilization than non-preemptable jobs. Detailed comparison of the performance behavior of non-preemptable and preemptable jobs is presented in Section 5.3.2.

Results obtained for Case NS.3 and their comparison with those obtained for cases NS.1 and NS.2 are also presented in Section 5.3.2.

5.3.1.3. Case NS.4: 50% Non-Preemptable and 50% Preemptable Jobs with Uniformly Distributed Service Times and No Starvation Prevention for On-Demand Requests

The results obtained with 50% preemptable and 50% non-preemptable jobs are presented in Appendix A.3. The results for this case lie almost exactly between the results

discussed in Section 5.3.1.1 and Section 5.3.1.2. This shows that when non-preemptable and preemptable jobs are combined equally, each type of jobs contributes equally to the results.

5.3.1.4. Case NS.5: Non-Preemptable Jobs with Hyper-Exponentially Distributed Service Times and No Starvation Prevention for On-Demand Requests

Utilization: Figure 5.6(a) and Figure 5.6(b) respectively show the effect of PAR and L on system utilization when non-preemptable jobs with hyper-exponentially distributed service times are used. The curves are similar to those obtained with uniformly distributed service times in Section 5.3.1.1 and are characterized by a knee. However, the results show that for any given values of PAR and L, U for hyper-exponentially distributed service times is lower than that obtained with uniformly distributed service times. The difference between U obtained with hyper-exponentially distributed service times and that obtained with uniformly distributed service times increases with PAR and can be as high as 10.31%. This shows that variability in job service times can significantly affect system performance. Higher variability in job service times results in more schedule conflicts and hence lower system utilization.

Response Time of Advance Reservations and On-Demand Requests: Figure 5.7(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 OD requests are delayed to accommodate ARs. This increases R_{OD}. The results show that for any given values of PAR and L, R_{OD} in Figure 5.7(a) is substantially higher than that in Figure 5.5(a). The difference increases with PAR and can be as high as 800%. This shows that high variability in job service times can severely affect the response times of on-demand requests.

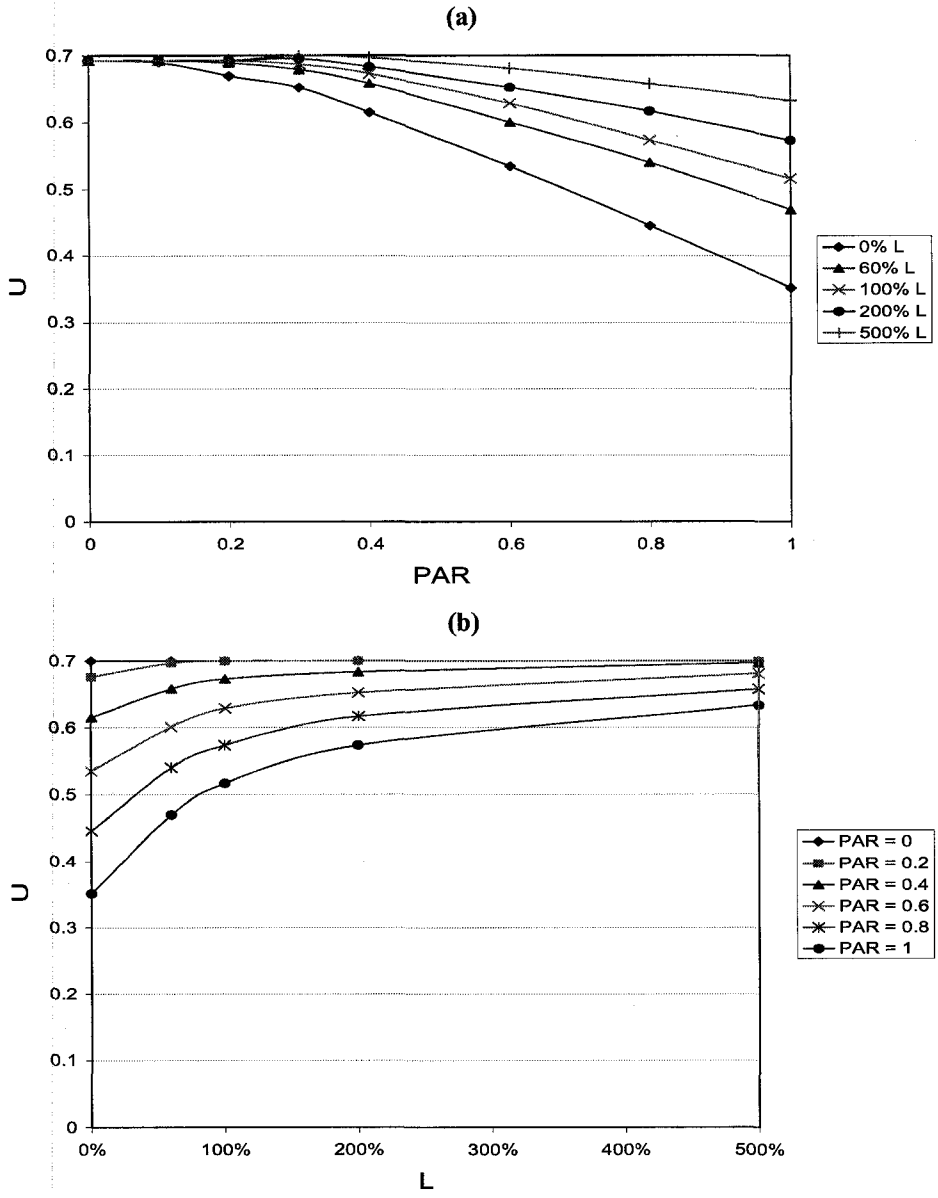


Figure 5.6: Utilization for Case NS.5
(a) Effect of PAR on U (b) Effect of L on U

The curves in Figure 5.7(b) for R_{AR} obtained with hyper-exponentially distributed service times are similar to those in Figure 5.5(b) obtained with uniformly distributed service times. Comparison of Figure 5.7(b) with Figure 5.5(b) shows that for lower L values R_{AR} for the NS.5 case is smaller than that for the NS.1 case and the difference between the two can be as high as 21.92%. The reason is that for smaller L values, there are more schedule conflicts in the NS.5 case due to higher variability in job service times.

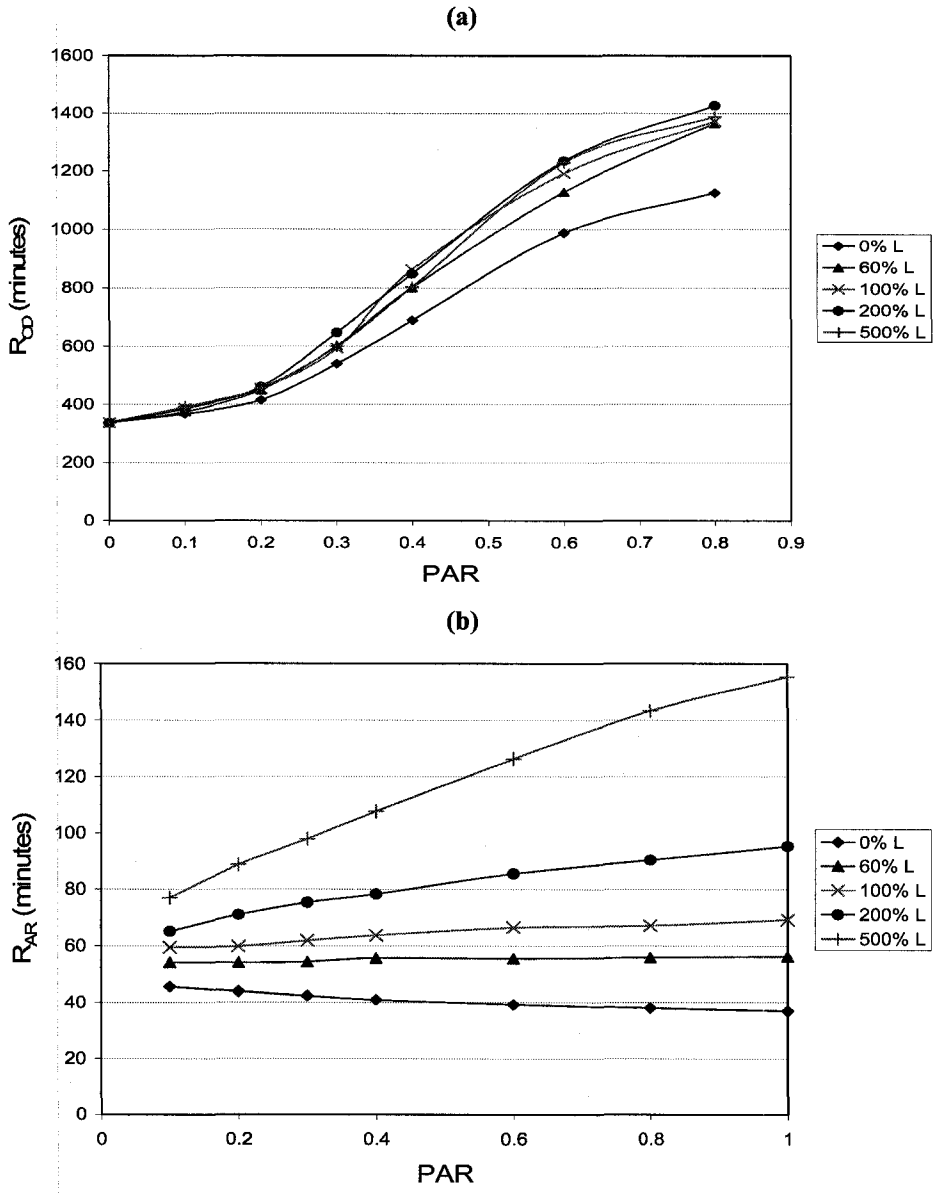


Figure 5.7: Response Time for Case NS.5
(a) Effect of PAR on R_{OD} (b) Effect of PAR on R_{AR}

Under such conditions, more jobs with large services times are rejected which substantially decreases the average execution time of the ARs accepted. Since for smaller L values mean wait times of ARs is small, substantial decrease in mean execution times of the ARs accepted significantly decreases response times of ARs.

For large L values, however, R_{AR} for the NS.5 case is higher than that with the NS.1 case and the two can differ by as much as 45.64%. The reason is that for high L

values mean wait times of ARs for the NS.5 case are substantially higher than that for the NS.1 case due to higher variability in job service times. High variability in job service times results in large mean wait times as a job with a very large execution time can substantially delay the execution of a number of jobs.

As there was no error in user-estimated runtimes in this set of experiment, no jobs are aborted by the resource and hence $W_A = 0$, $P_A = 0$ and U = UU for all values of L.

5.3.1.5. Case NS.6 and Case NS.7: Preemptable Jobs with Hyper-Exponentially Distributed Service Times and No Starvation Prevention for On-Demand Requests

Results for Case NS.6 presented in Appendix A.4 show that the behavior of preemptable jobs is similar to the one obtained for non-preemptable jobs. However, the results show that for $L > 0$, preemptable jobs result in a slightly higher utilization than non-preemptable jobs. Detailed comparison of the performance behavior of non-preemptable and preemptable jobs is presented in Section 5.3.2.

Results obtained for Case NS.7 and their comparison with those obtained for cases NS.5 and NS.6 are also presented in Section 5.3.2.

5.3.2. Effect of Preemption

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

This section compares the results for the NS.1 case in Table 5.1 where jobs are non-preemptable with the NS.2 and NS.3 cases where jobs are preemptable. Similarly, the NS.5 case is compared with the NS.6 and NS.7 cases. For the NS.3 and NS.7 cases, overheads are associated with job preemption. The mean overhead O is expressed as a

percentage of mean service times of the jobs E_E. O represents the mean time spent in overheads associated with job preemption and its resumption. For example, if jobs represent data transfers on a lightpath then O represents mean 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) [MAM03], the total time spent in overheads is just less than 1 minute [LAVir] 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. This research varies O between 0% of E_E (0 minutes) representing the ideal case to 30% of E_E (15 minutes) representing the worst-case scenario.

This section present the results obtained for PAR = 0.4 as such a value of PAR results in reasonable P_b values as shown in Figure 5.3(a). Results for other values of PAR are in Appendix A.5. Figure 5.8, Figure 5.9 and Figure 5.10 compares 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 jobs are preempted compared to the case when they are not preempted.

Utilization: Figure 5.8(a) shows that for a given value of L, U in the NS.2 and NS.3 cases where jobs are preempted is higher than that obtained in the NS.1 case. The reason is that segments of jobs 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 jobs 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 5.8(a) peaks at 1.05% when no overheads are associated with preemption. Results presented in Figure 5.8(b) for the hyper-exponentially distributed service times are similar to Figure

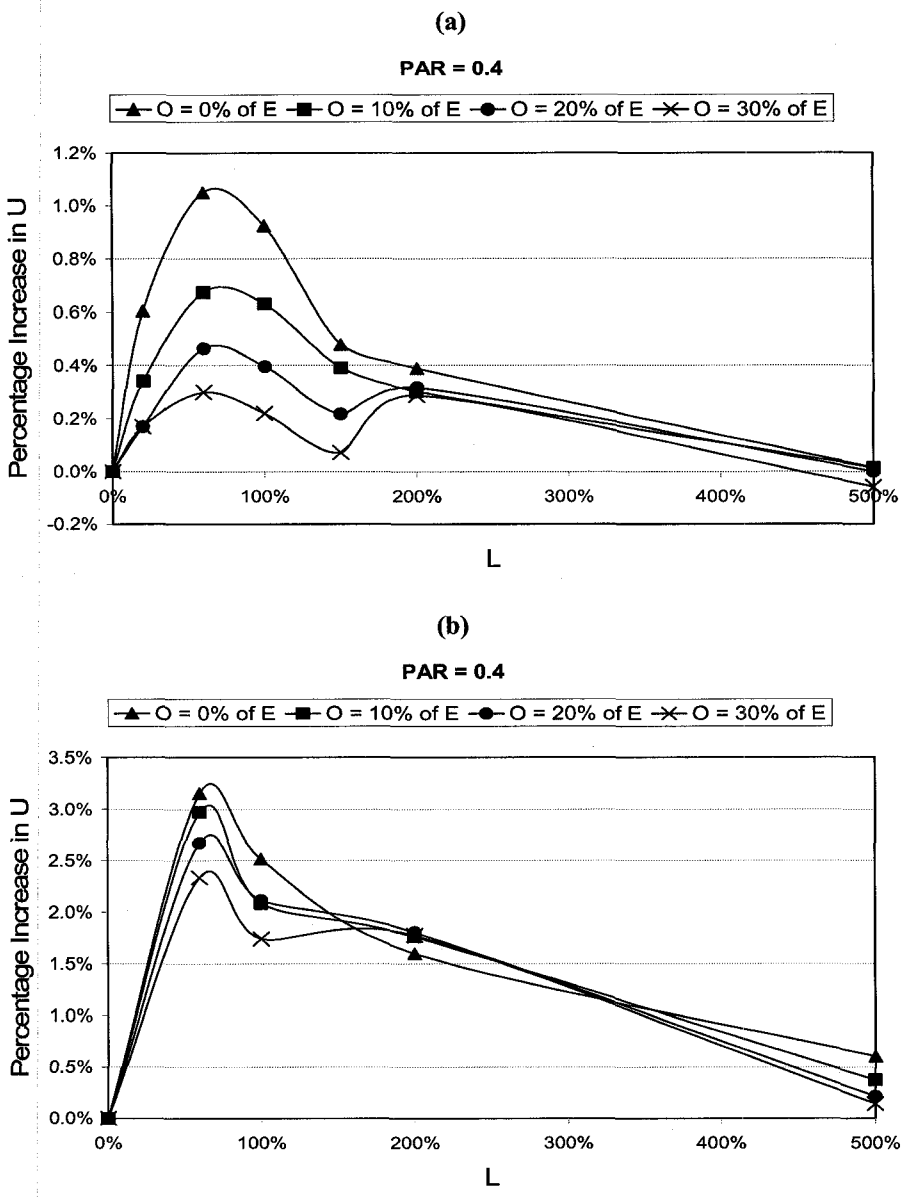


Figure 5.8: Effect of Preemption on Utilization for PAR = 0.4
(a) Effect on U for Uniformly Distributed Service Times
(b) Effect on U for Hyper-Exponentially Distributed Service Times

5.8(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 5.8 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 cases with preemptable jobs none of the requests can be preempted, as preemption for a non-zero time would make them miss their deadlines. 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 preemption for achieving high utilization.

Response Time of Advance Reservations and On-Demand Requests: Figure 5.9(a) shows that for lower L values, R_{OD} in the NS.2 and NS.3 cases is lower than that in the NS.1 case. This is because in the NS.2 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 NS.2 and NS.3 cases starts becoming higher than that in the NS.1 case). On the other hand, in Figure 5.10(a) with the increase in L, R_{AR} with the NS.2 and NS.3 cases becomes increasingly smaller compared to that with the NS.1 case. This shows that in the NS.2 and NS.3 cases many OD requests with infinite deadlines are preempted by the scheduler for AR requests with smaller deadlines. This is because the SCHRAGE heuristic in the SSS scheduler gives priority to the jobs with earlier deadlines while finding an initial

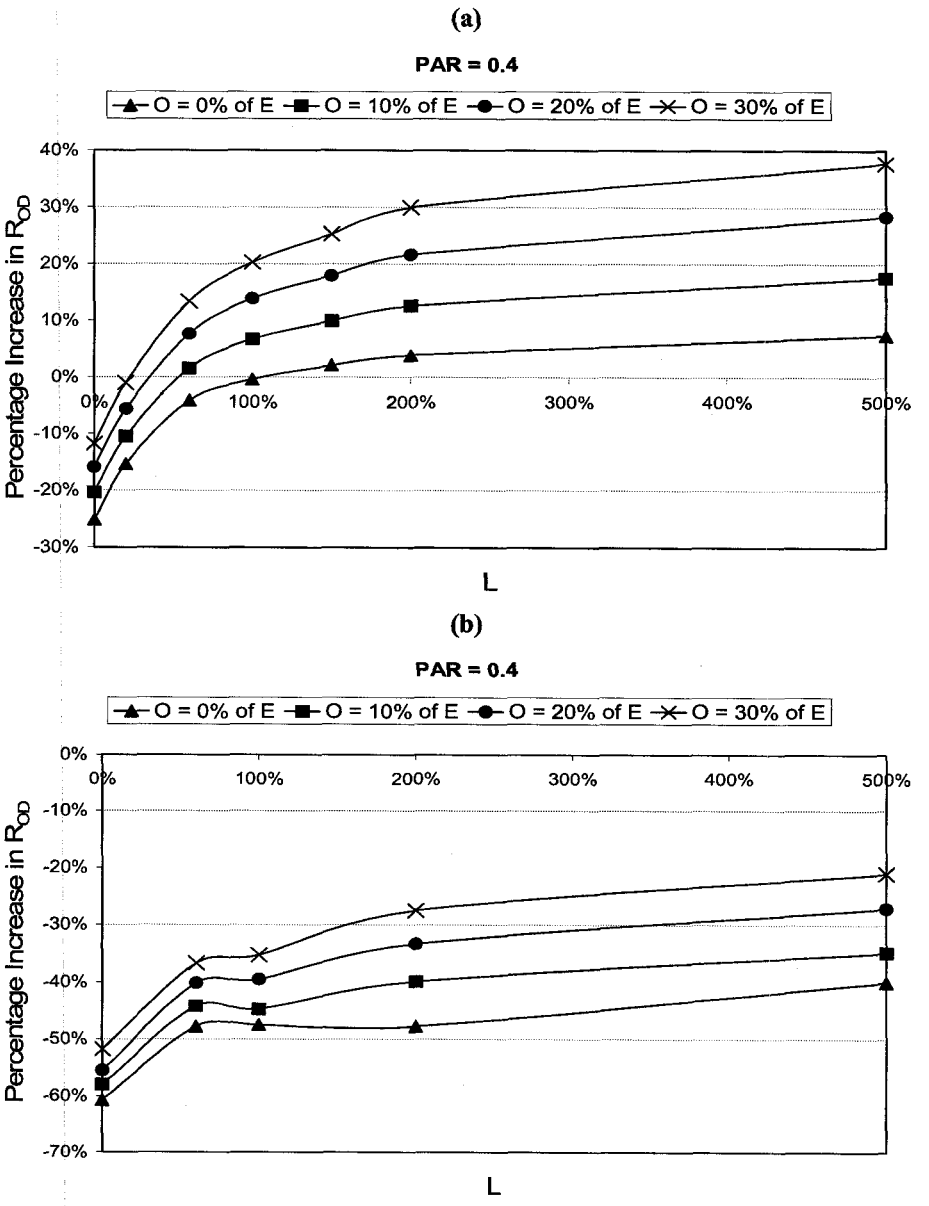


Figure 5.9: Effect of Preemption on Response Time of On-Demand Requests for PAR = 0.4
(a) Effect on R_{OD} for Uniformly Distributed Service Times
(b) Effect on R_{OD} for Hyper-Exponentially Distributed Service Times

solution. On the other hand, in the NS.1 case as L increases, comparatively fewer OD requests needs to be re-scheduled at a later time to accommodate ARs. For higher L values, this results in R_{OD} for the NS.2 and NS.3 cases to be higher than that for the NS.1 case while R_{AR} for the NS.2 and NS.3 cases to be smaller than that for the NS.1 case.

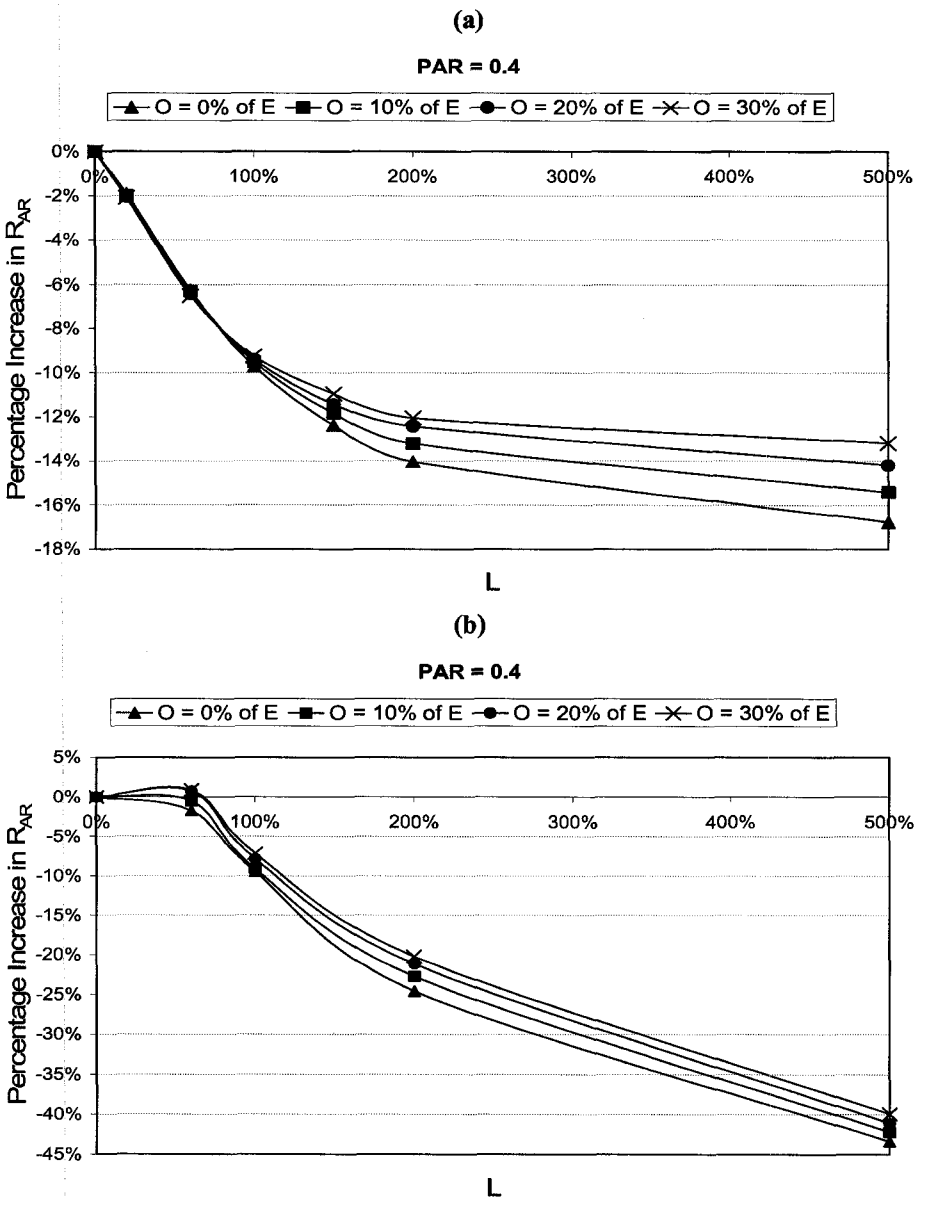


Figure 5.10: Effect of Preemption on Response Time of Advance Reservations for PAR = 0.4
(a) Effect on R_{AR} for Uniformly Distributed Service Times
(b) Effect on R_{AR} for Hyper-Exponentially Distributed Service Times

The results in Figure 5.9(b) and Figure 5.10(b) are similar to those in Figure 5.9(a) and Figure 5.10(a) respectively. However, in Figure 5.9(b), initially R_{OD} with the NS.6 and NS.7 cases is so small in comparison to the NS.5 case that although the difference between R_{OD} with preemptable jobs and that with non-preemptable jobs

becomes increasingly smaller with the increase in L, R_{OD} with preemptable jobs never becomes higher than that with non-preemptable jobs even for O = 30% of E_E.

The results thus show that for PAR = 0.4, job 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 job service times is high. However, preemption is not justified for uniformly distributed job service times with low co-efficient of variation.

Results in Appendix A.5 obtained with PAR = 0.1 are similar to those for PAR = 0.4 but shows that for PAR = 0.1, improvement in U and R_{OD} is smaller compared to PAR = 0.4. On the other hand, the results show that the effect of preemption becomes more pronounced with an increase in PAR. For PAR = 1.0 the improvement in utilization with preemptable jobs is 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 jobs such as the distribution of service times and the overheads associated with preemption, proportion of advance reservations should also be taken into account to decide whether or not preemption is justified for a given scenario.

5.3.3. Preventing Starvation of On-Demand Requests

Results in Section 5.3.1 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 OD requests 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 5.4 and Figure 5.6). 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 5.5 and Figure 5.7). In order to prevent these situations resulting from potential starvation of on-demand requests, the thesis presents Starvation Prevention Manager (SPM) which is discussed in detail in Section 3.3.3. In order to study the effect of SPM, this section uses the Deadlines Policy (DP) discussed in Section 3.3.3.1. The DP policy associates a hard deadline with every OD and ensures that each OD accepted meets its deadline. The deadline chosen for every OD is equal to 6 times its execution time as this is equivalent to L = 500% for ARs. This would provide us with a direct comparison of OD requests and ARs for equal deadlines.

5.3.3.1. Case NS.8: Non-Preemptable Jobs with Uniformly Distributed Service Times and Starvation Prevention for On-Demand Requests

Utilization: Figure 5.11(a) shows that for the NS.8 case, PAR affects U in a similar fashion as it affects U in the NS.1 case (discussed in detail in Section 5.3.1.1). Figure 5.11(b) shows the percentage decrease in U in the NS.8 case compared to the NS.1 case. The figure shows that the utilization achieved with a given PAR and L in this case is slightly lower than that in the NS.1 case. This is the cost a resource incurs in preventing the starvation of OD requests. However, Figure 5.11(b) shows that the difference in U for a given PAR and L is never more than 3.11% and it diminishes for high values of L.

Response Time of Advance Reservations and On-Demand Requests: Figure 5.12(a) shows the effect of PAR on R_{OD}. The figure shows that for values of L less than 100% the behavior for the NS.8 case is similar to that obtained with the NS.1 case (shown in Figure 5.5(a)). However, R_{OD} in Figure 5.12(a) for higher values of L is significantly lower than R_{OD} in Figure 5.5(a). This is shown in Figure 5.12(b) where the percentage decrease in R_{OD} in the NS.8 case compared to that in the NS.1 case is plotted.

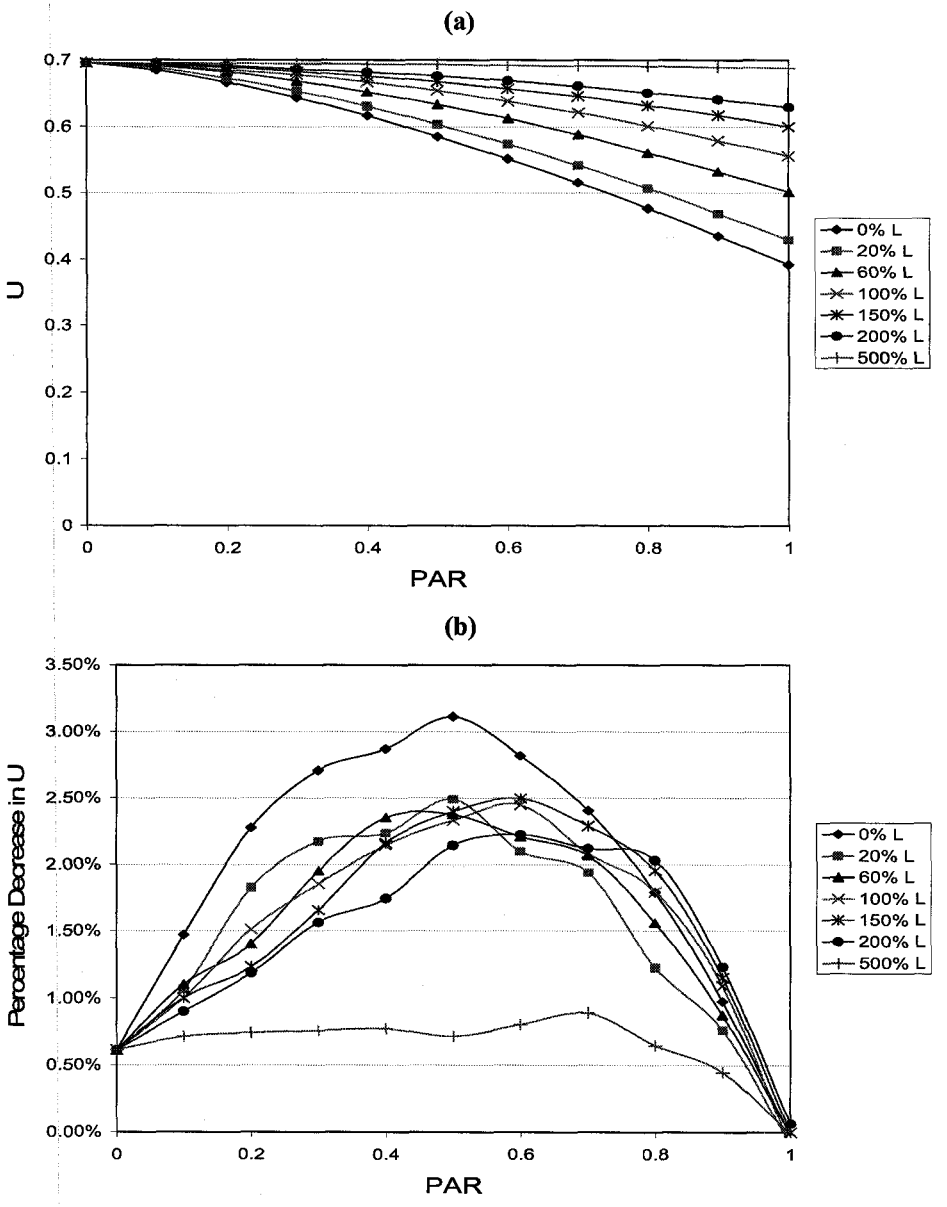


Figure 5.11: Effect of Starvation Prevention Manager on U for Non-Preemptable Jobs
(a) Effect of PAR on U for Case NS.8
(b) Percentage Decrease in U in Case NS.8 Compared to Case NS.1

The difference in R_{OD} in Figure 5.12(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 5.5(a) where for L values above 150%, R_{OD} for a given value of PAR increases with L, in Figure 5.12(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 OD requests and hence OD

requests start getting as much priority as ARs by the SSS scheduler which gives priority to jobs with earlier deadlines when obtaining an initial solution.

The effect of PAR on R_{AR} is given in Figure 5.13 which shows that for values of L up to 200%, R_{AR} for a given PAR and L is approximately the same as in Figure 5.5(b). However, for L = 500% with low PAR values, R_{AR} in this case is significantly higher (up

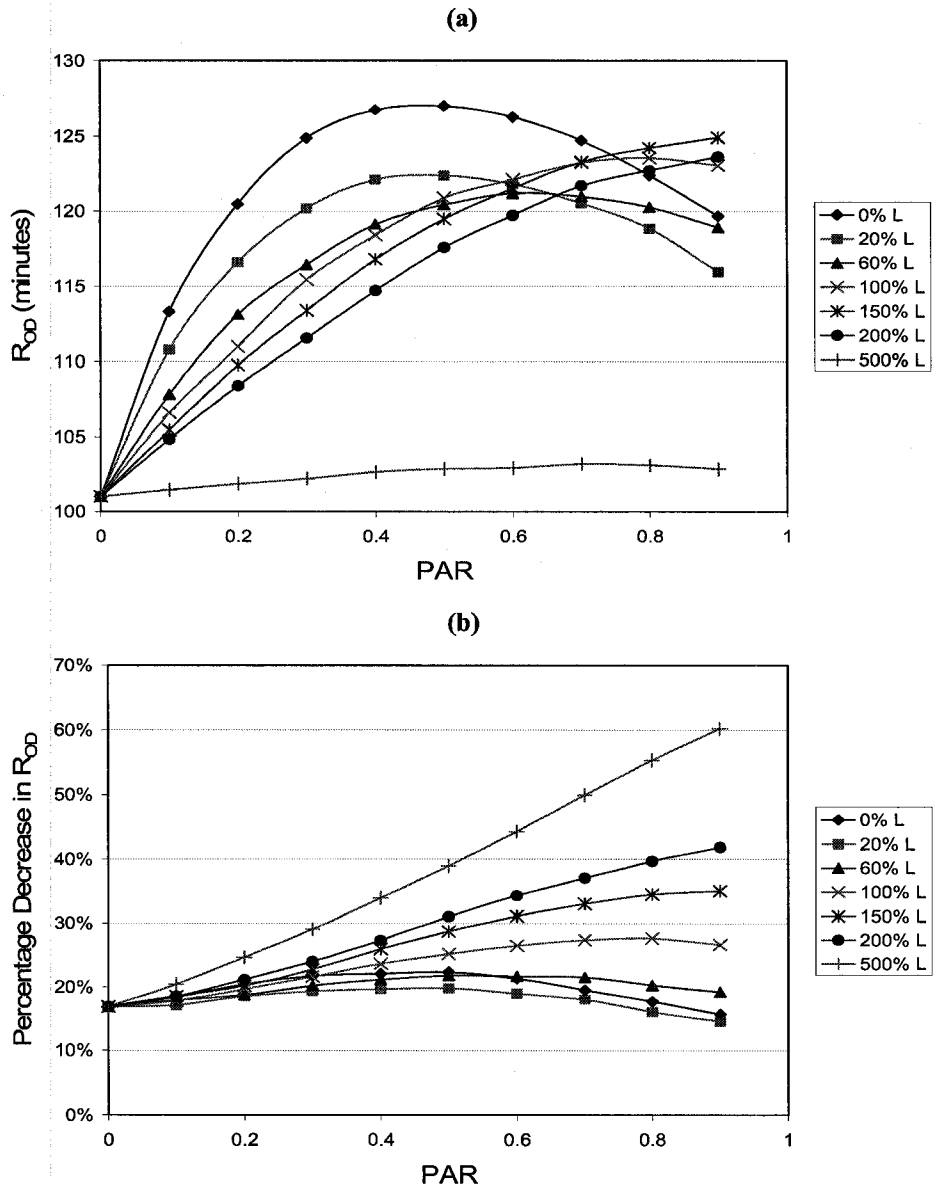


Figure 5.12: Effect of Starvation Prevention Manager on R_{OD} for Non-Preemptable Jobs
(a) Effect of PAR on R_{OD} for Case NS.8
(b) Percentage Decrease in R_{OD} in Case NS.8 Compared to Case NS.1

to 50%) than that in the NS.1 case. For higher PAR values, the difference approaches zero. For L = 500%, R_{AR} in Figure 5.13 is almost equal to R_{OD} in Figure 5.12(a) for a given value of PAR. This shows that if jobs 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 AR and its start time does not significantly affect performance.

5.3.3.2. Case NS.9: Preemptable Jobs with Uniformly Distributed Service Times and Starvation Prevention for On-Demand Requests

The results obtained by activating SPM for preemptable jobs are presented in Appendix A.6. The results show behavior similar to that obtained for non-preemptable jobs.

5.4. Performance of the Scaling Through Subset Scheduling Algorithm

5.4.1. Scalability

As discussed in Section 5.2, since Problem 1 (defined in Section 3.3.2.1) is an NP-Complete problem, it is possible to come up with pathological situations in which the

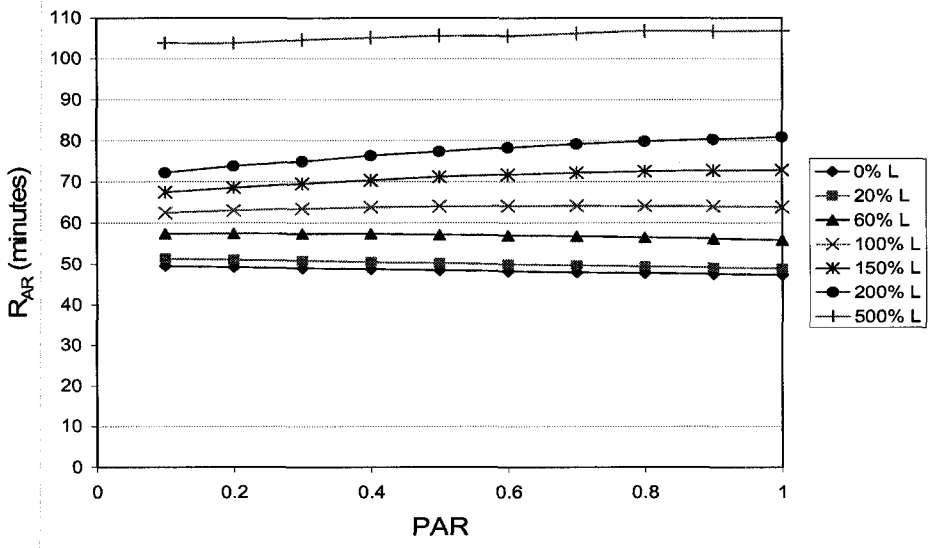


Figure 5.13: Effect of PAR on R_{AR} for Case NS.8

completion time of the SSS algorithm is very large and SSS becomes a limiting factor in successful scheduling of jobs. However, this research argues that such situations are not likely to arise in the Grid domain. The reason is that the execution times of the jobs in the Grid domain are generally orders of magnitude higher than the time required to schedule them (see Section 4.3). The execution times of the Grid jobs are generally of the order of minutes, if not hours (see Section 4.3), while this section shows that the time required by SSS to schedule a new job is of the order of a fraction of a millisecond. Nevertheless, modified SSS for handling such situations, where the completion time of the SSS algorithm may become a limiting factor in successful scheduling of jobs, is presented in Appendix A.1. The modified SSS limits the number of solution-nodes produced and hence prevents the increase in the completion time of the SSS algorithm.

In the experiments presented in Section 5.3, the SSS algorithm successfully scheduled hundreds of thousands of jobs. The results show that the total number of child solution-nodes $\check{N}$ produced to schedule a given number of jobs depends on PAR and L in addition to the properties of the jobs such as the variability in service times of the jobs. For Case NS.1 in Table 5.1, the total numbers of child solution-nodes $\check{N}$ produced to schedule 100,000 jobs at an arrival rate of 0.014 requests per minute are shown in Figure 5.14. The figure shows that comparatively, a large number of solution-nodes $\check{N}$ is produced for PAR values between 0.4 and 0.6. However, even for these values of PAR, with the workload parameters used in the experiments, the average number of solution-nodes produced was less than 1 node per job. Maximum numbers of solution-nodes open at a given time $\check{N}_{max}$ for 100,000 jobs with an arrival rate of 0.014 requests per minute and with uniformly distributed service times never exceeded 21.

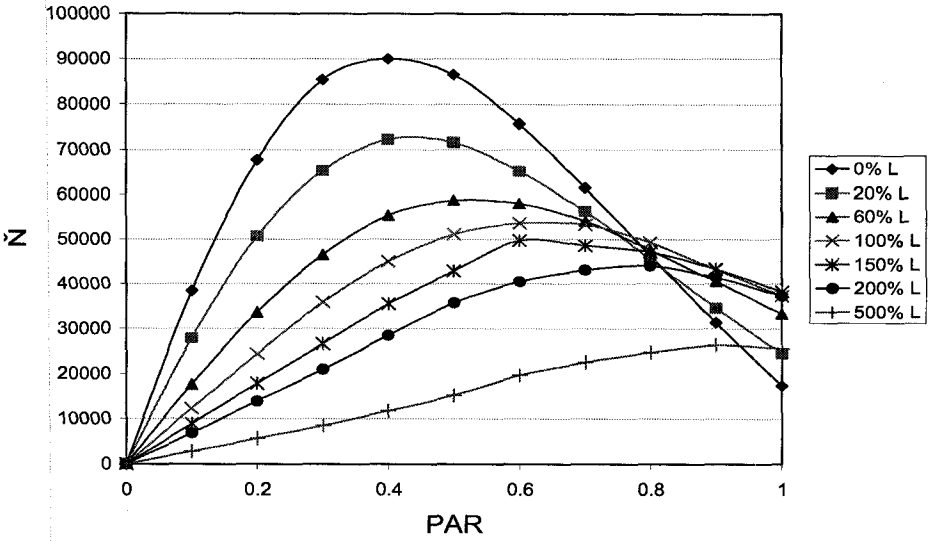


Figure 5.14: Effect of PAR on the Total Number of Child Solution-Nodes Ň Produced to Schedule 100,000 Tasks for Case NS.1

The simulator for conducting experiments with SSS was written in JAVA (JDK 1.4.2) and was executed on a Pentium IV machine, with 512 MB of RAM, running RedHat Linux 7.2. For uniformly distributed service times of the jobs with an arrival rate of 0.014 requests per minute, the simulator took on an average 24 seconds to schedule 100,000 jobs. This converts to per job 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 each job is even smaller. Since in a Grid environment each job on average is of the order of at least few minutes, it can be concluded that dynamic scheduling of jobs with the SSS algorithm is feasible in Grids even for a fairly high arrival rate.

Comparatively, a larger number of solution-nodes Ň was produced for the hyper-exponentially distributed service time. Nevertheless, even for such a distribution, maximum numbers of solution-nodes open at one time $\check{N}_{max}$ for 100,000 jobs never

exceeded 120. This shows that $\check{N}_{max}$ and $\check{N}$ produced by the SSS algorithm are much smaller than those of other algorithms [MCM75, XU90].

5.4.2. Optimality

The quality of the SSS algorithm can be assessed by comparing it with an optimal scheduler. An optimal scheduler is one that can always find a schedule if one exists. However, since Problem 1 (defined in Section 3.3.2.1) is an NP-Complete problem, it is computationally expensive to obtain an optimal solution for a reasonably large number of jobs. Recently, there has been a theoretical analysis of Problem 1 in [MAJ07], where the authors have established an un-schedulability criterion. If a given set of requests satisfy the un-schedulability criterion then they can not be scheduled under *any* scheduling policy without violating the deadline of at least one of the jobs [MAJ07]. With the help of this criterion, the authors have been able to obtain a lower bound on P_b for a given set of jobs. In this section, we compare P_b obtained with the SSS algorithm for a given set of jobs with the lower bound on P_b obtained with the help of the un-schedulability criterion for the same set of jobs. If P_b obtained with the SSS algorithm is close to the lower bound on P_b, it will demonstrate that the SSS algorithm results in near optimal schedules. Note however that the lower bound on P_b obtained with the un-schedulability criterion may be lower than that can be obtained with an optimal solution. Hence, even if P_b obtained with the SSS algorithm is slightly higher than that obtained with the un-schedulability criterion, this will not imply that the schedule obtained with the SSS algorithm is not optimal.

The results obtained for Case NS.1 with 50 jobs for different values of PAR and L are shown in Figure 5.15. The figure clearly shows that irrespective of PAR and L, P_b

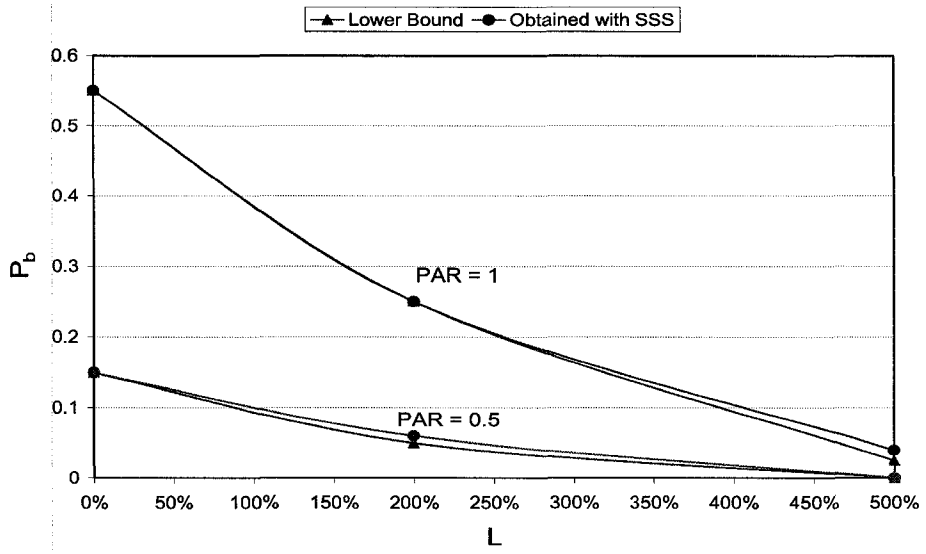


Figure 5.15: Comparison of P_b Obtained with the SSS Algorithm with the Lower Bound on P_b

obtained with the SSS algorithm is either equal to or is very close to the lower bound on P_b. These tests thus show that for a wide range of workload parameters, the SSS algorithm produces at least near optimal, if not optimal, schedules.

5.5. Summary

This chapter presented the SSS algorithm for an NP-Complete problem of scheduling advance reservation and on-demand requests on non-shared Grid resources. Basic idea behind SSS is suitable selection of a subset of jobs and using a pruned branch and bound technique to obtain the solution. SSS supports both non-preemptable and preemptable jobs including the cases where overheads are associated with job preemption. The results show that SSS is a scalable algorithm and dynamic scheduling of jobs with the SSS algorithm is feasible in Grids even for a fairly high arrival rate.

Performance results obtained with SSS brings important insights into the dynamics of the system. The results demonstrate that under-constraining ARs through the introduction of laxity is effective in maintaining high system performance while meeting

QoS constraints of applications. The results show that 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 jobs, a resource can thus limit the proportion of requests it accepts as ARs equal to the value of PAR corresponding to the knee of the curve.

The effect of data segmentation on system performance is investigated. The results show that segmentation is effective in achieving high system performance for scenarios where the service times of the jobs have high variability and/or PAR is high. The results also show that laxity can be exchanged for data segmentation to achieve high utilization. Results in Section 5.3.3.1 show that changing the minimum time between the arrival of an AR and its earliest start time does not significantly affect performance.

The simulation results demonstrate the effectiveness of the Deadlines Policy for preventing starvation of OD jobs. It can substantially decrease the response time of OD jobs without significantly affecting utilization.

Chapter 6

QoS Aware Scheduling on Shared Grid Resources

6.1. Scheduling Problem on Homogeneous Shared Resources

As discussed in Section 3.3.2, the performance of individual resources and hence that of a Grid system as a whole depends largely on the effectiveness of the scheduler deployed to schedule on-demand and advance reservation requests. The scheduler should ensure that the QoS constraints of all jobs accepted will be met. On shared resources, in addition to determining a suitable scheduled-time (defined in Section 5.2) for each job one has to determine a subset of nodes in a resource for running each job in such a way that the utilization of the resource is maximized. Moreover, various tasks of a job may need to communicate during execution and hence the algorithm should allow for gang scheduling of tasks. The problem of scheduling on-demand and advance reservation requests with laxity on homogeneous shared resources has been formally defined in Section 3.3.2.3. Since the problem is an NP-Complete problem and since a large number of jobs are possible in a Grid environment, scalability with respect to the number of requests is one of the desired characteristics of the scheduling algorithm.

As discussed in Section 2.6.1.1 most of the algorithms presented in the real-time domain for solving similar problems make certain assumptions (such as all jobs are known a priori, tasks of the jobs are independent, jobs are periodic and/or completely preemptable) that may not be true for the Grid domain. This thesis presents the Grid Scheduling with Deadlines (GSD) algorithm for scheduling OD requests and ARs with laxities on homogeneous shared resources. GSD is a scalable algorithm and can handle non-preemptable and preemptable jobs. GSD is adaptable to various workload conditions

and it can gang schedule all tasks of the jobs to allow for inter-task communication. GSD is discussed next.

6.2. Grid Scheduling with Deadlines (GSD) Algorithm

At the arrival of every new job, GSD seeks to obtain a feasible schedule for the set of jobs already in schedule and the new job. If a feasible schedule is not found, the job is rejected. Otherwise, if the given resource is selected by MMC for that job, the job is added to the set of scheduled jobs. As each job finishes executing on the resource, it is removed from the set of scheduled jobs. It can also be adapted for independent scheduling of tasks where tasks of the jobs are not needed to be gang scheduled. A mix of gang and independent scheduling is also possible with GSD.

A brief high-level description of the algorithm for non-preemptable gang-scheduled jobs is presented next. Detailed algorithm for both non-preemptable and preemptable gang-scheduled jobs is presented in Section 6.2.1. Independent scheduling with GSD is discussed in Appendix B.1.

Step 1: Whenever a new job arrives, determine the list A of ARs and OD requests to be scheduled and order them using a job-ordering heuristic.

Step 2: Remove the first job j_i from the ordered list A and determine its scheduled-time on each node of the resource. Find the earliest scheduled-time of the job after t_i where it can be allocated at least its required number of nodes.

Step 3: If by scheduling job j_i at the scheduled-time determined in Step 2, it cannot be completed before its deadline reject the new request, keep the previous schedule and terminate the algorithm.

Step 4: Determine the set of nodes that can schedule job j_i at its scheduled-time determined in step 2.

Step 5: If the number of nodes found in Step 4 is greater than the size of the job j_i, use a node-selection heuristic to choose the nodes to schedule the job on. Schedule j_i on the selected nodes at the determined scheduled-time.

Step 6: If list A is not empty, go to Step 2. Otherwise, accept the new request and update the resource schedule.

The sequence of steps presented shows that whenever a new request arrives at a shared resource, GSD first orders all ARs previously committed and all OD requests in queue using a particular *job-ordering heuristic* such as the earliest-deadline-first and the earliest-start-time-first and stores them in a list A. It then picks up the first job from this list A and computes the earliest scheduled-time of the job on the resource such that the job starts executing after its earliest start time and it can be allocated its required number of nodes for the complete duration of its execution. Since all tasks of the jobs are gang scheduled, the scheduled-time of the job is also equal to the scheduled-time of each of its tasks. Once the scheduled-time of the job has been determined, a subset of nodes of the resource that can schedule the tasks of the jobs at the determined scheduled-time without any schedule conflicts is obtained. If the size of this subset is greater than the number of nodes required by the job, the required number of nodes is selected from this subset. This selection is done using a particular *node-selection heuristic* such as first-fit and best-fit. The process is repeated for all jobs in A. If all jobs in A can be scheduled without violating the deadline of any of the jobs, the new job is accepted. Note that in Step 3, if any of the previously committed jobs cannot be scheduled without violating its deadline, the *new job* is rejected. In that case, the resource keeps its previous schedule.

With preemptable jobs the complexity of the problem grows. This is because while preemption helps improve resource utilization by utilizing small idle segments in resource schedule, uncontrolled preemption can increase system overheads due to context

switching. It may also result in division of each job into a large number of small sub-jobs. This may substantially increase the runtime of the scheduling algorithm used to schedule the jobs. Moreover, when overheads are associated with preemption, scheduled-time computation and node selection in GSD should be done in such a way that unnecessary preemption is avoided. The complete GSD algorithm presented next prevents uncontrolled preemption of jobs and the resulting overheads.

6.2.1. Grid Scheduling with Deadlines (GSD) Algorithm for Non-Preemptable and Preemptable Gang-Scheduled Jobs

The GSD algorithm can be given as a sequence of following steps. Each step is followed by an explanation.

Step 1: *Declare and initialize mAT equal to the value provided as a parameter to the algorithm. If no such parameter is passed, use the default value of mAT. Whenever a new request arrives, store all OD requests in queue and all ARs previously committed in a list A. Add the new request to A. Declare and initialize an array rM of integers with the size equal to the number of nodes in the resource. For each node x of the resource, set rM[x] equal to the current system time.*

This step declares and initializes some of the variables for the algorithm. Different segments of a preemptable job can be scheduled at different times. mAT represents the size of the smallest allowed segment of a preemptable job. If mAT is very small, substantial overheads are imposed on the algorithm and its completion time increases. Also, because of the overheads involved in job preemption and its resumption, it does not make sense to run a preemptable job on the resource for a very short time before preempting it. mAT thus sets the limit on the minimum time for which the job would be allocated a resource at a time. If an idle segment in the resource schedule is smaller than

mAT, that segment is not allocated to any job. Note that if mAT is large, idle segments in the resource schedule would result. This would decrease the utilization of the resource. Thus, mAT should be carefully selected according to the needs of the system.

The array rM[x] represents the time at which the last job scheduled on node x is expected to finish its execution. For the new schedule of the resource with the new request, rM is initialized with the current system time since none of the jobs have been scheduled on any of the nodes so far. List A represents the total set of all jobs that needs to be scheduled.

List A represents the total set of all jobs that needs to be scheduled.

Step 2: *Order the jobs in list A using the job-ordering heuristic provided as a parameter to the algorithm. If no such parameter is provided, choose the default (earliest-deadline-first) heuristic.*

This step orders the jobs in list A according to a particular heuristic. This research has investigated several job-ordering heuristics such as the earliest-deadline-first (EDF), earliest-start-time-first (ESF), least-laxity-first (LLF) and earliest-arrival-first (EAF). The jobs are picked up sequentially from list A for scheduling. Therefore, job-ordering heuristic can significantly improve the optimality of the schedule.

Step 3: *Initialize two arrays eSF and wS with sizes equal to the number of nodes of the resource. Remove the first job j_i from list A. If the job j_i is non-preemptable, go to Step 4. Otherwise, go to Step 5.*

The array variable eSF[x], declared in Step 3, will contain the earliest feasible scheduled-time (feasible for the node) of the job on node x. wS[x] will contain the size of the scheduled-time window that corresponds to the maximum delay from time eSF[x] before which the task of the job must start executing on the node x otherwise it would not be able to complete before the scheduled-time of the task of some other job on that node.

Step 4: *For each node x of the resource, determine the earliest scheduled-time of job j_i and its corresponding window size and store them in eSF[x] and wS[x] respectively. If t_i > rM[x], then set eSF[x] = rM[x] and wS[x] equal to infinity. If t_i < rM[x], then find the earliest segment after t_i in the node schedule during which the node is idle and that has a size $\hat{s}$ such that $\hat{s} >= e_{E,ib}$. Set eSF[x] equal to the start time of the segment and $wS[x] = \hat{s} - e_{E,ib}$. If no such segment exists, set eSF[x] = rM[x] and wS[x] equal to infinity. Go to Step 6.*

If t_i of a non-preemptable job j_i is greater than the time at which the last job scheduled on node x is expected to complete, job j_i can start executing on that node at its earliest start time and since there is no scheduled job on node x after t_i there is no chance of a schedule conflict and hence wS[x] is set to infinity. On the other hand, if t_i is less than rM[x], a suitable idle segment in the node schedule with size greater than $e_{E,ib}$ is found. Since there are other jobs scheduled on node x after the segment, there is a limit on time before which job j_i must start executing on that node in order to avoid a schedule conflict. If no suitable segment is found, the feasible scheduled-time of job j_i on node x is set equal to the completion time of the last job scheduled on node x.

Step 5: *For each node x of the resource, determine the earliest scheduled-time of job j_i and its corresponding window size and store them in eSF[x] and wS[x] respectively. If t_i > rM[x], then set eSF[x] = rM[x] and wS[x] equal to infinity. If t_i < rM[x], then find the earliest segment after t_i in the node schedule during which the node is idle and that has a size $\hat{s}$ such that $\hat{s} >= mAT$. Set eSF[x] equal to the start time of the segment and $wS[x] = \hat{s} - mAT$. If no such segment exists, set eSF[x] = rM[x] and wS[x] equal to infinity.*

This step is similar to Step 4 for non-preemptable jobs. The only difference is that the size of an idle segment does not have to be equal to or greater than $e_{E,ib}$. As long as it is equal to or greater than mAT, a segment of preemptable job j_i can be scheduled.

Step 6: *Based on eSF and wS for each node, determine the earliest feasible scheduled time of job j_i where at least s_{ib} nodes can start the job and store the determined time in a variable schTime.*

This step finds the earliest feasible scheduled time of the job j_i.

Step 7: *Check whether schTime + $e_{E,ib}$ <= d_i. If it is not, go to Step 11. Otherwise, if job j_i is non-preemptable go to Step 8. On the other hand, if job j_i is preemptable, go to Step 9.*

This step checks whether the scheduled time determined for job j_i in step 6 results in the completion of the job before its deadline.

Step 8: *Find out nodes which are idle from schTime to schTime + $e_{E,ib}$. If the number of such nodes is greater than s_{ib}, choose s_{ib} nodes among them by using a node-selection heuristic provided as a parameter to the algorithm. If no such parameter is provided, use the default (best-fit) heuristic. Schedule job j_i on the chosen s_{ib} nodes from schTime to schTime + $e_{E,ib}$. Set rM[x] = schTime + $e_{E,ib}$ for each of the chosen nodes for which rM[x] < schTime + $e_{E,ib}$. Go to Step 12.*

If job j_i can be successfully completed before its deadline by scheduling it at scheduled-time determined in step 6, nodes that can start the job at the determined scheduled-time without any schedule conflicts are found. If the number of found nodes is greater than the required number of nodes, node-selection heuristic is used to obtain the required number of nodes. This research has investigated several node-selection heuristics such as best-fit (BF), first-fit (FF) and worst-fit (WF). BF and WF asses the degree of fitness of job j_i on a certain node by calculating the size of idle segments in the node schedule before the scheduled-time and after the completion time of job j_i. First-fit chooses the first s_{ib} nodes without assessing the degree of fitness.

Step 9: *Find out a set of nodes Y which are idle from schTime to schTime + mAT. If the size of Y is greater than s_{ib}, determine for each node y in Y duration t that*

corresponds to period between schTime and time after schTime at which the node has next job scheduled. If there are at least s_{ib} nodes in Y for which $t > e_{E,ib}$, go to Step 8. Otherwise, arrange the nodes in Y in the decreasing order of t and store their t's in an array pTA such that first element in pTA corresponds to first node in Y and so on. If $pTA[s_{ib}] <= o$, go to Step 10. Here o is the overhead in terms of time associated with preemption and resumption of job j_i. Otherwise, declare preemptionTime and set preemptionTime = schTime + $pTA[s_{ib}]$. Schedule job j_i among the first s_{ib} nodes in Y from schTime to preemptionTime – o. Set rM[x] = preemptionTime for each of the first s_{ib} nodes in Y for which rM[x] < preemptionTime. Create a new job j_i' with all the parameters same as job j_i but with $t_{i'}$ = preemptionTime + 1 and $e_{E,i'b} = e_{E,ib} - pTA[s_{ib}] + o$. Insert job j_i' as the first element in list A. Go to Step 12.

If preemptable job j_i can be scheduled without a need to preempt it, the algorithm goes to Step 8 to find suitable nodes for the job. Otherwise, among the nodes that can successfully schedule job j_i at the scheduled-time determined in step 6, the required s_{ib} nodes are chosen in such a way that preemption of the job is delayed as much as possible. Doing so results in lesser average number of preemptions per job and hence smaller preemption overheads. The unscheduled segment of preemptable job j_i is added back to the head of list A so that it can be scheduled.

Step 10: *Create a new job j_i' with all the parameters same as job j_i but with $t_{i'}$ = schTime + 1. Insert job j_i' as the first element in list A. Go to Step 14.*

If the scheduled-time of job j_i as determined in step 6 results in execution of the job for less time than required to preempt and resume it later, the job is not scheduled at the currently determined scheduled-time. Instead, the job is added back to the head of list A to find a new scheduled-time for it.

Step 11: *Reject the new job and terminate the algorithm.*

If any of the jobs in A can not be completed before its deadline, the new request is rejected and the resource keeps its previous schedule.

Step 12: *If list A is not empty, go to Step 3. Otherwise, accept the new job and terminate the algorithm.*

All jobs in list A are picked up one-by-one and scheduled on the resource. If all jobs in list A are scheduled successfully such that all their constraints are met, the new job is accepted.

Note that there can be different variants of GSD corresponding to the different combinations of various job-ordering and node-selection heuristics. GSD thus can be easily adapted to a particular workload by plugging different job-ordering and node-selection strategies. This research has investigated several heuristics for job-ordering including earliest-deadline-first (EDF), least-laxity-first (LLF), earliest-start-time-first (ESF) and earliest-arrival-time-first (EAF). In EDF jobs are arranged in a non-decreasing order of their deadlines. Similarly, in LLF, ESF and EAF jobs are arranged in a non-decreasing order of their laxities, earliest start times and arrival times, respectively. This research has investigated a number of heuristics for node-selection including best-fit (BF), first-fit (FF) and worst-fit (WF). BF selects those nodes on which the allocation of a job results in the smallest size fragments. WF selects those nodes on which job allocation results in the largest size fragments while FF selects the first available nodes that can start the job at the determined scheduled-time.

6.3. Performance Analysis of Resource Liaison and Controller with Homogeneous Shared Resources

Table 6.1 presents a small subset of experiments, conducted with RLC with homogeneous shared resource model, which will be presented in this chapter. The

experiments in this chapter assume that estimated runtimes of the jobs are equal to their actual runtimes. A large number of experiments were conducted with RLC with homogeneous shared resources for scenarios where the exact runtimes of jobs are not known *a priori*. All such experiments are presented in Chapter 7.

The setup for the experiments conducted with RLC with homogeneous shared resources has been discussed in detail in Section 4.1.1 while the workload model has been discussed in Section 4.3.2. The values of the parameters for each experiment set are given in Table 6.1. GSD is used as the scheduling algorithm in these experiments.

Case S.1 studies the impact of laxity on system performance when all jobs submitted to the system reserve the resource in advance. Case S.2 investigates system performance when the workload consists of a mix of advance reservation and on-demand requests. Case S.3 studies the impact of arrival rate on system performance.

Table 6.1: A Subset of Cases Studied with RLC with Homogeneous Shared Resources

Case No.	PAR	L	Starvation Prevention		Maximum Possible Utilization with a Given Arrival Rate
			SPM Active	Deadline of OD Requests	
S.1	1	50% to 1000%	--	--	100%
S.2	0.5	50% to 1000%	Yes	2 days	100%
S.3	1	400%	--	--	50% to 125%

6.3.1. Impact of Job-Ordering and Node-Selection Heuristics

This section presents the results obtained for the S.1 case in Table 6.1. As expected, the results show that as laxity increases, percentage of work rejected W_R decreases (Figure 6.1) while utilization U increases (Figure 6.2). The figure also shows that the combination of the earliest-deadline-first and the best-fit heuristics (EDF-BF) in the GSD scheduling algorithm results in the lowest W_R and the highest U for all values of L. The high performance of EDF-BF can be attributed to the efficient ordering of jobs by

EDF that seeks to meet the deadline of as many jobs as possible and the efficient allocation by BF that seeks to minimize fragmentation in the resource schedule. LLF-BF, EDF-FF and LLF-FF results in performance close to that of EDF-BF. Although not shown in the figures to avoid cluttering, the performance of LLF-WF is close to that of EDF-WF and that of ESF-FF and ESF-WF is close to that of ESF-BF. As can be seen, the

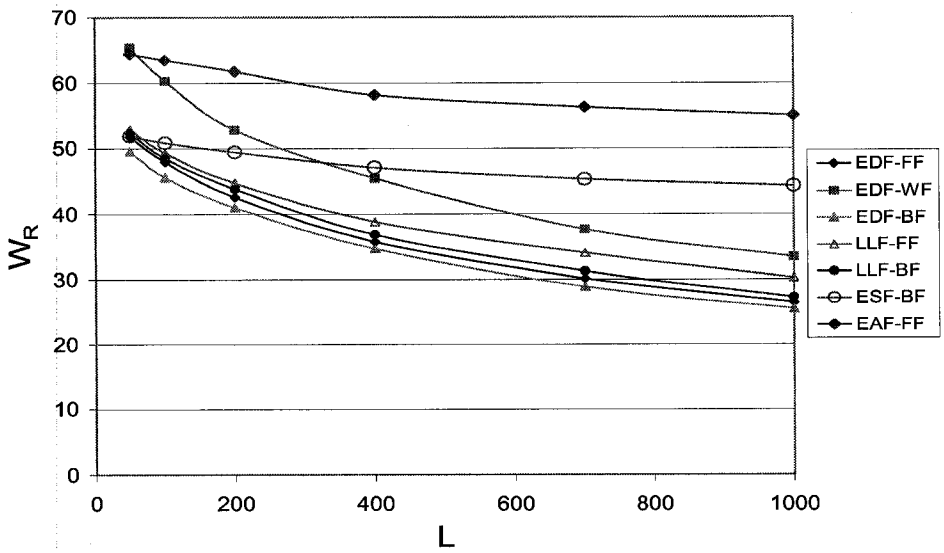


Figure 6.1: Impact of Job-Ordering and Node-Selection Heuristics on Work Rejected for Case S.1

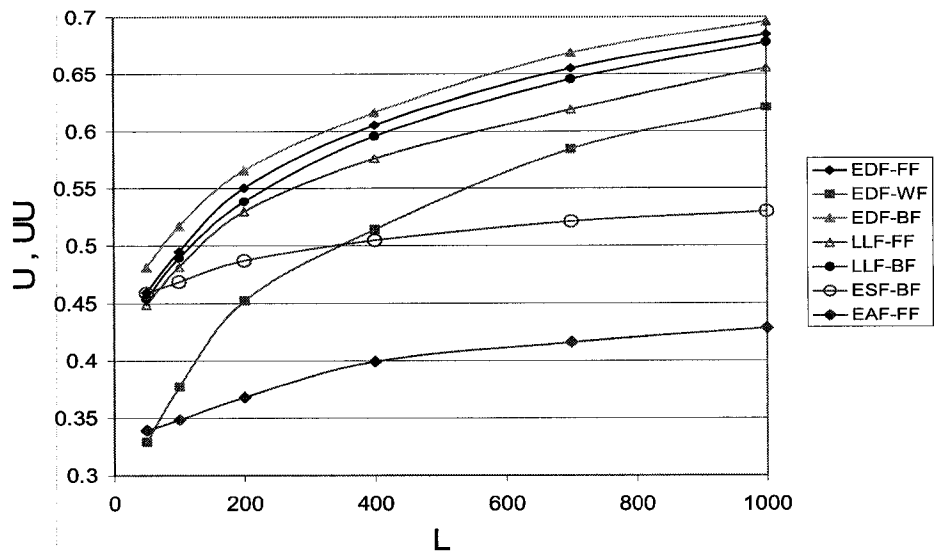


Figure 6.2: Impact of Job-Ordering and Node-Selection Heuristics on Utilization for Case S.1

ESF and EAF heuristics for job ordering and the WF heuristic for node selection generally result in high W_R and low U. As there was no error in user-estimated runtimes in Case S.1, no jobs are aborted by the resource and hence $W_A = 0$, $P_A = 0$ and U = UU for all values of L.

Figure 6.3 shows that the ESF heuristic for ordering the jobs results in the lowest

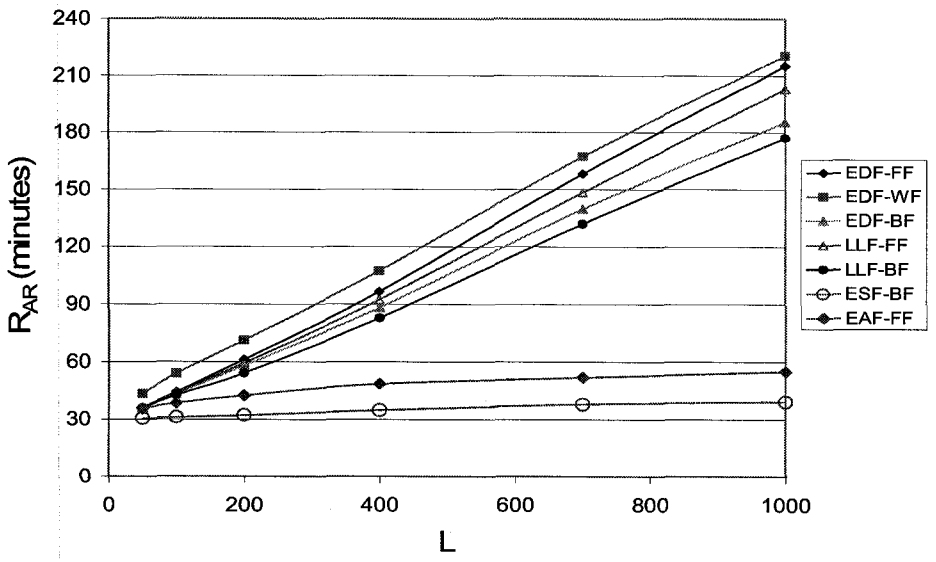


Figure 6.3: Impact of Job-Ordering and Node-Selection Heuristics on Response Time of Advance Reservations for Case S.1

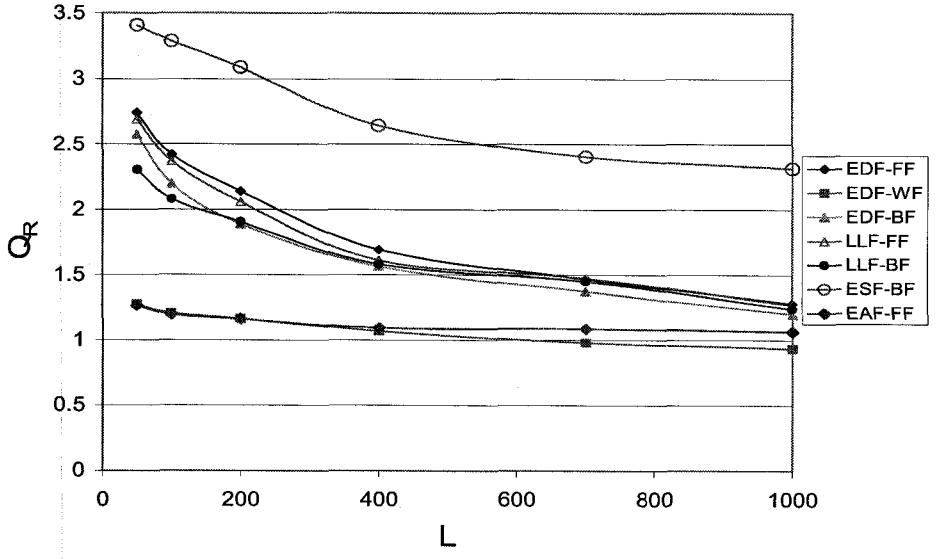


Figure 6.4: Impact of Job-Ordering and Node-Selection Heuristics on Fairness for Case S.1

R_{AR} values. This is because with the ESF heuristic, jobs with earlier start times are given priority in the schedule and hence the mean wait time of the jobs decreases. In addition, ESF has significantly lower U than other heuristics and hence results in comparatively less contention for the resource. LLF and EDF heuristics with high utilizations results in higher R_{AR}.

Figure 6.4 shows that when L is small, for all heuristics apart from EDF-WF and EAF-FF, Q_R is many times higher than 1. This shows that a larger proportion of 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_R approaches 1 showing that the probability of rejecting a large and a small job almost becomes equal. As WF results in large fragments in resource schedules, it is better able to accommodate large jobs compared to the BF heuristic that seeks to minimize the size of the fragments. Hence, the combination of WF node-selection heuristic results in lower Q_R than the corresponding combination of BF or FF node-selection heuristic with the same job-ordering heuristic. Since EAF-FF results in substantially higher W_R and significantly lower U, it leaves enough capacity in the resource schedule to accommodate large jobs. Hence, Q_R obtained with EAF-FF remains close to 1 for all values of L.

P_b curves (not presented to save space) show that except for WF based heuristics, P_b does not change significantly with the increase in L. It can be shown that $P_b = W_R / (100^* Q_R)$ and hence as both W_R and Q_R decrease at almost the same rate with the increase in L, P_b tends to remain constant. For WF based heuristics where W_R decreases at a much higher rate than Q_R, P_b decreases with the increase in L.

6.3.2. Effect of Mixed Workload

This section presents results obtained by using a mix of advance reservation and on-demand (OD) requests. The results correspond to Case S.2 in Table 6.1.

Figure 6.5 and Figure 6.6 show that W_R and U curves obtained for this case have trends similar to those that were obtained in Figure 6.1 and Figure 6.2 respectively.

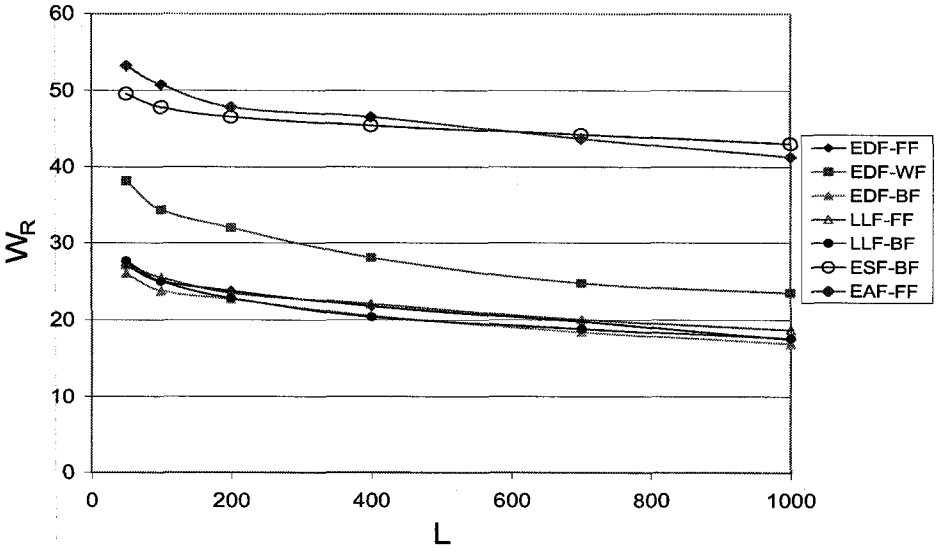


Figure 6.5: Impact of Job-Ordering and Node-Selection Heuristics on Work Rejected for Case S.2

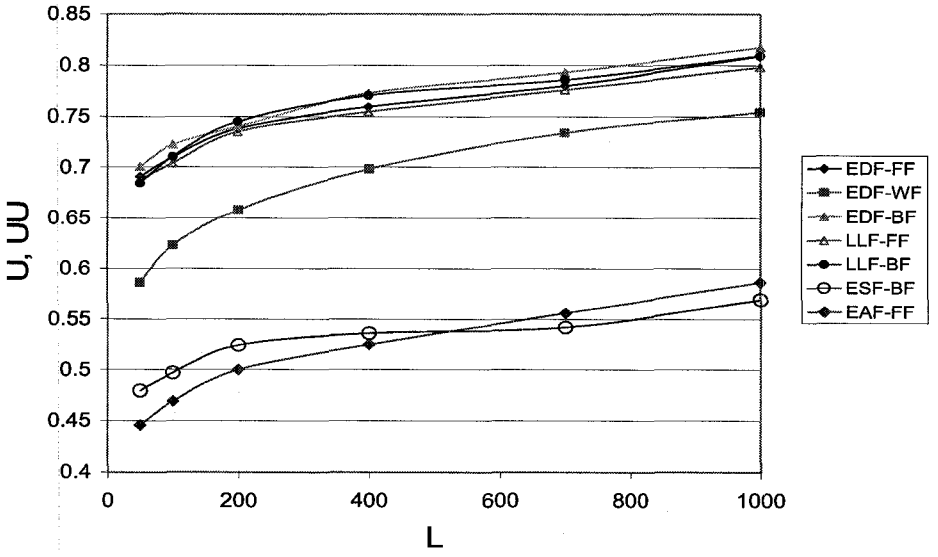


Figure 6.6: Impact of Job-Ordering and Node-Selection Heuristics on Utilization for Case S.2

However, the results show that for the same values of the parameters, W_R for PAR = 0.5 is significantly lower (upto 27% lower) while U is substantially higher (upto 25% higher) than that obtained for PAR = 1. This is because OD requests have less stringent timing constraints and hence provide more flexibility in scheduling. For most of the heuristics, this difference in W_R and U decreases with the increase in L.

Figure 6.7 presents curves similar to those obtained in Figure 6.3. Trends of Q_R and P_b (not shown) are similar to those obtained in Section 6.3.1. But the results show that for the same values of the parameters significantly lower probability of blocking is obtained when OD requests are introduced in the workload.

Since the ESF heuristic results in lower wait time for the jobs as well as lower utilization of the resource, it results in lower response time for the on-demand jobs as shown in Figure 6.8. As there was no error in user-estimated runtimes in Case S.2, no jobs are aborted by the resource and hence $W_A = 0$, $P_A = 0$ and U = UU for all values of L.

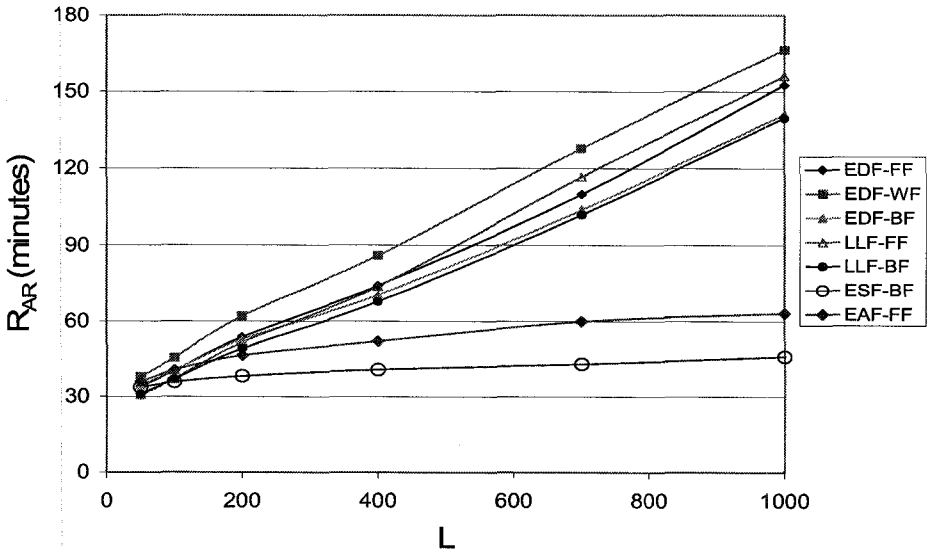


Figure 6.7: Impact of Job-Ordering and Node-Selection Heuristics on Response Time of Advance Reservations for Case S.2

6.3.3. Impact of Arrival Rate

This section studies the impact of arrival rate on system performance. The results presented in this section correspond to Case S.3 in Table 6.1. In these experiments, the arrival rate is varied to vary the maximum possible utilization of the system. A higher arrival rate corresponds to a higher maximum possible utilization and vice versa. System

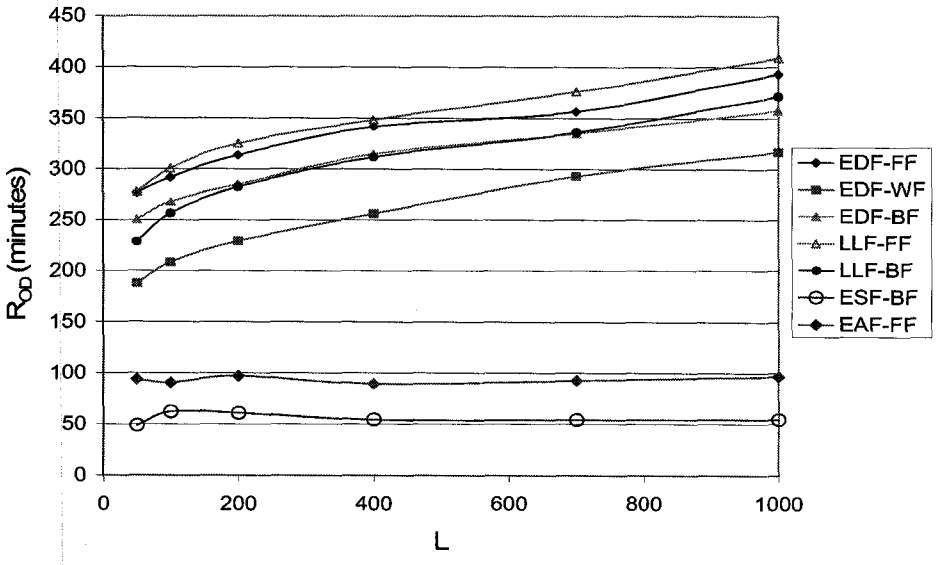


Figure 6.8: Impact of Job-Ordering and Node-Selection Heuristics on Response Time of On-Demand Requests for Case S.2

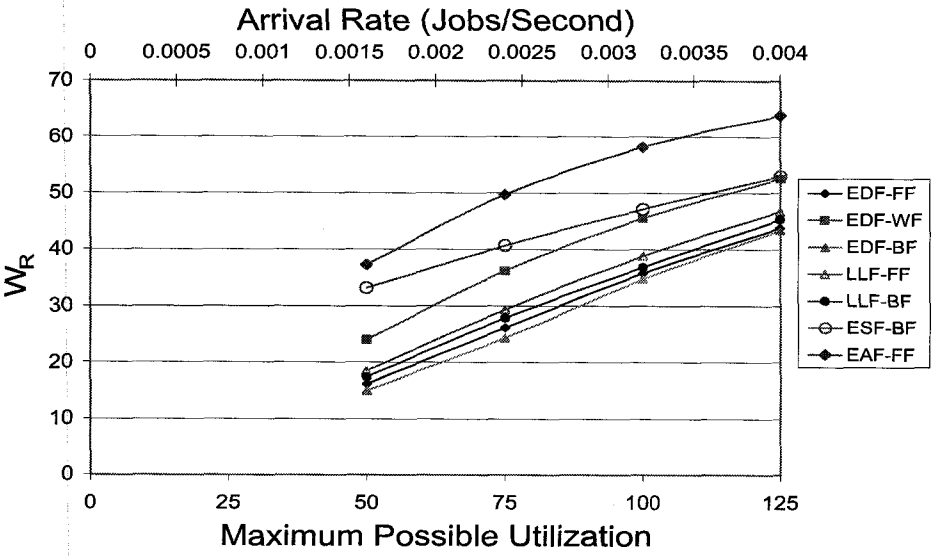


Figure 6.9: Impact of Arrival Rate on Work Rejected for Case S.3

behavior is investigated at different maximum possible utilization values.

Figure 6.9 shows that when arrival rate is increased, W_R increases. A higher arrival rate means that the system is subjected to a higher total work and since the system has only a limited capacity, an increase in arrival rate results in an increase in W_R. As the slopes of W_R curves in Figure 6.9 are less than 1, this shows that as the number of arrivals increases not all the new jobs are rejected by the system. This result is confirmed by Figure 6.10 which shows an increase in U with an increase in arrival rate. As arrival rate increases, the system is able to accept more jobs. This is because given a higher number of job arrivals there is a higher probability of fitting one of the jobs in one of the idle segments of resource schedule. However, after a certain point the system tends to saturate and the rate of increase in U with an increase in the arrival rate decreases. The results also show that irrespective of the maximum possible utilization of the system, EDF-BF results in the lowest W_R values and the highest U values.

As the system utilization increases with an increase in arrival rate so does the contention for the resource. Figure 6.11 thus shows an increase in R_{AR} with an increase in arrival rate. As in Section 6.2.1 and Section 6.2.3, ESF-BF and EAF-FF results in the lowest R_{AR} values because of the reasons discussed in Section 6.2.1.

Figure 6.12 shows that with an increase in arrival rate Q_R decreases and approaches a value of 1. At a lower arrival rate a higher number of large jobs are rejected by the system (see Section 6.2.1) as compared to small jobs. As the arrival rate increases, the system tends to saturate and a higher number of small and large jobs are rejected by the system. This increases the proportion of small jobs in the total jobs rejected and hence Q_R decreases.

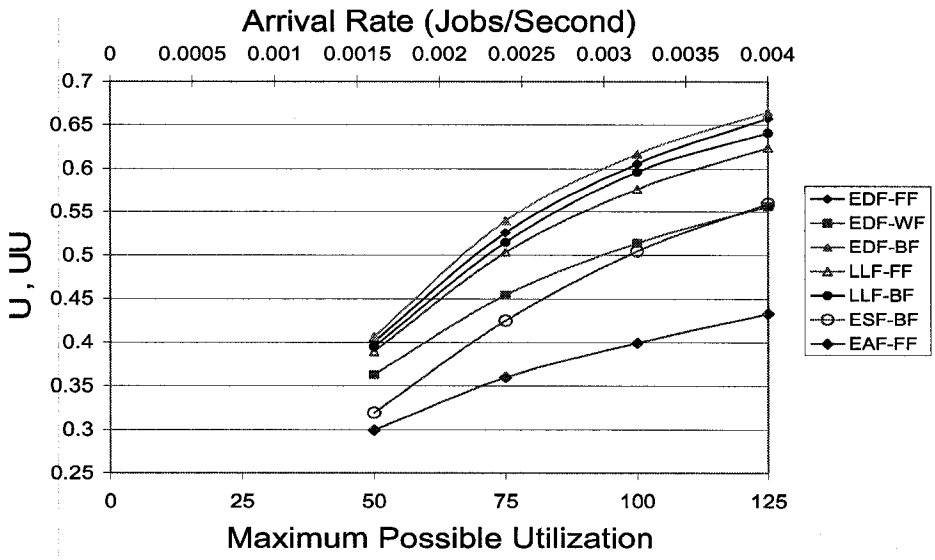


Figure 6.10: Impact of Arrival Rate on Utilization for Case S.3

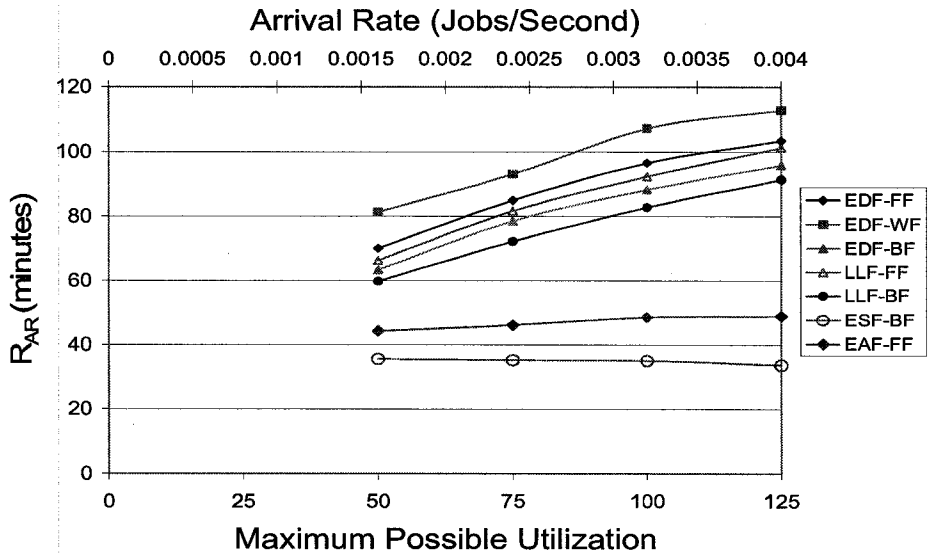


Figure 6.11: Impact of Arrival Rate on Response Time of Advance Reservations for Case S.3

6.4. Performance of the Grid Scheduling with Deadlines Algorithm

6.4.1. Scalability

Since Problem 2 (defined in Section 3.3.2.3) is an NP-Complete problem, scalability with respect to the number of jobs is one of the desired characteristics of the scheduling algorithm. This section analyses the worst case time complexity of the

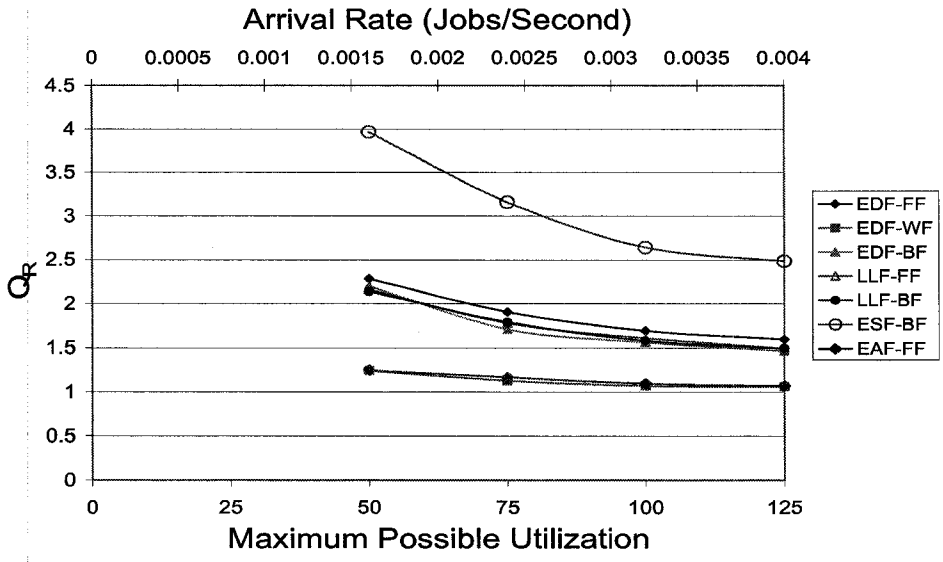


Figure 6.12: Impact of Arrival Rate on Fairness for Case S.3

heuristics-based GSD algorithm for gang-scheduled non-preemptable jobs and proves that it is of the order of n^2 where n is the number of jobs.

The GSD algorithm as presented in Section 6.2 can be divided into three major high-level operations.

A. Ordering of jobs (Step 2 in Section 6.2.1).

B. Determination of a suitable scheduled time for each job (Step 4 and Step 6 in Section 6.2.1).

C. Determination of suitable nodes for running each job (Step 8 in Section 6.2.1).

Given n jobs, operation A is performed only once while operations B and C are repeated for each of the n jobs. Since operation A is a sorting operation, one can use any of the well-known sorting algorithms to perform operation A. Many sorting algorithms such as merge-sort, heap-sort and smooth-sort have the worst case complexity of

n*log(n). If any of these sorting algorithms is used, the worst case time complexity **T** of operation A can be given as:

$$T\,[Operation\ A] \quad = \quad O(n*log(n)) \qquad \textbf{(6.1)}$$

Operation B involves determining the possible times at which a task of the job j_i can execute on a particular node. This is determined in terms of one or more scheduled-time windows on each node such that if a task of job j_i is scheduled on the node at any time in the scheduled-time window, it can be completed before a task of another job is scheduled to execute on that node. A node schedule consists of busy segments during which a job is scheduled to execute and idle segments during which no job is scheduled. As the scheduled-time windows can only exist in the segments of schedule during which the node is idle, the complexity of this operation is directly proportional to the number of idle segments in a node schedule. It can be shown that whenever a new job is scheduled on the node, the number of idle segments in the node increases by at the most 1. When the first job arrives, the node schedule is one idle segment. In the worst case, every new arrival is scheduled on a node and hence the number of idle segments in that node increases by 1 for every job arrival. Hence, at the arrival of the i^{th} job there can be a maximum of i idle segments in the node. Thus, for n jobs the complexity of determining the scheduled-time windows on a given node can be given as the following sum:

$$T\,[Determining\ Scheduled\ Time\ Windows\ on\ a\ Given\ Node\ for\ n\ Jobs]$$

$$= \quad k(1+2+3+...+n) \quad = \quad k*n(n+1)/2 \qquad \textbf{(6.2)}$$

$$= \quad O(n^2) \qquad \textbf{(6.3)}$$

Since the scheduled time windows need to be determined on each of the ω nodes in the resource. The complexity of operation B can be given as:

$$T\,[Operation\ B] \quad = \quad O(\omega*n^2) \qquad \textbf{(6.4)}$$

Operation C involves selecting s_{ib} nodes from the set of nodes that can start the job at the determined scheduled-time. This involves evaluating the degree of fitness of a given job, based on the node-selection heuristic, on each of the nodes that can start the job at the determined scheduled-time. In the worst case, all ω nodes in the resource can start a given job at the determined scheduled-time. Hence, the degree of fitness needs to be evaluated on each of the ω nodes. The worst case execution time of this process is thus directly proportional to ω. Once the degrees of fitness have been evaluated for all of the ω nodes, they are arranged in the non-increasing order of fitness and the first s_{ib} nodes are selected from the sorted degrees of fitness. As this process involves sorting ω values, the worst case complexity of this operation is of the order of ω*log(ω). The worst case time complexity of operation C for *n* jobs can thus be given as:

$$T\,[Operation\ C] \quad = \quad O(n^* \omega + n^* \omega^* log(\omega)) = O(n^* \omega^* log(\omega)) \qquad \textbf{(6.5)}$$

Note that Step 1, Step 3, Step 5, Step 11 and Step 12 in Section 6.2.1 involve operations that are either done once for scheduling *n* jobs or are repeated once for every job. The time complexity of these operations in the worst case is thus not more than O(n). The worst case time complexity of GSD for *n* jobs is hence equal to the combination of the worst case time complexities of each of the operations A, B and C, and O(n). It can be given as:

$$T\,[GSD] \quad = \quad O(n + n^* log(n) + \omega^* n^2 + n^* \omega^* log(\omega)) \qquad \textbf{(6.6)}$$

Note that for a given system ω is a constant and for many systems it is much smaller compared to *n*. For such systems, the worst case time complexity of GSD in Equation 6.6 can be reduced to:

$$T\,[GSD] \quad = \quad O(n^2) \qquad \textbf{(6.7)}$$

Equation 6.6 and Equation 6.7 show that GSD is a scalable algorithm and can be used for dynamic scheduling of a large number of jobs in Grids. This fact was confirmed by the experimental investigation of this research where GSD successfully scheduled thousands of jobs in a reasonable amount of time.

6.4.2. Optimality

The quality of the GSD algorithm can be assessed by comparing it with an optimal scheduler. An optimal scheduler is one that can always find a schedule if one exists. However, since Problem 2 (defined in Section 3.3.2.3) is an NP-Complete problem, it is virtually impossible to obtain an optimal solution for even a relatively small number of jobs. To the best our knowledge, there neither exists any algorithm that can find an optimal solution for problems similar to Problem 2 nor any other work for finding a near optimal solution. Even modeling the problem in genetic algorithms may not be computationally feasible.

To assess the quality of the schedule obtained with GSD, this research relies on a theoretical analysis of advance reservations with laxity on a single resource in [MAJ07], where the authors have established an un-schedulability criterion. With the help of this criterion, the authors have been able to obtain a lower bound on P_b for a given set of jobs as discussed in Section 5.4.2. This section compares P_b obtained with the GSD algorithm for a given set of jobs with the lower bound on P_b obtained for the same set of jobs. The only limitation of assessing the performance of GSD with this technique is that the analysis can be done only for the cases where ω is equal to 1. This is because the theoretical analysis in [MAJ07] is limited to only a single resource.

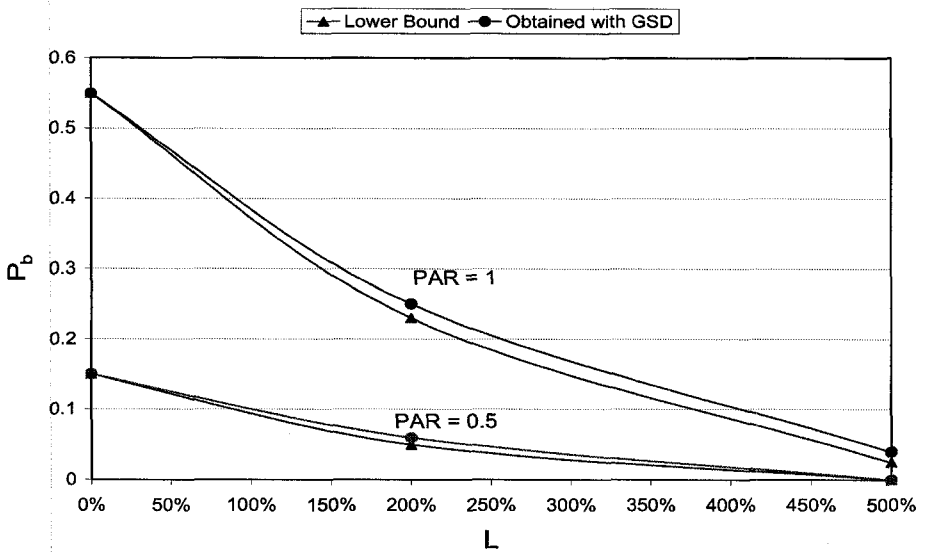


Figure 6.13: Comparison of P_b Obtained with the GSD Algorithm with the Lower Bound on P_b

If P_b obtained with the GSD algorithm is close to the lower bound on P_b, it will demonstrate that the GSD algorithm results in near optimal schedules. The results obtained with GSD for Case NS.1in Table 5.1 with ω equal to 1 and 50 jobs for different values of PAR and L are shown in Figure 6.13. The figure clearly shows that irrespective of PAR and L values, P_b obtained with the GSD algorithm is either equal to or is very close to the lower bound on P_b. These results thus show that for a range of workload parameters, the GSD algorithm produces near optimal schedules.

6.5. Summary

The chapter presented a novel heuristic algorithm, Grid Scheduling with Deadlines, for an NP-Complete problem of scheduling OD jobs and ARs with laxities on a shared resource. The basic idea behind GSD is the suitable selection of a heuristic for ordering the jobs and a heuristic for selection of nodes to allocate the tasks of the jobs. GSD can be configured with the help of pluggable strategies to adapt to various workload conditions and needs of the system. GSD is a scalable algorithm and can be used for

dynamic scheduling of a large number of jobs in Grids. GSD supports both preemptable and non-preemptable jobs. It also supports the gang scheduling paradigm.

The results show that for a wide range of parameters, a combination of EDF and BF heuristics in GSD gives the best performance in terms of the highest U and the lowest W_R. The results also show that with the increase in L not only W_R decreases and U increases but also the system becomes fairer.

Section 6.4.1 shows that if the number of nodes in a resource is small compared to the number of jobs to be scheduled then the worst case time complexity of GSD is of the order of n^2.

Chapter 7

Providing QoS Guarantees under Uncertain Runtimes of Jobs

7.1. Managing Abnormal Terminations of Jobs and Error in User Estimated Runtimes

As discussed in Section 2.6.1, inaccuracies in user-estimated runtimes and abnormal terminations of jobs that are not usually accounted for in theoretical analysis are unavoidable in a real-world setting. The analysis of real workloads for IBM SP2 machines in Section 4.3.3 also illustrated that there is usually a large error in user-estimated runtimes. As will be shown in Section 7.5, impact of such errors in estimation on performance can be very severe and thus one needs mechanisms for handling it effectively. This thesis presents a Schedule Exceptions Manager (SEM) and related components that can effectively handle abnormal termination of jobs and errors in user-estimated runtimes by adapting the resource schedule to the changing conditions. The research also presents a Pre-Scheduling Engine that is effective in preventing performance degradation especially in scenarios where the error in estimation is very high.

7.2. Schedule Exceptions Manager (SEM)

As will be shown in Section 7.5, abnormal terminations of jobs combined with inaccuracies in user-estimated runtimes can adversely affect the efficiency of the system. The goal of SEM is thus to prevent such performance degradations by adapting the resource schedule to the changing conditions.

As discussed in Section 3.3.2, the schedulability analysis at the time of admission is done using the estimated runtimes of jobs. However, SEM monitors the progress of the jobs in the resource schedule. When a job with overestimated runtime leaves earlier than expected or when a job terminates abnormally, SEM invokes the scheduler. SCH re-schedules the jobs to make use of the spare capacity resulting from the earlier departure of the job. On the other hand, when a job with underestimated runtime does not finish at its expected completion time, SEM consults Abortion Policy Block to determine the next action.

7.3. Abortion Policy Block (APB)

Abortion Policy Block (APB) governs policies to determine suitable actions when an underestimated job does not finish at its expected completion time. Different policies can be used with APB to suit the needs of the systems. For example, in some cases it might be desirable to terminate the job that does not finish at its expected runtime. This is to penalize such jobs and discourage underestimations. In some cases, however, some other low priority jobs may be terminated by the resource to allow room for high priority underestimated jobs. In some cases, the amount of work already done on a particular job might also become a factor in deciding whether or not to terminate the underestimated job. The thesis presents a policy called Feasibility Policy to deal with underestimated jobs. Feasibility Policy seeks to prevent unnecessary job abortions but if abortion is unavoidable it penalizes underestimated jobs.

7.3.1. Feasibility Policy (FP)

When a job with underestimated runtime does not finish at its expected completion time, instead of aborting the job, Feasibility Policy triggers the schedulability

analysis to determine if a small quantum of time τ can be allocated to the job without violating QoS constraints of the other jobs already committed. If SCH can find a feasible schedule, the new schedule is kept. Otherwise, the job is aborted. If the job still does not finish with the extra time, the job can be allocated further quanta if doing so does not result in violation of the QoS constraints of other jobs. Otherwise, the job is aborted.

Note that every time scheduler is invoked, overheads are imposed on the system. Thus, if the value of τ is very small compared to the actual error in underestimated runtime, more overheads will be imposed on the system by frequent analysis and re-scheduling. On the other hand, if the value of τ is larger than the actual error, the feasibility analysis might result in an infeasible schedule while a feasible schedule exists with the actual extension time required to complete the job. This would increase the amount of work aborted by the resource and hence decrease its *useful utilization* (defined in Section 4.2.8). Since the error in estimation is to an extent proportional to underestimated runtime of the job, FP selects τ to be some percentage σ of estimated runtime of the job $e_{E,ib}$ i.e. $\tau = \sigma * e_{E,ib}$.

If mean error in underestimated runtimes is some percentage θ of the mean estimated runtime E_E, then the average number of times χ underestimated jobs results in scheduler invocations prior their successful completion can be given as:

$$\chi = (\theta * E_E) / (\sigma * E_E) \quad \textbf{(7.1)}$$

$$\chi = \theta / \sigma \quad \textbf{(7.2)}$$

The tradeoff between overheads in frequent invocation of scheduler and unnecessary job abortion thus depends on the number χ which acts a tuning parameter for FP. In the framework presented by this thesis, SEM keeps tracks of current value of θ and hence FP can adjust σ accordingly. For the experiments, σ was adjusted such that value of

χ stays near 5. Such a value assures that the number of unnecessary job abortions stays low while minimizing overheads imposed on the system.

7.4. Pre-Scheduling Engine (PE)

The analysis on inaccuracies in user-estimated runtimes using logs collected for IBM SP2 machines presented in Section 4.3.3 shows that runtimes of the jobs are substantially overestimated by the users. This combined with abnormal terminations of the jobs, results in extremely large mean error in user-estimated runtimes with a positive sign showing over-estimation. Since the feasibility analysis at the time of admission of the job is performed using user-estimated runtimes, extremely large overestimation in runtimes results in a high degree of unnecessary rejections of jobs. The goal of Pre-Scheduling Engine (PE) is thus to prevent low utilizations of the resources resulting from unnecessary job rejections. For this purpose, user-estimated runtimes are transformed by PE before being passed to the Scheduler. PE can use different policies for runtimes transformation as required. Also, it can adjust its policy according to θ which is being measured by SEM. The thesis presents two novel pre-scheduling policies which are discussed next.

7.4.1. Compression Policy (CMP)

Since the analysis of the traces presented in Section 4.3.3 shows that most of the jobs overestimate their runtimes, CMP reduces the estimated runtimes of all jobs by a fraction f before passing it to the scheduler. For the results presented in Section 7.5.4, the research fixes f for a given set of experiments but varies it as a parameter to study its impact on performance.

7.4.2. Multiple Feedback Loops Policy (MFP)

Since the analysis of the traces presented in Section 4.3.3 shows that different estimated runtimes of jobs have different distributions of errors, in MFP, jobs are divided into multiple classes based on their estimated runtimes. SEM continuously monitors the resource schedule and calculates mean error in estimation for each of the classes. Based on the values of the mean errors, for each class Z, a factor f_Z is maintained. The factor f_Z is calculated by dividing the mean actual runtimes of the jobs in a class with the mean estimated runtimes of the jobs in that class. The estimated runtime of every new job is transformed by multiplying it with factor f_Z (that depends on the class of a job) before passing it to the scheduler. Note that SEM continuously updates the values of f_z's based on the changes in mean errors of estimation in each class.

7.5. Performance Analysis of Resource Liaison and Controller with Homogeneous Shared Resources with Uncertain Runtimes of Jobs

Table 7.1 presents the sets of experiments, conducted with RLC with homogeneous shared resource model with uncertain runtimes of jobs. The setup for these experiments has been discussed in detail in Section 4.1.1. The workload model has been discussed in Section 4.3.2 while the models of errors in user-estimated runtimes have been discussed in Section 4.3.3. For these experiments, PAR was set equal to 1. The values of other parameters for each of the experiment sets are given in Table 7.1. GSD is used as the scheduling algorithm in these experiments.

Case SE.1, Case SE.2, Case SE.3 and Case SE.4 in Table 7.1 study the effect of comparatively small error in user-estimated runtimes and investigate the impact of SEM on performance. Case SE.5, Case SE.6 and Case SE.7 investigate the impact of

over/underestimation on performance for a given mean percentage error in estimated runtimes. In Case SE.8, Case SE.9 and Case SE.10 error in runtime estimation is modeled based on the traces collected for IBM SP2 supercomputers. These cases study the impact of extremely large error in estimation and investigate the effectiveness of the CMP and MFP pre-scheduling policies in preventing performance degradation for such scenarios.

Table 7.1: Cases Studied with RLC with Homogeneous Shared Resources with Uncertain Runtimes of Jobs

Case No.	L	Error in e_E		SEM and APB Active	PE Active	PE Policy
		B	η_{mean}			
SE.1	50% to 1000%	20%	0%	No	No	--
SE.2	50% to 1000%	20%	0%	Yes	No	--
SE.3	50% to 1000%	20%	10%	Yes	No	--
SE.4	50% to 1000%	20%	-10%	Yes	No	--
SE.5	400%	20 to 100%	10% to 50%	Yes	No	--
SE.6	400%	20 to 100%	0%	Yes	No	--
SE.7	400%	20 to 100%	-50% to -10%	Yes	No	--
SE.8	589%	Error in e_E Based on Traces		Yes	No	--
SE.9	589%	Error in e_E Based on Traces		Yes	Yes	CMP
SE.10	589%	Error in e_E Based on Traces		Yes	Yes	MFP

7.5.1. Impact of Error in User-Estimated Runtimes

This section investigates the impact of error in estimated runtimes on performance. The results correspond to Case SE.1 in Table 7.1. The results in this section are compared with Case S.1 in Table 6.1 and Case SE.2 in Table 7.1. For a fair comparison, the results for cases S.1, SE.1 and SE.2 (also SE.3 and SE.4) were obtained using exactly the same workload including the same job arrival times, earliest start times, actual runtimes and laxities. The only difference is that unlike cases SE.1 and SE.2, for Case S.1 the estimated runtimes of jobs are exactly equal to their actual runtimes. For

Case SE.1 SEM was not enabled while for Case SE.2 SEM was enabled to adapt the resource schedule. For cases SE.1 and SE.2, since the laxity of a job and error in its runtime are generated from two independent distributions, some anomalous cases, where the deadline of the job is less than its start time plus estimated runtime, are produced. All such cases were discarded from the workload before running the experiments.

Figure 7.1 and Figure 7.2 shows that W_R and U for Case SE.1 have same trends as

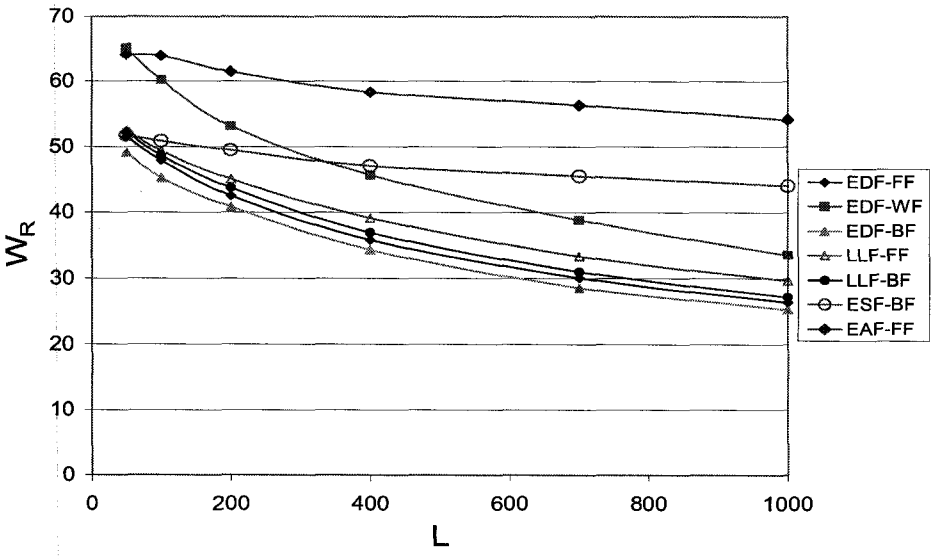


Figure 7.1: Work Rejected for Case SE.1

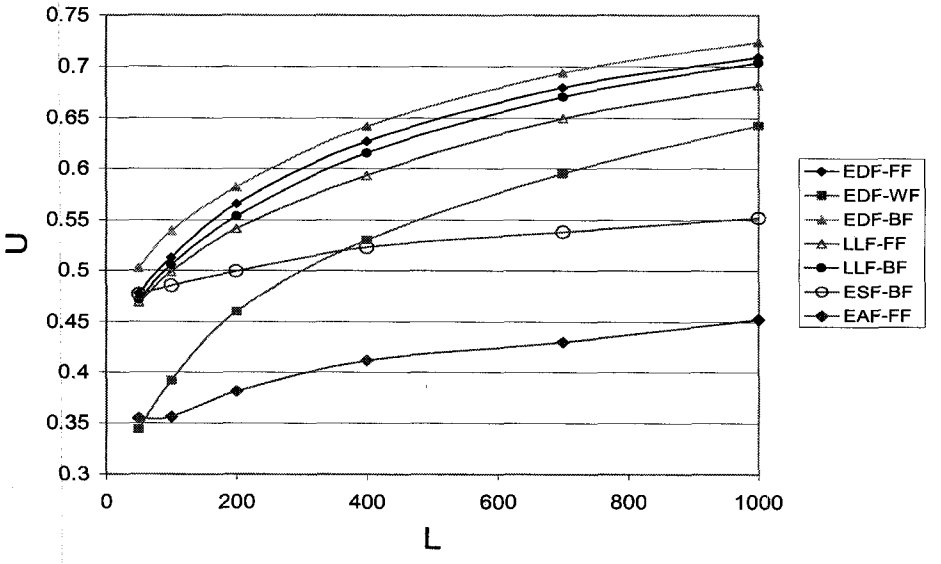


Figure 7.2: Utilization for Case SE.1

those of W_R and U obtained with complete *a priori* knowledge of job runtimes shown in Figure 6.1 and Figure 6.2. However, for any given value of L, W_R is slightly lower in Figure 7.1 than that in Figure 6.1 while U in Figure 7.2 is slightly higher than that in Figure 6.2. This is because when the jobs with overestimated runtimes finish earlier, the resource is able to accept slightly more work. Note that jobs with underestimated runtimes are not extended and are aborted after their estimated completion time and hence they do not take up the extra space left by the jobs with overestimated runtimes. This decreases W_R and hence increases U. However, since a number of jobs accepted by the resource are aborted, the useful utilization UU of the resource in Figure 7.4 is substantially lower than that in Figure 6.2. For example with EDF-BF, for any given value of L, UU in Figure 7.4 is approximately 45% lower than that in Figure 6.2. This shows that if not handled properly, even small errors in user-estimated runtimes can severely degrade system performance.

Figure 7.3 shows that with the increase in L, W_A increases. The reason is that with the increase in L the percentage of work accepted increases and so does the amount of work with underestimated runtimes. Since all jobs with underestimated runtimes are aborted, W_A increases with L. EDF-BF has the highest W_A, because it has the lowest W_R and hence the highest percentage of work accepted.

The results show that R_{AR} in Figure 7.5 is similar to R_{AR} in Figure 6.3 while Q_R in Figure 7.6 is similar to Q_R in Figure 6.4.

7.5.2. Performance of Schedule Exceptions Manager

This section presents the results for Case SE.2 in Table 7.1. These results are obtained with exactly the same workload and same values of all parameters as in Section

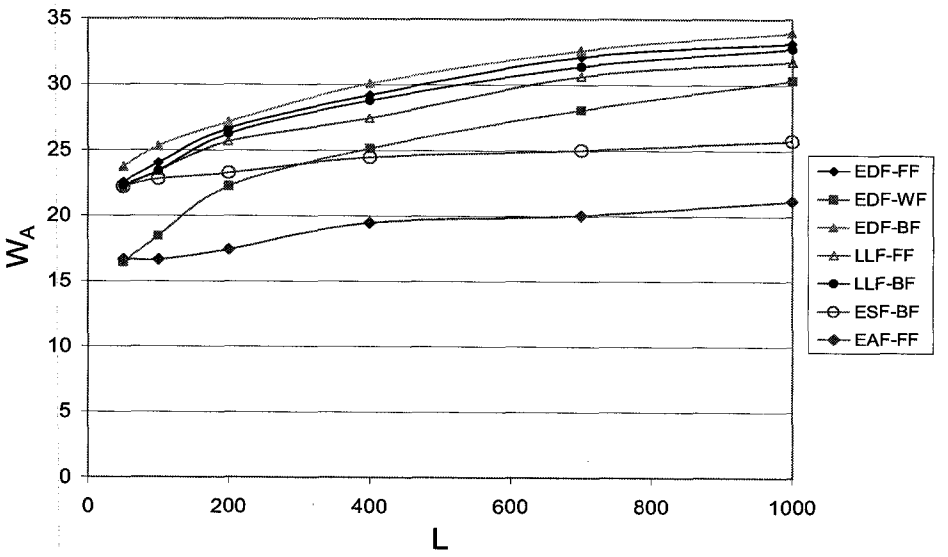


Figure 7.3: Work Aborted for Case SE.1

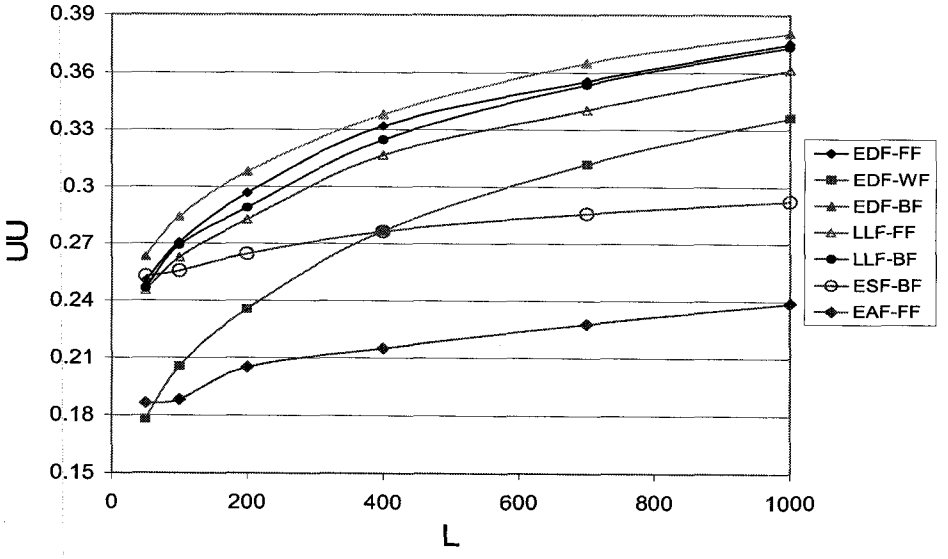


Figure 7.4: Useful Utilization for Case SE.1

7.5.1 but by activating SEM that monitors the resource schedule and adapts it to the changing conditions.

The results presented in Figure 7.7, Figure 7.8 and Figure 7.11 show that W_R, U and R_{AR} in this case are almost the same as obtained without activating SEM in the SE.1 case discussed in Section 7.5.1. However, for any given value of L, W_A in Figure 7.9 is substantially lower than that in Figure 7.3. For example, with EDF-BF the work aborted

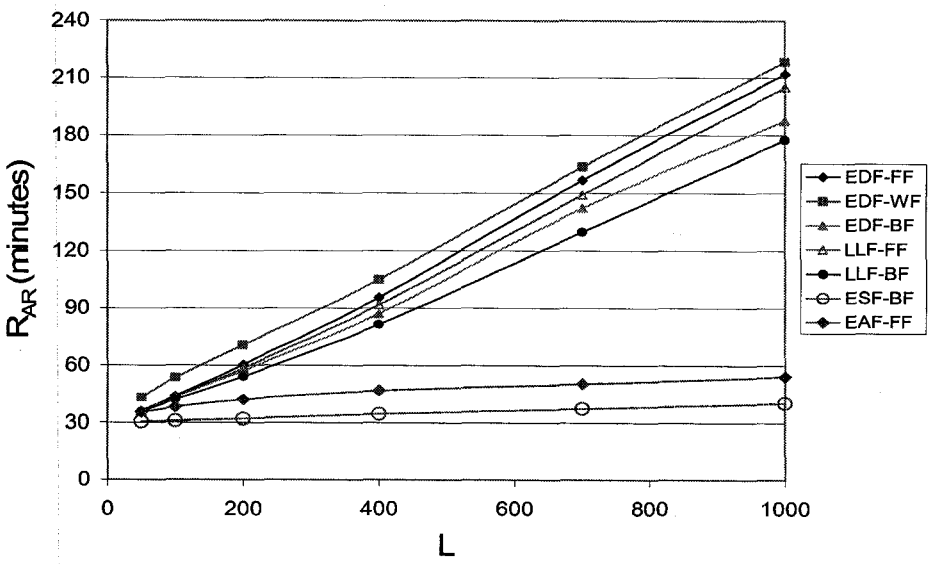


Figure 7.5: Response Time of Advance Reservation Requests for Case SE.1

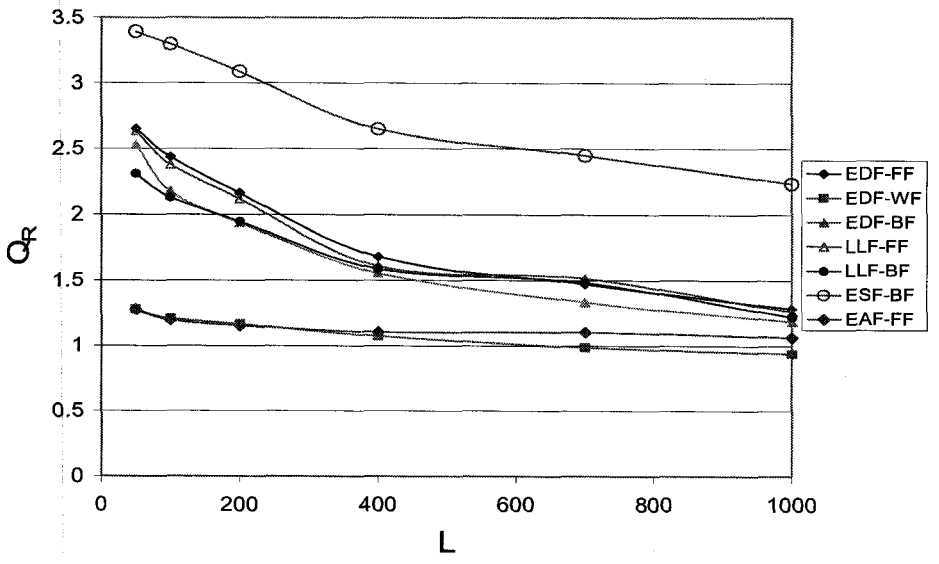


Figure 7.6: Fairness for Case SE.1

in Figure 7.9 is only 23% to 53% of that aborted in Figure 7.3. As a result, useful utilization UU of the resource in Figure 7.10 is significantly higher than that obtained in Figure 7.4. With EDF-BF for example, UU of resource with SEM is 1.44 to 1.71 times of that obtained without SEM. A comparison of Figure 7.4 and Figure 6.2 shows that SEM is effective in bringing UU of the resource in the presence of errors in user-estimated runtimes close to that obtained when the estimated runtimes of the jobs are exactly equal

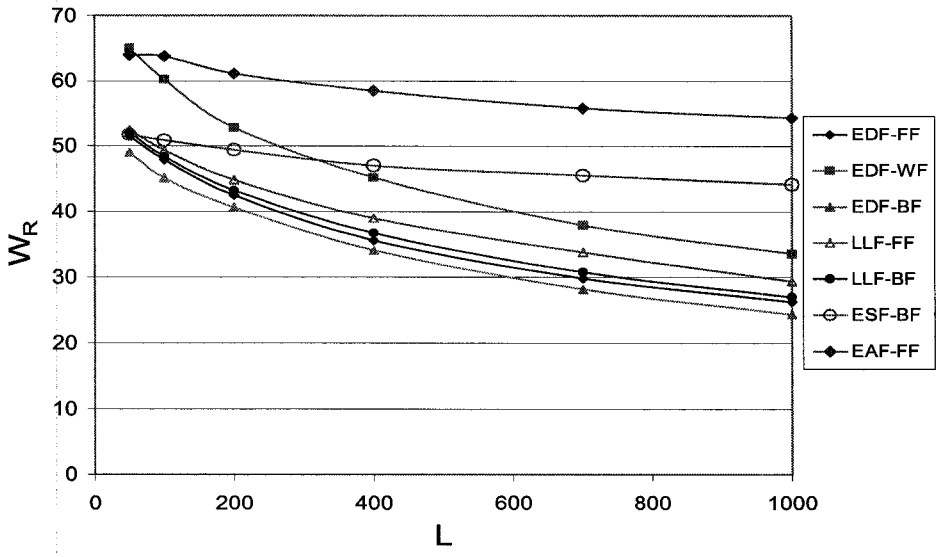


Figure 7.7: Work Rejected for Case SE.2

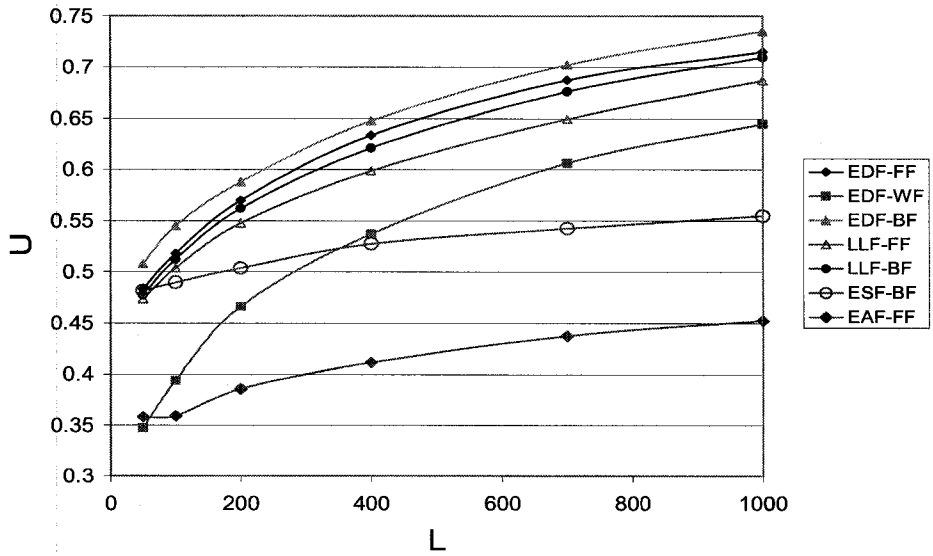


Figure 7.8: Utilization for Case SE.2

to their actual runtimes. With EDF-BF, UU in Figure 7.4 is 79.1 to 93.8 percent of that obtained in Figure 6.2.

Figure 7.9 shows that with the increase in L, W_A increases. This is because of two factors. First, with the increase in L, W_R decreases which increases the number of jobs with under-estimated runtimes in the resource schedule. Second, increasing L increases U of the resource and hence there is less capacity in the resource schedule to allow job

extensions. For a given job ordering heuristic, the BF heuristic for node selection results in the highest W_A because by fitting the jobs too close together it does not leave them space to *expand* if they need to. Hence, although we get the highest U with EDF-BF, unlike in Figure 6.2 the highest UU in Figure 7.10 is obtained with EDF-FF.

Results show that Q_R in Figure 7.12 is similar to that obtained in Figure 6.4.

Results for cases SE.3 and SE.4 are presented in Appendix C.1 and Appendix C.2

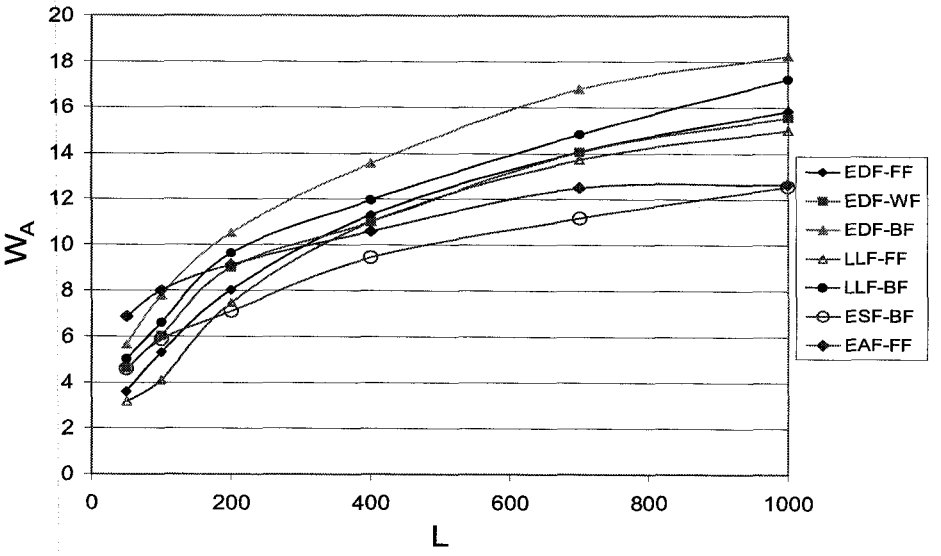


Figure 7.9: Work Aborted for Case SE.2

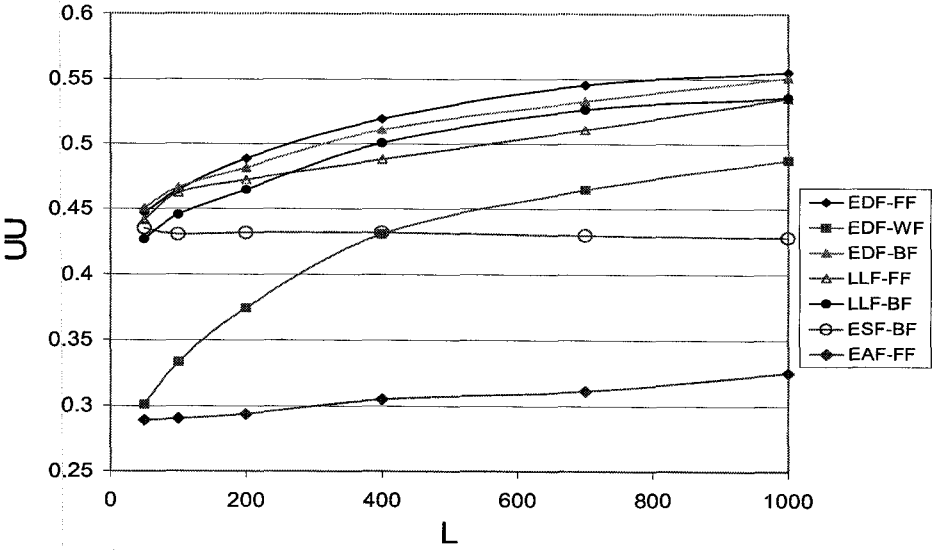


Figure 7.10: Useful Utilization for Case SE.2

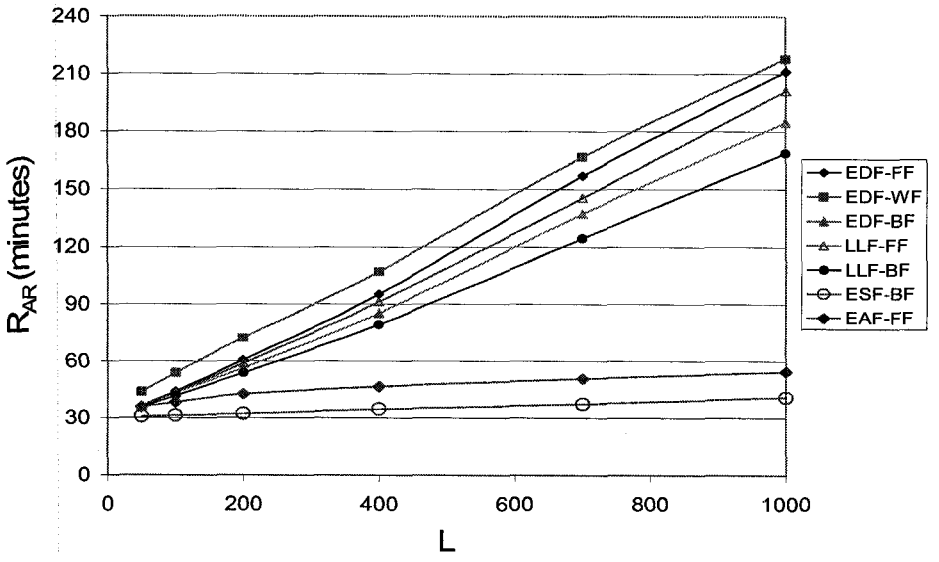


Figure 7.11: Response Time of Advance Reservation Requests for Case SE.2

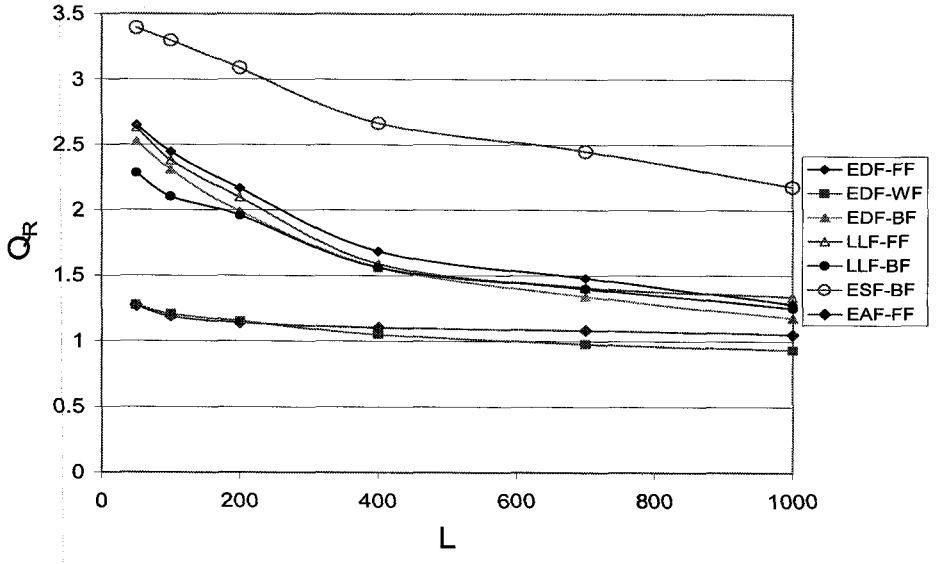


Figure 7.12: Fairness for Case SE.2

respectively. In Case SE.3 all jobs overestimate their runtimes while in Case SE.4 all jobs underestimate their runtimes. The results presented in Appendix C.1 and Appendix C.2 confirm that SEM is effective in preventing performance degradation in the presence of errors in user-estimated runtimes irrespective of whether jobs underestimate or overestimate their runtimes.

7.5.3. Impact of Under/Overestimating Runtimes of Jobs

If jobs overestimate their runtimes, W_R is expected to increase since the feasibility analysis at the time of admission is done using user-estimated runtimes. Hence, although a feasible schedule with actual runtimes of the jobs may exist, it may not exist when runtimes are greatly overestimated. On the other hand, if jobs underestimate their runtimes, some of the accepted jobs may not get the additional CPU times they need and W_A is likely to increase. This section investigates the impact of under/overestimating runtimes of the jobs on system performance in detail so as to establish guidelines for runtime-predicting-algorithms. The results in this section correspond to experiments sets SE.5, SE.6 and SE.7 in Table 7.1. For these experiment sets, L is fixed at 400% and the band of error B is varied from 20% to 100%. For each value of B, results are obtained for each of the three SE.5, SE.6 and SE.7 cases. In the SE.7 case (U-case), the runtimes of all jobs are underestimated with $\eta_{mean} = -B/2$. This is achieved by setting $\check{a} = -B/2$ in Equation 4.8. In the SE.5 case (O-case), $\check{a}$ is set equal to B/2 and thus the runtimes of all jobs are overestimated with $\eta_{mean} = B/2$. In the SE.6 case (H-case), half of the jobs overestimate their runtimes and half underestimate their runtime. For this case $\check{a} = \eta_{mean} = 0$. Since the results in Section 7.5.2 show that EDF-FF gives better performance than EDF-BF in terms of higher UU when there are underestimated jobs in the system, for each of the three cases investigated two curves are obtained for the performance metrics – one for EDF-FF and the other for EDF-BF. For the sake of comparison, the graphs presented in this section also show the corresponding values of the performance metrics obtained for Case S.1 in Section 6.3.1 when there were no errors in user-estimated runtimes (see lines with legend NE in the figures).

Figure 7.13 shows that with the increase in B, for the O-case, W_R increases substantially. This is because it is difficult to find out a feasible schedule with comparatively large (estimated) runtimes. With the increase in B, the U-case presents smaller and smaller estimated runtimes to the scheduler to find out a feasible schedule for. W_R for this case hence decreases significantly with B. For the hybrid H-case, which consists of mix of overestimated and underestimated jobs, W_R more or less remains

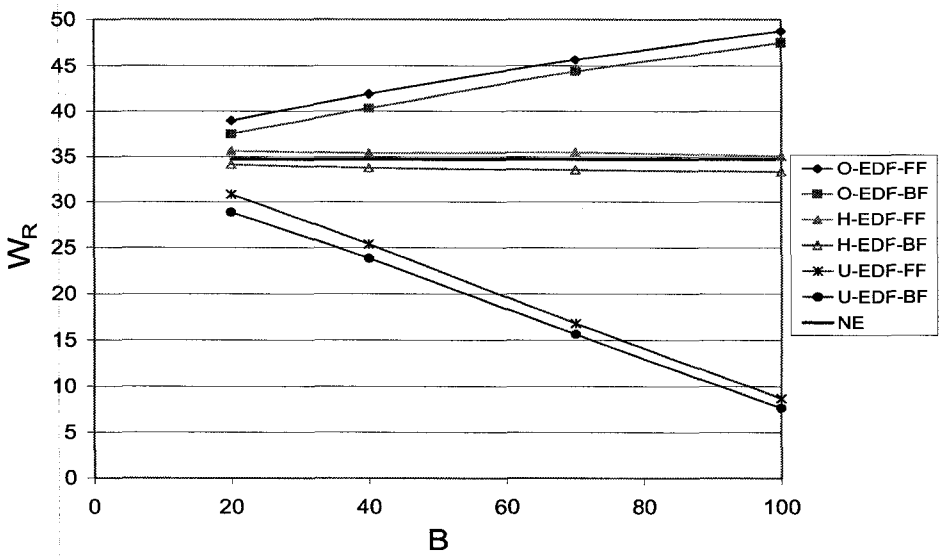


Figure 7.13: Impact of Over/Underestimation of Runtimes of Jobs on Work Rejected

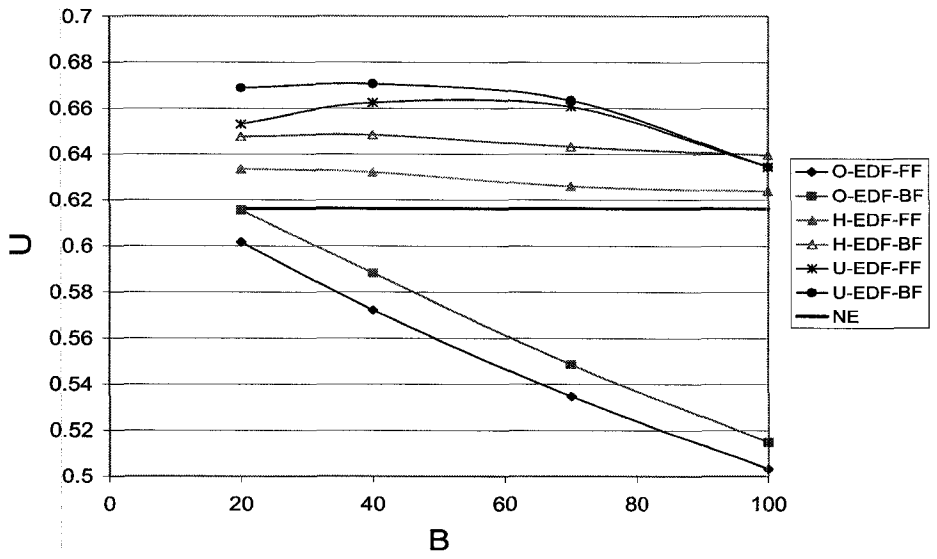


Figure 7.14: Impact of Over/Underestimation of Runtimes of Jobs on Utilization

invariant of the different values of B.

As W_R of the O-case increases with B, its U decreases in Figure 7.14. Since all jobs for the O-case overestimate their runtimes, none of the jobs is aborted. This results in $W_A = 0$ for the O-case for all values of B as shown in Figure 7.15. For the U-case, U first increases slightly with the increases in B and then decreases. The reason is that with the increase in B, W_R for the U-case decreases which tends to increase U. However, as

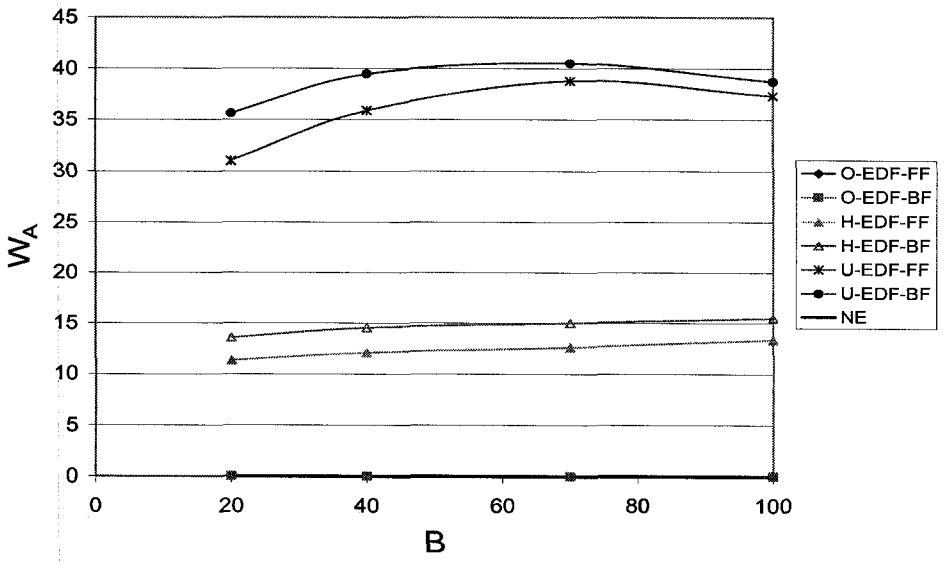


Figure 7.15: Impact of Over/Underestimation of Runtimes of Jobs on Work Aborted

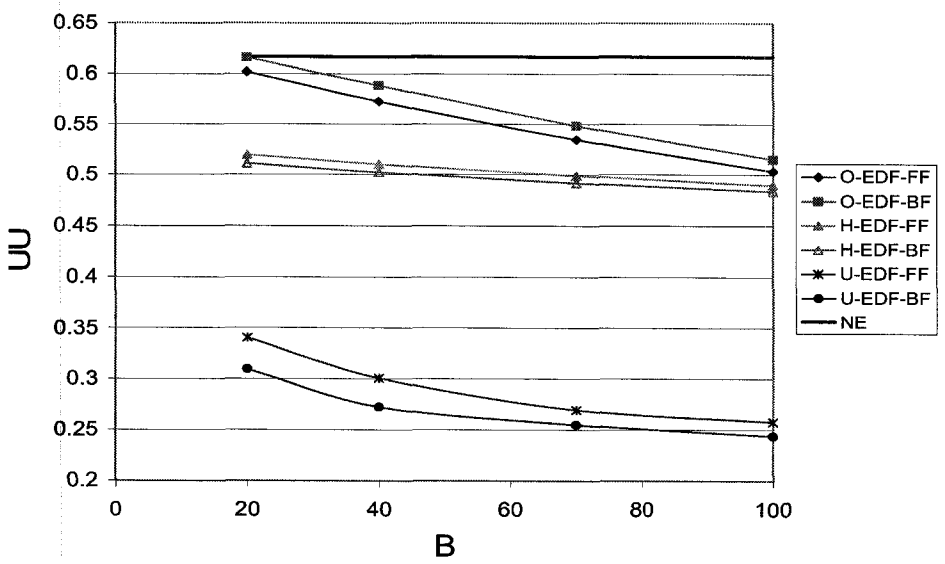


Figure 7.16: Impact of Over/Underestimation of Runtimes of Jobs on Useful Utilization

the degree of underestimation of runtimes of the jobs increases, not only the probability of job abortion increases (as shown in Figure 7.17) but also jobs are aborted much earlier. Early abortion of jobs means that jobs are aborted after they have been extended for only a few quanta τ. Early job abortions increase with an increase in the degree of underestimation as it becomes increasingly difficult for the scheduler to find out a feasible schedule for the jobs that require large extension times. This early abortion of jobs results in lower U. Figure 7.15 shows that a large percentage of work is aborted for the U-case. W_A first increases with an increase in B for the U-case but as the degree of underestimation increases further, W_A decreases because of early job abortions. U and W_A do not change significantly with the increase in B for the H-case.

Figure 7.16 shows that for a given value of B, the O-case always results in the highest useful utilization. For small values of B, UU with EDF-BF in the O-case is close to that obtained with EDF-BF for the S.1 case in Figure 6.2. However, with higher values of B, W_R increases significantly which decreases UU for the O-case. The U-case results in the lowest UU due to the large number of jobs aborted. As P_A for the U-case increases with the increase in B, UU decreases. Figure 7.16 also shows that for the U-case, EDF-FF performs better than EDF-BF by allowing more space for the jobs to *expand.* UU obtained with the H-case is some where in between the O-case and the U-case and it is least affected with the increase in B.

These results thus show that choice of over/underestimation can significantly affect overall system performance. For the parameters used in the experiments, the difference in UU obtained with EDF-FF for the U-case and that obtained with EDF-BF for O-case may be as large as 0.288 which amounts to 46.75% of U obtained with EDF-

BF in Figure 6.2. These results strongly suggest that with the error handling policies used in the thesis and for a wide range of η, it is always better to overestimate runtimes of the job than underestimate them. As no jobs are aborted in the O-case, it may be attractive from the user's perspective as well. Figure 7.16 also shows that even if the mean percentage error for the overestimation case is 5 times that in the underestimation case,

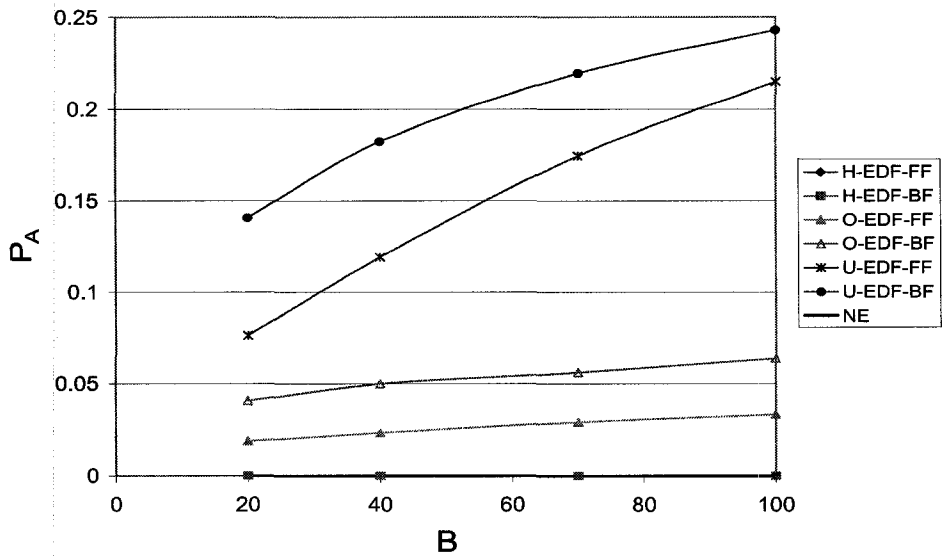


Figure 7.17: Impact of Over/Underestimation of Runtimes of Jobs on Probability of Aborting

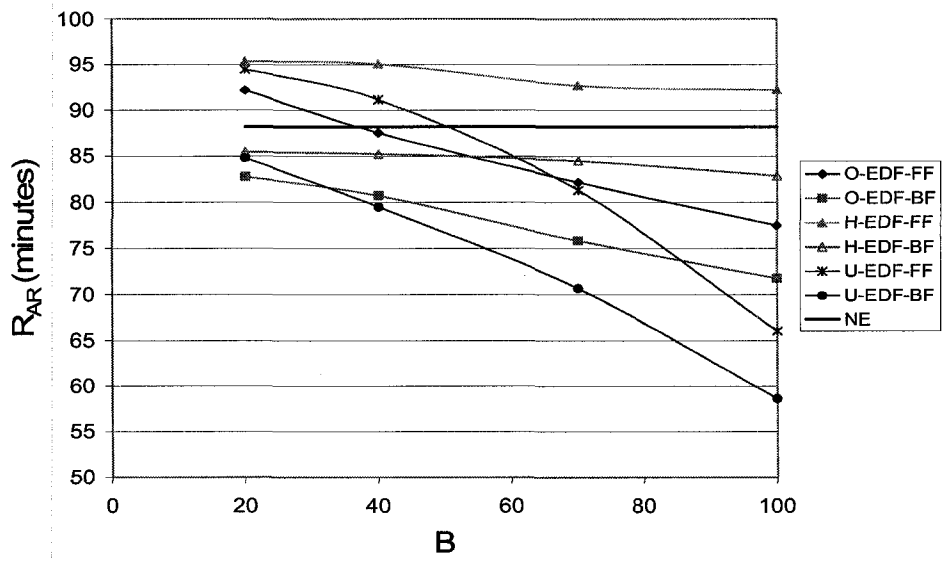


Figure 7.18: Impact of Over/Underestimation of Runtimes of Jobs on Response Time of Advance Reservations

UU obtained with overestimation is still 1.51 times that obtained with underestimation. The thesis thus strongly suggests that the runtime-predicting-algorithms should always try to avoid underestimation even if that leads to a higher percentage error.

Figure 7.18 shows that with the increase in B, R_{AR} decreases. For the O-case and H-case, it is because of the decrease in U with B. As with the increases in B in U-case, estimated runtimes of the jobs decrease and so do their estimated completion times. Hence, as the resource is estimated to become idle earlier, jobs are scheduled at an earlier time. This decreases the mean wait times of the jobs and hence R_{AR} decreases. Note that the jobs that cannot be extended are aborted and hence as P_A increases with B, mean wait times of the jobs do not increase.

Q_R curves (not shown to save space) shows that there is no significant effect of B on Q_R.

7.5.4. Performance of Pre-Scheduling Engine

Results in Section 7.5.2 and Section 7.5.3 show that SEM is effective in countering degradation in performance in the presence of uncertain runtimes of jobs. If jobs overestimate their runtimes then results in Section 7.5.3 show that, with SEM, for even B = 100% and η_{mean} = 50%, UU of the resource is 83.6% of that obtained with the exact *a priori* knowledge of the runtimes of jobs. However, the analysis of SP2 traces, presented in Section 4.3.3, show that most of the times error in estimation is orders of magnitudes higher than 100% – sometimes as much as 25000%. As schedulability analysis at the time of admission is done using user-estimated runtimes, even SEM may not be able to prevent many unnecessary job rejections for such highly overestimated jobs. Under these conditions, a pre-scheduling engine PE in the framework is needed that

can transform job runtimes before passing them to the scheduler for schedulability analysis. Section 7.4.1 and Section 7.4.2 propose two novel pre-scheduling policies, Compression Policy (CMP) and Multiple Feedback Loops Policy (MFP), for runtime transformations. For CMP, compression factor f is fixed for a given set of results but is varied as a parameter to study its impact on performance. For the results presented in this section, four job classes are used for MFP. The first class corresponds to jobs with runtimes less than 10000 seconds. Preliminary experimentation revealed that transforming runtimes of small jobs results in a substantial number of job abortions without a significant decrease in W_R. Thus, runtimes of jobs in the first class are not compressed before passing them to SCH. For the other three classes, f_Z is dynamically chosen in accordance with the latest value of the mean error in estimation for each of the classes. These classes are formed such that approximately an equal number of jobs belong to each class. For the experiments, all jobs with runtimes between 10000 and 100000 seconds are in the second class, those with runtimes between 100000 and 200000 seconds in the third class and those with runtimes above 200000 seconds in the fourth class. Experimental results have shown that increasing the number of classes increases the overheads imposed on the system without a significant improvement in useful utilization of the resource.

This section presents results obtained for the cases SE.8, SE.9 and SE.10 in Table 7.1, where error in estimation is modeled based on the analysis of traces. The results are obtained with L = 589% (50% with respect to estimated runtimes) and using the EDF-BF heuristic with GSD. The graphs in this section present the values of the performance metric obtained: a) by running SEM only (Case SE.8), b) when there is no error in user-

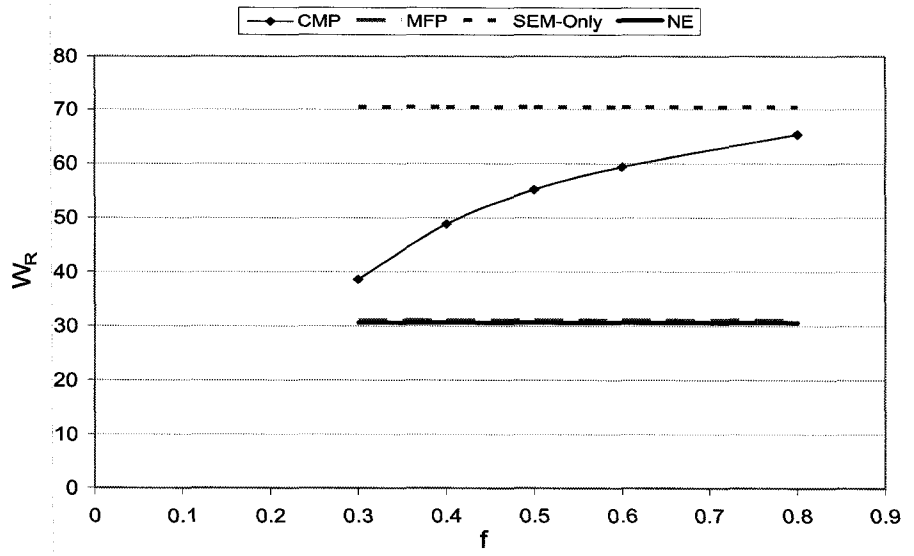


Figure 7.19: Impact of Pre-Scheduling Engine on Work Rejected

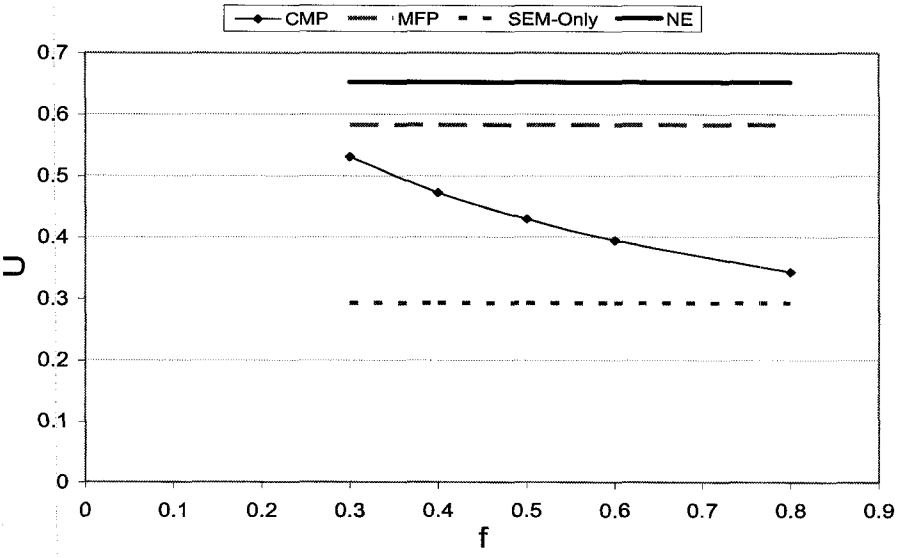


Figure 7.20: Impact of Pre-Scheduling Engine on Utilization

estimated runtimes (Case S.1 in Table 6.1), c) when CMP is used as a pre-scheduling policy (Case SE.9) and d) when MFP is used as a pre-scheduling policy (Case SE.10).

Figure 7.19 shows that without PE, W_R for the case when there are errors in user-estimated runtimes can be as high as 40% over that obtained when there are no errors in user-estimated runtimes. As PE is enabled, W_R decreases significantly. For CMP, smaller

values of f present to the scheduler a lower amount of work to schedule and thus result in a lower W_R.

Figure 7.20 shows that U of the resource improves with CMP. U increases substantially with the decrease in f. However, the decrease in f for CMP also results in a substantial increase in W_A in Figure 7.21 and P_A in Figure 7.23. This is because as f decreases, the proportion of jobs which have higher actual runtimes than that presented to

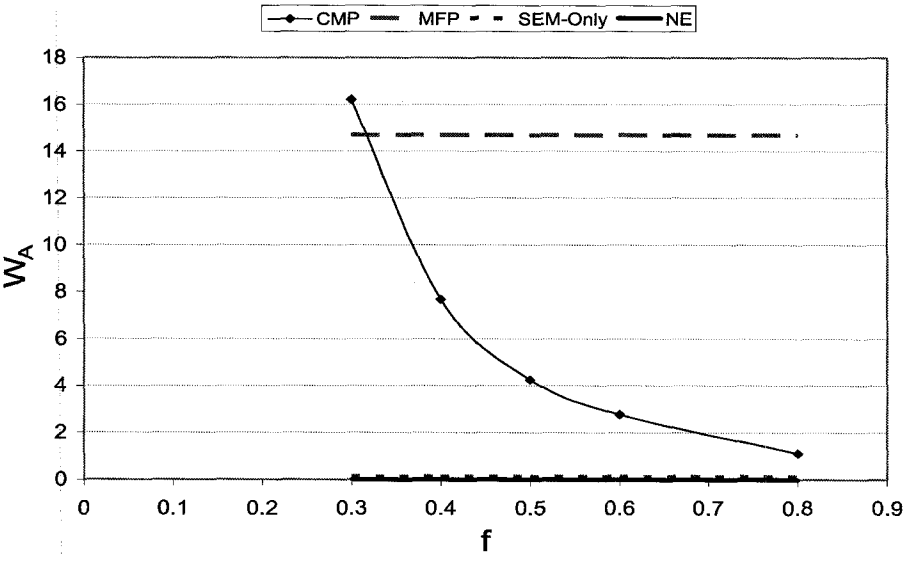


Figure 7.21: Impact of Pre-Scheduling Engine on Work Aborted

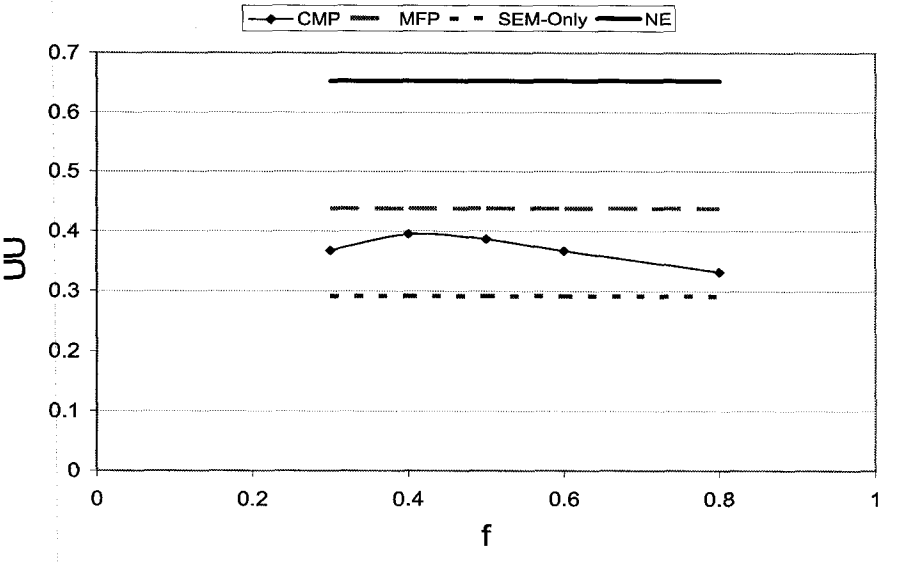


Figure 7.22: Impact of Pre-Scheduling Engine on Useful Utilization

the scheduler increases and it becomes increasingly difficult for the scheduler to find out a feasible schedule when a number of those *underestimated* jobs require extra time to complete.

Figure 7.22 shows that for CMP, the UU of the resource increases upto a certain point (f = 0.4) with a decrease in f due to the decrease in W_R. For further decrease in f the effect of P_A becomes more dominant and UU starts decreasing. This shows that for a

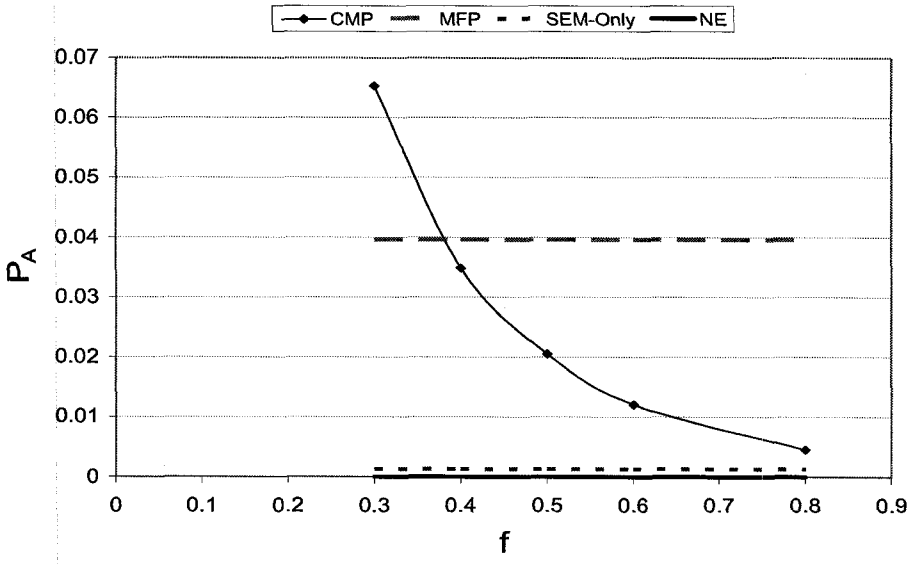


Figure 7.23: Impact of Pre-Scheduling Engine on Probability of Aborting

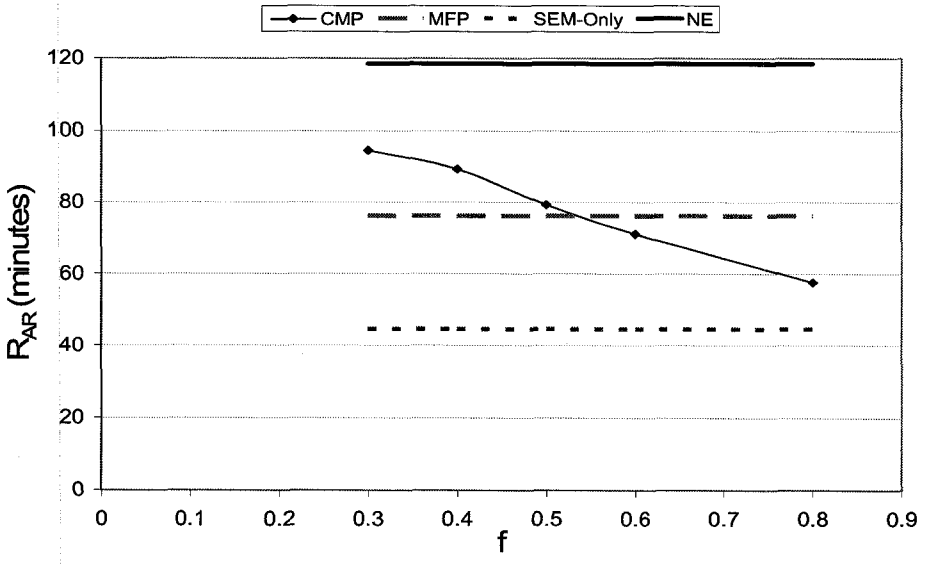


Figure 7.24: Impact of Pre-Scheduling Engine on Response Time of Advance Reservations

given percentage error in user-estimated runtimes, there exists a fraction f where the decrease in W_R and the increase in P_A are such that we get the highest UU possible with CMP. At f = 0.4, UU with CMP is 1.35 times that obtained with SEM-only.

Results in Figure 7.22 show that for MFP, we get a higher UU than that obtained with CMP. This shows that dividing jobs into classes and dynamically adjusting f_Z for each class can result in a better runtime transformation, where the transformed value of the runtime is closer to the actual runtime. MFP results in 1.5 times the UU obtained with SEM-only. These results thus show that pre-scheduling engine can substantially improve performance of scenarios where there are extremely high percentage errors in user-estimated runtimes.

Figure 7.24 shows that with CMP there is an increase in R_{AR} with a decrease in f. This can be attributed to an increase in U with a decrease in f. A higher utilization gives rise to a higher resource contention leading to an increase in R_{AR}.

7.6. Summary

Providing QoS guarantees under uncertain runtimes of the jobs is a challenging task. This chapter presented Schedule Exceptions Manager to manage uncertain user-estimated runtimes and abnormal termination of jobs. SEM seeks to adapt the resource schedule dynamically depending upon the conditions. This chapter also presented Pre-Scheduling Engine which is effective in the presence of extremely large errors in estimation. The performance analysis of the uncertainty handling components of the framework helps in gaining important insights.

The results show that error in user-estimated runtimes can significantly affect system performance. Even with a B of 20%, for the parameters used in the experiments,

UU of the system can be as low as 55% of that obtained with an accurate *a priori* knowledge of the runtimes of the jobs. The results show that SEM is highly effective in handling user-estimated runtimes. For example, in Figure 7.10, UU obtained with SEM was 1.44 to 1.71 times that obtained without it and 79.1 to 93.8% of that obtained with the exact *a priori* knowledge of the runtimes of the jobs.

Results in Section 7.5.3 show that for a given percentage error in user-estimated runtimes, overestimation and underestimation can have a significantly different impact on system performance. Given a certain η, for the parameters used in the experiments, UU obtained with overestimation can be as high as twice of that obtained with underestimation. Figure 7.14 also shows that even if the mean percentage error for the overestimation case is 5 times that in the underestimation case, UU obtained with overestimation is still 1.51 times that obtained with underestimation. Since overestimating job runtimes prevents premature job abortions as well, the thesis strongly suggests that the runtime-predicting-algorithms should always try to avoid underestimation even if that leads to a higher percentage error.

The thesis presents two novel pre-scheduling policies that have been observed to improve system performance under extremely large errors in estimation of job runtimes. For example, in Figure 7.22, MFP resulted in a 50% increase in UU obtained with SEM only.

Chapter 8

Matchmaking in Multi-Institutional Grids

8.1. Matchmaking Problem in Multi-Institutional Grids

As discussed in Section 3.4.2, effective application-to-resource mapping and load balancing are important to obtain the best possible performance. This process is generally referred to as *matchmaking*. The matchmaker should select suitable resource from the candidate set in such a way that not only the applications' constraints are met but also overall system performance is maximized. As the results in Section 8.5 will show, traditional matchmaking algorithms used in distributed computing result in poor performance when applied in multi-institutional settings, particularly for workloads consisting of both advance reservations and best-effort jobs. This thesis investigates several matchmaking algorithms for multi-institutional Grids and presents a novel algorithm for matchmaking, Minimum Laxity Impact (MLI) that outperforms all other algorithms investigated in almost every respect for a wide range of workload parameters. Another application of MLI is as an effective meta-scheduler within Platform's Community Scheduler Framework (CSF) [PLA05c] that is used to dispatch jobs to resource managers in multiple domains. The thesis further extends MLI for scenarios involving heterogeneous different speed resources.

8.2. Algorithms for Matchmaking in Multi-Institutional Grids

8.2.1. Random (RAN)

This algorithm randomly selects a resource from the candidate set. If the resource meets the QoS requirements of the job, the resource is selected. Otherwise, the process is repeated until either a match is found or all options are exhausted.

8.2.2. First Fit (FF)

With this algorithm, all resources in the candidate set are sequenced in a particular order. Each time a new job is submitted, the first resource in sequence is evaluated. If the resource meets the QoS constraints of the job, the resource is selected. Otherwise, the next resource in sequence is evaluated. 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.

8.2.3. Next Fit (NF)

In this algorithm, all resources in the candidate set 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 resource does not meet QoS constraints of the job the next resource in sequence is evaluated. 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 [SUN05]. This algorithm has the same limitations as FF.

8.2.4. Initial Minimum Completion Time (IMCT)

Whenever a new request arrives, this algorithm queries all resources in the candidate set 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.

8.2.5. Resource with Minimum Utilization (MinU)

Whenever a new request arrives, this algorithm queries all resources in the candidate set and selects the one with minimum utilization among the resources that meet the QoS constraints of the job. This algorithm thus seeks to balance the load on resources. It is used by Sun Grid engine [SUN05].

8.2.6. Resource with Maximum Utilization (MaxU)

Whenever a new request arrives, this algorithm queries all the resources in the candidate set 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 admit jobs with large demands.

8.3. Minimum Laxity Impact Algorithm

In this algorithm, whenever a new request arrives, all resources in the candidate set 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, the impact on laxity is considered to be high. On the other hand, if the resource can accommodate the job without significant re-scheduling, the impact on laxity is said to be low. Thus, if the resource that has minimum impact on laxity to accommodate the new job is always selected, there

would be a higher probability of finding out a feasible schedule for later arrivals by utilizing the spare laxity. Quantitatively, the impact on laxity is measured by subtracting from cumulative *remaining laxity* of the jobs in schedule before the new job was admitted the cumulative remaining laxity of the jobs after the new job is accommodated.

The laxity LX_i of a job j_i on a resource b can be given as:

$$LX_i = d_i - e_{E,ib} - t_i \quad \textbf{(8.1)}$$

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 j_i measures the "laxity" of the job at its scheduled-time sch_i at which it is currently scheduled to start. Thus,

$$RL_i = d_i - e_{E,ib} - sch_i \quad \textbf{(8.2)}$$

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

$$\zeta = \sum_{(For\ all\ jobs\ ji\ in\ \epsilon)} [d_i - e_{E,ib} - sch_{i(j-1)}] - \sum_{(For\ all\ jobs\ ji\ in\ \epsilon\ and\ the\ new\ job)} [d_i - e_{E,ib} - sch_{i(j)}] \quad \textbf{(8.3)}$$

Thus, from Equation 8.2,

$$\zeta = \sum_{(For\ all\ jobs\ ji\ in\ \epsilon)} RL_{i(j-1)} - \sum_{(For\ all\ jobs\ ji\ in\ \epsilon\ and\ the\ new\ job)} RL_{i(j)} \quad \textbf{(8.4)}$$

If all resources are identical, the algorithm selects the resource that reports the smallest value of ζ. If more than one resource has same value of ζ, ties are broken by selecting the one that reports the minimum completion time. For heterogeneous resources with different speeds, ζ should be scaled with respect to the relative speeds of the resources. The extension of MLI for heterogeneous resources is the focus of the next section.

8.4. Extending Minimum Laxity Impact Algorithm for Heterogeneous Environments

Heterogeneous environments with different resources of the same type but different relative speeds are prevalent in Grid environments. As discussed in Section 4.4.7, this thesis measures heterogeneity H of the Grid environment by taking into account the relative execution times of a given job on different resources. The relative speed of the resources in this context always refers to the relative execution times of a given job on different resources. The thesis makes no assumption about the relationship between for instance, the speed of the processor and the execution time of the job.

This section extends the MLI algorithm for heterogeneous environments. As discussed in Section 8.3, for different speed resources, the impact on laxity ζ should be scaled according to the speed of the resource before making the matchmaking decision. The following factors affect the scaling of ζ.

A. Since in a heterogeneous environment different resources have different speeds, same ζ value for different resources might mean significantly different impact on performance. For example, let us consider two resources b_1 and b_2 with speeds $\acute{\upsilon}_1$ and $\acute{\upsilon}_2$ respectively, where $\acute{\upsilon}_2$ is much greater than $\acute{\upsilon}_1$. If during matchmaking with MLI, the value of ζ (assume ζ is greater than 0) happens to be the same for both b_1 and b_2, selection of b_2 for job allocation may result losing more resource power as $\acute{\upsilon}_2$ is much greater than $\acute{\upsilon}_1$. This seems to suggest that ζ should be properly scaled with respect to the relative speed of the resource. ζ values for higher speed resources should be amplified before comparing ζs of different resources. For the scenario just described, this can be done, for instance, by multiplying ζ for b_2 with $\acute{\upsilon}_2/\acute{\upsilon}_1$.

Following the same argument one can show that for the scenarios where ζs for both b_1 and b_2 are negative, the mentioned scaling scheme would encourage jobs to be run on b_2. This will help in fully utilizing the higher speed resource. For a scenario, where ζ for b_2 is positive and that for b_1 is negative, b_1 should be selected for job allocation. This is because a positive ζ for b_2 means that by admitting the new job the net remaining laxity of the jobs is decreasing while a negative ζ for b_1 means that by admitting the new job the net remaining laxity of the jobs is increasing. Similarly, for a scenario where b_2 is negative and b_1 is positive, b_2 should be selected for job allocation. It can be seen that for both of these scenarios, the above mentioned scaling scheme results in the correct selection of the resource. This scaling scheme has thus been used in an extension of MLI called Minimum Laxity Impact with Scaling A (MLI-SA).

B. Alternatively, since the execution time of a job is inversely proportional to the speed of the resource, given the start time and deadline of the job, the job laxity is directly proportional to the resource speed. This means that jobs allocated to b_2 would have more laxities than the jobs with the same execution times would have on b_1. As a result, schedule on b_2 has more flexibility than the schedule on b_1. Given the same ζ (assume ζ is greater than 0) for both b_1 and b_2, this seems to suggest that b_2 should be selected for job allocation as b_1 has less flexibility. As execution time of a job on b_2 is equal to $\acute{\upsilon}_1/\acute{\upsilon}_2$ of that on b_1 and since laxities of jobs are directly proportional to their runtimes, this seems to suggest that ζ for b_2 should be scaled by multiplying it with $\acute{\upsilon}_1/\acute{\upsilon}_2$.

Following the same argument one can show that for the scenarios where ζs for both b_1 and b_2 are negative, the mentioned scaling scheme would encourage jobs to be run on b_1. This will help in getting more flexibility in schedule of resource b_1. For a scenario, where ζ for b_2 is positive and that for b_1 is negative, b_1 should be selected for job allocation and for a scenario where b_2 is negative and b_1 is positive, b_2 should be selected for job allocation. It can be shown that for both of these scenarios, the above mentioned scaling scheme results in the correct selection of the resource. This scaling scheme has thus been used in an extension of MLI called Minimum Laxity Impact with Scaling B (MLI-SB).

The two factors A and B are exactly opposite to each other. As will be shown in Section 8.5 that during the experiments if the scaling strategy is based on factor A, the effect of factor B increases and cancels the effect of scaling. Similarly, if the scaling strategy is based on factor B, the effect of factor A increases and cancels the effect of scaling. Thus, one gets approximately the same performance with MLI, MLI-SA and MLI-SB. Hence contrary to intuition, no scaling is required in MLI. However, given the unique needs of heterogeneous environments a boost in performance can be achieved by accommodating a self partitioning mechanism in MLI. This has been achieved in an algorithm that is called Minimum Laxity Impact with Self Dynamic Partitioning (MLI-SDP). MLI-SA, MLI-SB and MLI-SDP are presented next.

8.4.1. Minimum Laxity Impact with Scaling A (MLI-SA)

In this algorithm, whenever a new request arrives, all resources in the candidate set are queried and the algorithm selects one from those that meet the QoS constraints of the job. This selection is done by scaling ζ for those resources that can meet the QoS

constraints of the job. ζ is scaled for each such resource by multiplying it with the relative speed of the resource with respect to the standard resource (defined in Section 4.4.7). In other words, ζ for a given resource is multiplied with the ratio of execution time of the job on the standard resource and the execution time of the job on the given resource. The algorithm then selects that resource for job allocation which results in the smallest value of the scaled ζ.

Note that for homogeneous environments, MLI-SA automatically reduces to MLI.

8.4.2. Minimum Laxity Impact with Scaling B (MLI-SB)

In this algorithm, whenever a new request arrives, all resources in the candidate set are queried and the algorithm selects one from those that meet the QoS constraints of the job. This selection is done by scaling ζ for those resources that can meet the QoS constraints of the job. ζ is scaled for each such resource by dividing it with the relative speed of the resource with respect to the standard resource (defined in Section 4.4.7). In other words, ζ for a given resource is multiplied with the ratio of execution time of the job on the given resource and the execution time of the job on the standard resource. The algorithm then selects that resource for job allocation which results in the smallest value of the scaled ζ.

Note that for homogeneous environments, MLI-SB automatically reduces to MLI.

8.4.3. Minimum Laxity Impact with Self Dynamic Partitioning (MLI-SDP)

In a heterogeneous environment, partitioning resources into different sets and allocating different sets to different categories of jobs is a well-known paradigm that has been shown to perform well (see for example [KAP07]). The jobs can be categorized, for instance, with respect to their execution times with small jobs allocated to a particular

resource set and large jobs allocated to another set. Partitioning can be either static where a particular type of job is *always* allocated to a particular resource set or it can be dynamic where job allocation may take into account other dynamic factors, such as the current state of the system, before a particular job is mapped to a particular resource set. MLI-SDP accommodates a dynamic partitioning mechanism and makes matchmaking decisions according to the given state of the system to better utilize the resources. As the results in Section 8.5 will show this dynamic partitioning helps MLI-SDP achieve better performance than all other algorithms investigated.

Resource partitioning is usually done manually where the administrator calculates various partitioning related parameters and configures the system according to the parameter values. Such a scheme is less flexible and every time the workload or the resource configuration changes, the whole process of parameter calculations and system configuration need to be repeated to optimize system performance. On the other hand, MLI-SDP accommodates a self partitioning mechanism where no parameter calculations and system configuration is required as the algorithm automatically adjusts itself to the changing workload and resource configurations.

MLI-SDP assigns different ranks to different resources such that the lowest speed resource in the candidate set is assigned the lowest rank and the highest speed resource in the candidate set is assigned the highest rank. Resources with same relative speeds are assigned the same rank while resources with different speeds are always assigned a different rank. Whenever a new job arrives, all resources in the candidate set are queried and the algorithm selects one from those that meet the QoS constraints of the job. All such resources report to matchmaker MM a tuple consisting of an estimated completion

time of the job on the resource and impact on laxity ζ. MLI-SDP first finds the resource that reports the minimum completion time of the job. Such a resource is called the *threshold resource*. MLI-SDP then obtains a subset of resources from the candidate set that can meet the QoS constraints of the job and have the same or lower rank as that of the threshold resource. Such a subset of resources is called *active candidate set*. The algorithm selects that resource from the active candidate set which reports the smallest value of ζ and allocates it to the job. Note that for homogeneous environments, MLI-SDP automatically reduces to MLI.

In order to understand how MLI-SDP results in self dynamic partitioning, let us consider a hypothetical scenario with two resources b_1 and b_2 such that the speed of b_2 is 10 times that of b_1. Let b_1 be the standard resource. Let us consider the arrival of a job j_1 with an execution time of 100 minutes. The estimated completion time of the job j_1 depends on two factors – the wait time of the job on the resource and execution time of the job on the resource. Since speed of b_2 is 10 times that of b_1, the execution time of j_1 on b_2 is 10 minutes. Thus, for j_1 to finish on b_1 earlier than on b_2, the wait time of j_1 on b_2 should be at least 90 minutes more than that on b_1. On the other hand, if we consider another job j_2 with an execution time of 10 minutes, one can show that for j_2 to finish on b_1 earlier than on b_2, the waiting time of j_2 on b_2 should be at least 9 minutes more than that on b_1. Hence, the probability that job j_2 would finish on b_1 earlier than on b_2 is many times more than the probability that job j_1 would finish on b_1 earlier than on b_2. Since MLI-SDP includes only those resources in the candidate set that has the same or lower rank than that of the resource that reports the minimum completion time, probability of b_2 being included in the active candidate set for job j_2 is many times lower than the

probability of b_2 being included in the active candidate set for job j_1. Hence, on average there is a higher probability of b_1 being selected for smaller jobs such as j_2 than for large jobs such as j_1. The experimental investigation confirmed that during simulation, with MLI-SDP, most of the small jobs are allocated to lower speed resources and large jobs are allocated to higher speed resources. Note that minimum completion time is not the only criterion for resource selection; MLI-SDP selects that resource from the active candidate set that results in the smallest value of ζ. The dynamic state of the system is hence included in the matchmaking decision and this prevents the limitations of static partitioning to affect the performance of MLI-SDP.

8.5. Performance Analysis of Matchmaker

Table 8.1 summarizes the sets of experiments conducted with MMC for studying the performance of the Matchmaker. The setup for these experiments has been discussed in detail in Section 4.1.2 and Section 4.4.7, while the workload model has been discussed in Section 4.3.2 and Section 4.3.3. The results in Section 6.3.1 show that the EDF-BF combination of heuristics for the GSD algorithm results in the lowest W_R and the highest U. Thus, for the experiments in this chapter, the EDF-BF combination of heuristics was used with GSD for scheduling jobs on individual clusters. For the experiments where H is greater than 1, the standard resource is shown in bold face in Table 8.1 under the Relative Speed of Resources column.

Case MM.1 in Table 8.1 investigates the performance of matchmaking algorithms for a workload consisting only of advance reservation requests while Case MM.2 investigates the performance of the matchmaking algorithms for a workload consisting of both ARs and OD jobs. Case MM.3 studies the impact of the size of the candidate set.

Case MM.4 and Case MM.5 investigate the impact of arrival rate on performance while Case MM.6 and Case MM.7 investigate the impact of errors in user-estimated runtimes on the performance of the matchmaking algorithms. Case MM.8, Case MM.9, Case MM.10 and Case MM.11involve heterogeneous systems with resources of different speeds. These cases study the performance of the matchmaking algorithms in heterogeneous environments.

Table 8.1: Cases Studied with MMC for Studying the Performance of the Matchmaker

Case No.	PAR	L	Starvation Prevention		Error in e_E (η_{mean} = 0% for all cases) B	SEM and APB Active	N	Maximum Possible Utilization with a Given Arrival Rate	Heterogeneity of the System	
			SPM Active	Deadline of OD Requests					H	Relative Speeds of Resources
MM.1	1	50% to 1000%	--	--	0%	--	4	100%	1	--
MM.2	0.5	50% to 1000%	Yes	2 days	0%	--	4	100%	1	--
MM.3	1	400%	--	--	0%	--	2 to 6	100%	1	--
MM.4	1	50% to 1000%	--	--	0%	--	4	50% to 125%	1	--
MM.5	0.5	50% to 1000%	Yes	2 days	0%	--	4	50% to 125%	1	--
MM.6	1	50% to 1000%	--	--	20%	No	4	100%	1	--
MM.7	1	50% to 1000%	--	--	20%	Yes	4	100%	1	--
MM.8	0.5	50% to 1000%	Yes	2 days	0%	--	4	100%	8	1,2,4,8
MM.9	1	50% to 1000%	--	--	0%	--	4	100%	8	1,2,4,8
MM.10	0.5	50% to 1000%	Yes	2 days	0%	--	4	100%	4	0.5,1,1,2
MM.11	1	50% to 1000%	--	--	0%	--	4	100%	4	0.5,1,1,2

8.5.1. Impact of Matchmaking Algorithm

This section studies the impact of matchmaking algorithm on system performance when laxity is varied. The results in this section refer to Case MM.1 in Table 8.1. As expected, Figure 8.1 shows that for every algorithm with the increase in laxity, work rejected W_R decreases significantly while U (Figure 8.2) increases. These figures also

show that the matchmaking algorithm can significantly affect system performance justifying the use of an intelligent Matchmaker at MMC. At L = 1000% for example, the spread in W_R is over 25% of W_R of RAN. The comparison of the algorithms shows that when laxity 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) and it increases U of each cluster by 2% over that of the next best

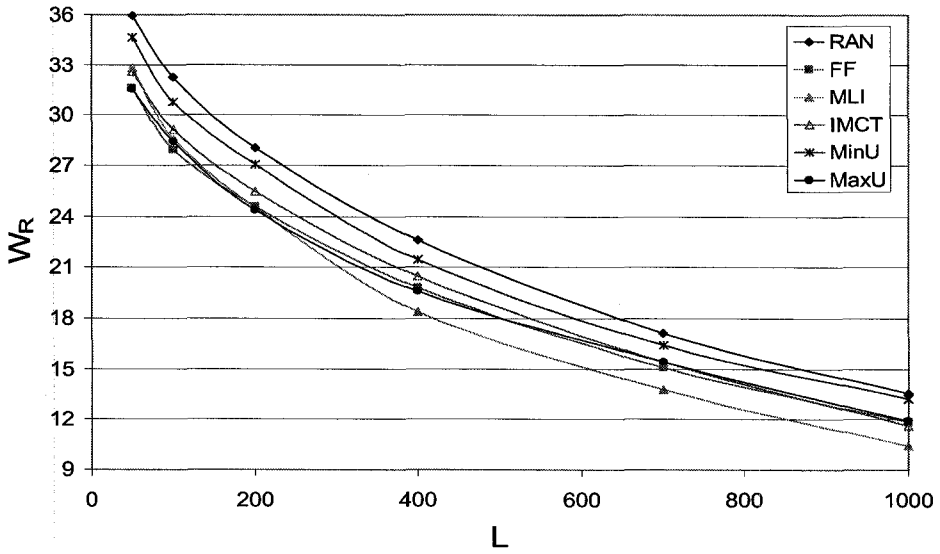


Figure 8.1: Impact of Matchmaking Algorithm on Work Rejected

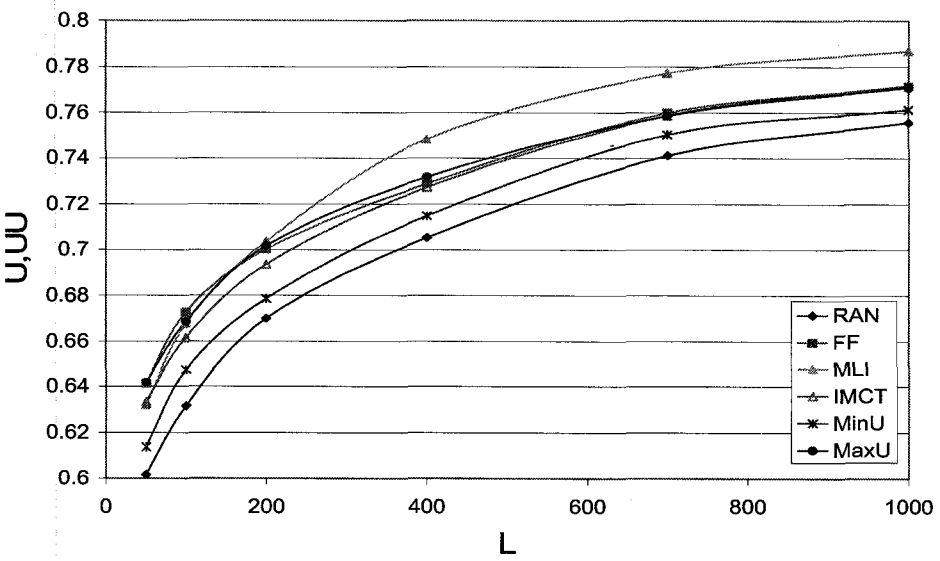


Figure 8.2: Impact of Matchmaking Algorithm on Utilization

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 try to 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 the figure to avoid cluttering, the performance of NF is close to that of RAN.

Figure 8.3 shows that MLI results in the lowest response times for ARs. 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 8.4 shows that when L is small, Q_R 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_R approaches 1. As FF and MaxU retain enough spare capacity on some of the clusters to accommodate large jobs, their Q_R 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_R higher than those achieved by FF and MaxU.

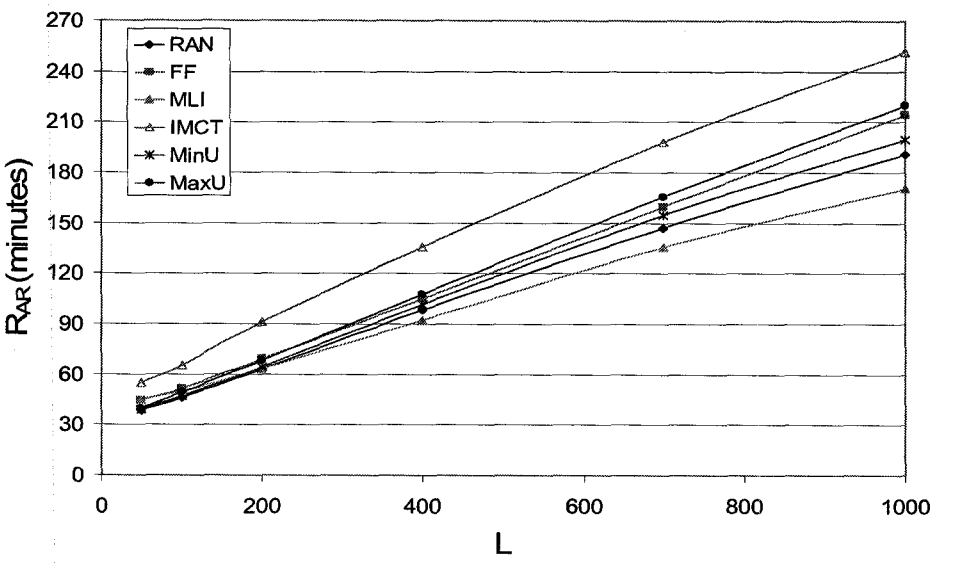


Figure 8.3: Impact of Matchmaking Algorithm on Response Time of Advance Reservations

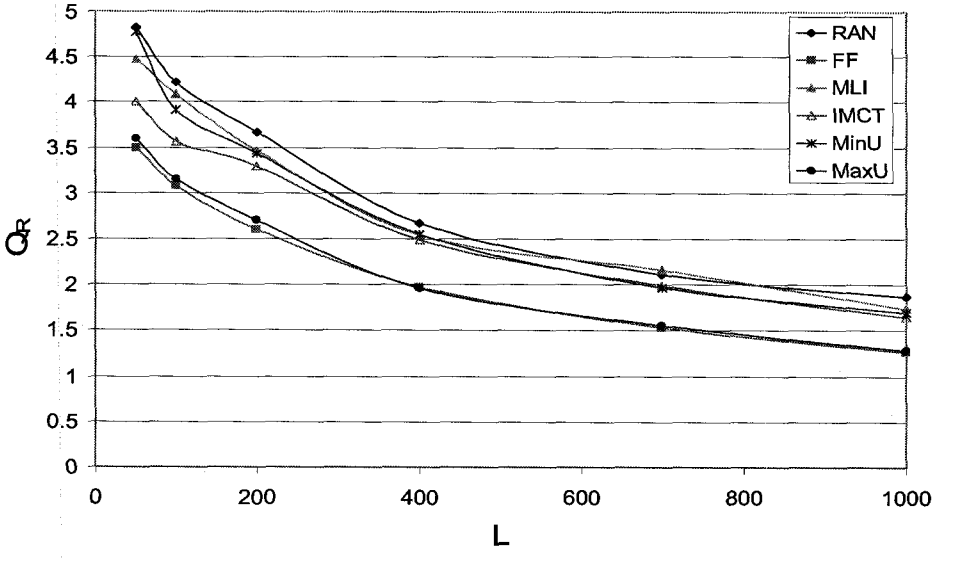


Figure 8.4: Impact of Matchmaking Algorithm on Fairness

As there was no error in estimated runtimes in this set of experiment, no jobs are aborted by any of the resources and hence $W_A = 0$, $P_A = 0$ and $U = UU$ for all values of L.

P_b curves (not presented to save space) show that P_b does not change significantly with the increase in L. It can be shown that $P_b = W_R /(100*Q_R)$ and hence as both W_R and Q_R decreases at almost same rate with the increase in L, P_b tends to remain constant. MLI results in the lowest P_b values.

8.5.2. Effect of Mixed Workload

This section presents results obtained by using a mix of advance reservation and on-demand (OD) requests. The results correspond to Case MM.2 in Table 8.1.

Figure 8.5, Figure 8.6, Figure 8.7 and Figure 8.8 show that, for this case, trends of the curves for W_R, U, R_{AR} and Q_R are the same as in Case MM.1 with PAR = 1. However, with 50% OD requests in Figure 8.5 W_R is significantly lower (by up to

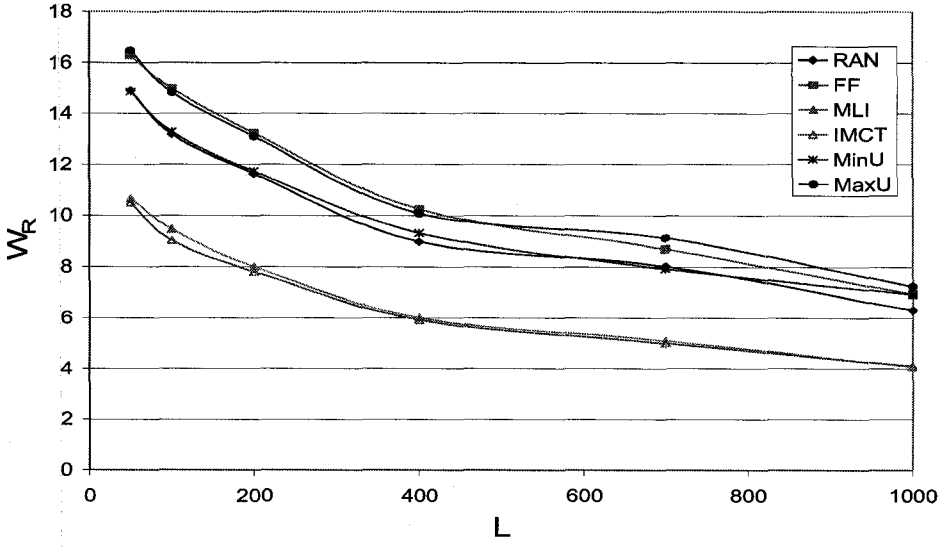


Figure 8.5: Effect of Mixed Workload on Work Rejected

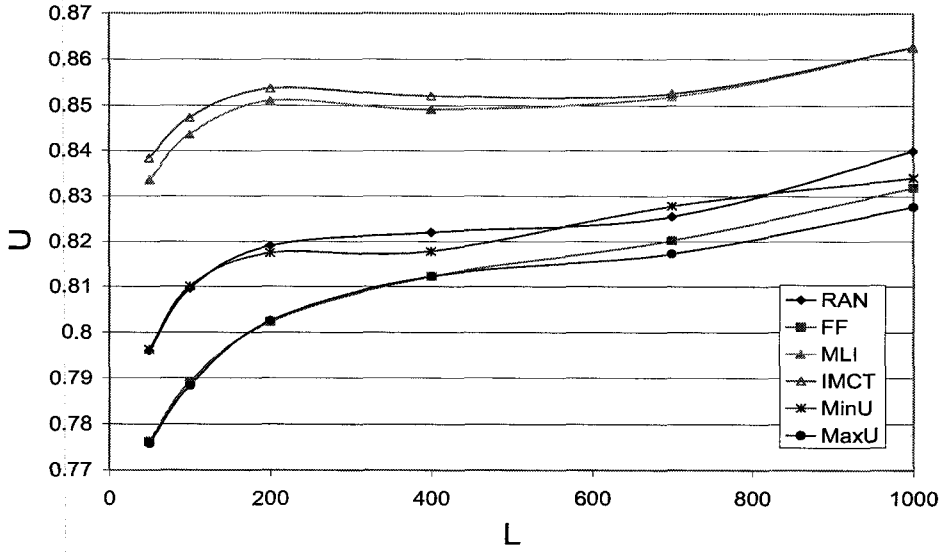


Figure 8.6: Effect of Mixed Workload on Utilization

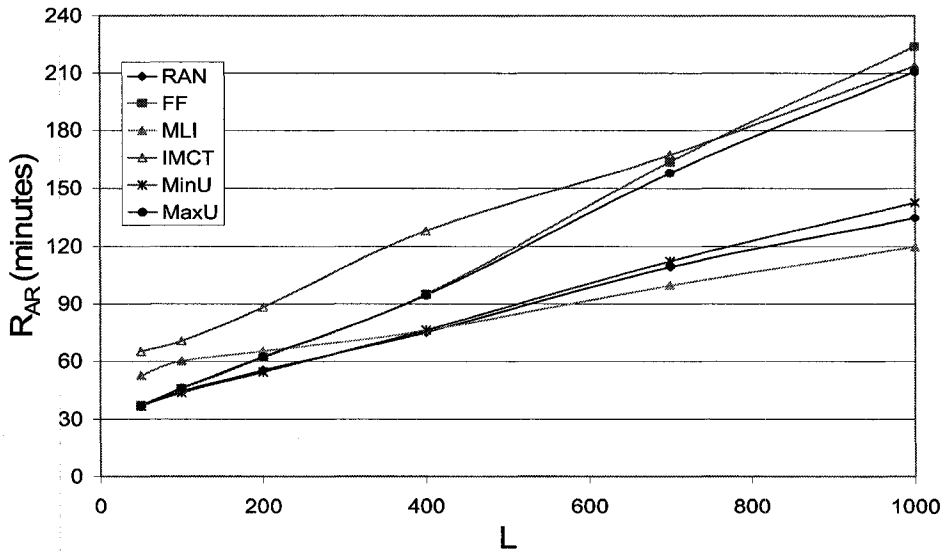


Figure 8.7: Effect of Mixed Workload on Response Time of Advance Reservations

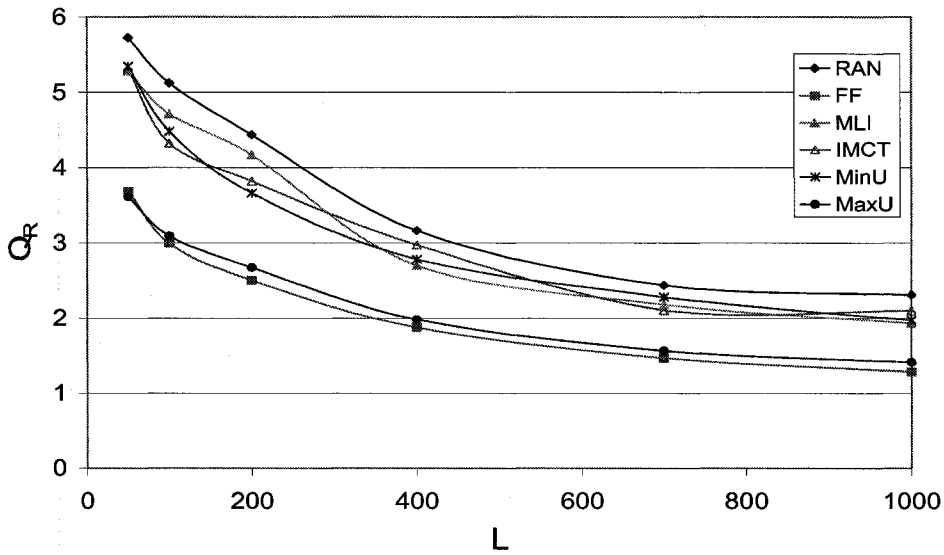


Figure 8.8: Effect of Mixed Workload on Fairness

66.74%) and U in Figure 8.6 is substantially higher (by up to 30.63%). This is because OD requests have less QoS constraints and hence provide more flexibility in scheduling.

The graphs show that when OD requests 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 8.5 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 OD requests 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 the worst R_{AR} for the reasons described in Section 8.5.1.

Figure 8.9 shows that MLI results in the lowest response time for on-demand requests.

8.5.3 Impact of Size of Candidate set

This section studies the impact of size of candidate set N on performance. The results in this section correspond to Case MM.3 in Table 8.1. N is varied as a parameter in these experiments. Hence, to keep the comparison fair, the amount of total work is adjusted by changing the arrival rate such that the maximum possible utilization of clusters remains the same for all values of N.

The results show that when N is increased, W_R decreases (Figure 8.10) while U increases (Figure 8.11). This is due to the positive effect of having greater number of

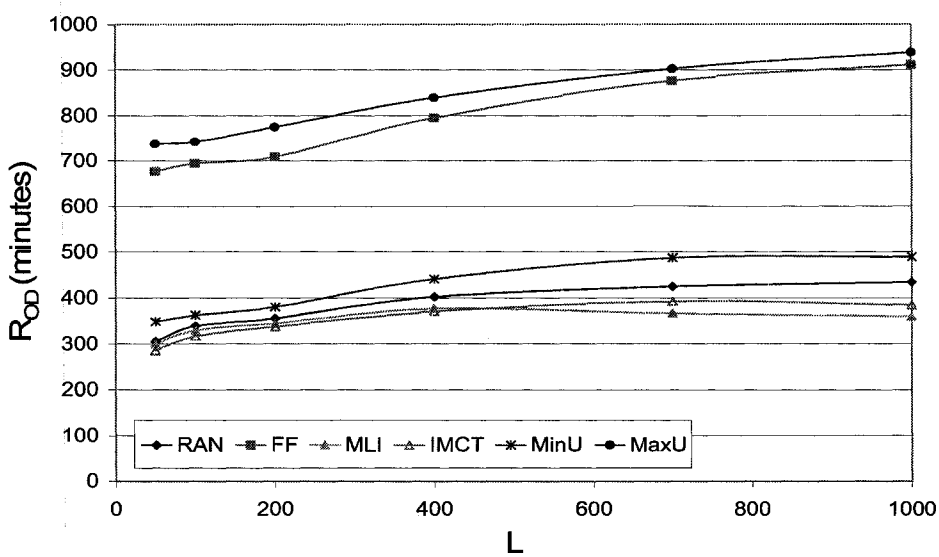


Figure 8.9: Effect of Mixed Workload on Response Time of On-Demand Jobs

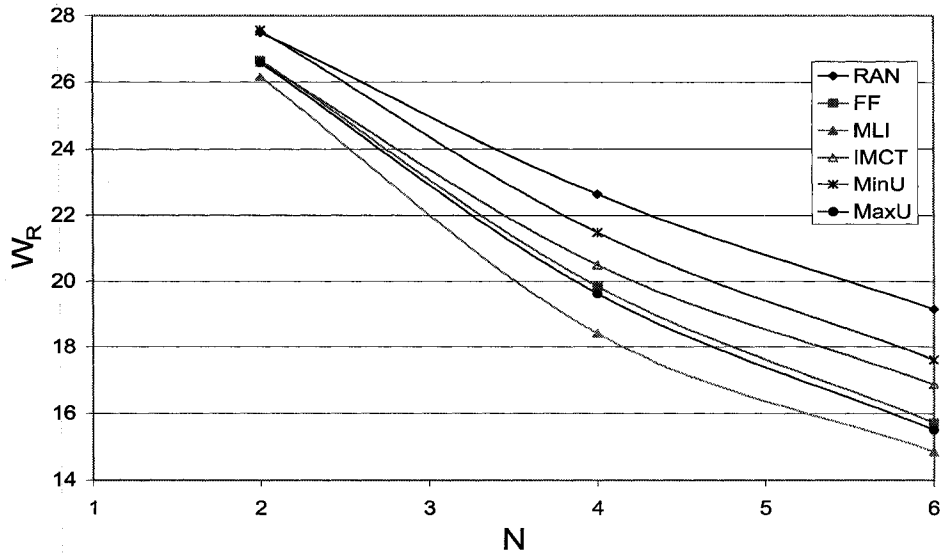


Figure 8.10: Impact of Size of Candidate set on Work Rejected

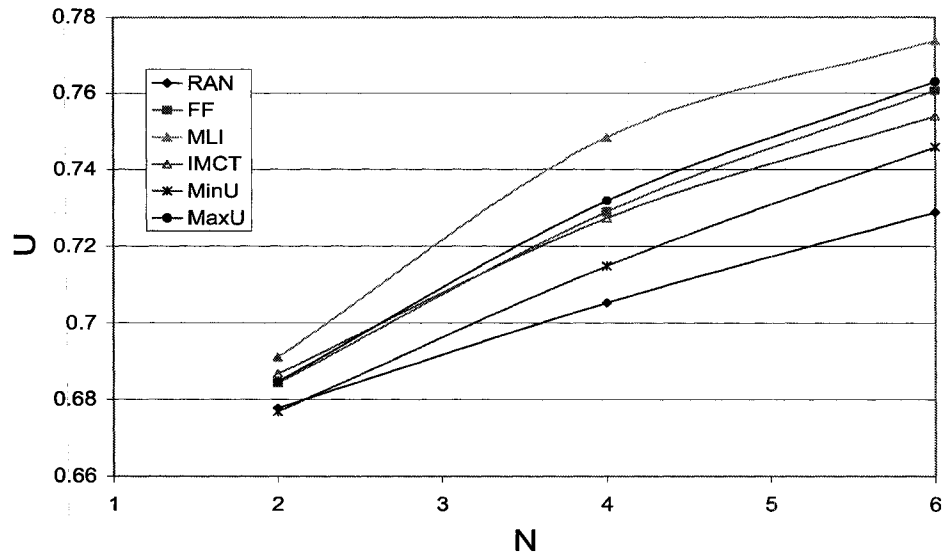


Figure 8.11: Impact of Size of Candidate set on Utilization

options to fit the job in. Increasing the size of candidate set thus results in a better fit. However, candidate set size cannot be increased arbitrarily because of the overheads involved in querying and matching. Figure 8.10, Figure 8.11 and Figure 8.12 also show that irrespective of N, MLI performs the best among all other algorithms giving lower W_R and R_{AR} and higher U. Figure 8.12 shows that with the increase in N, R_{AR} increases slightly. This is because with the increase in N, U increases and hence there is a greater

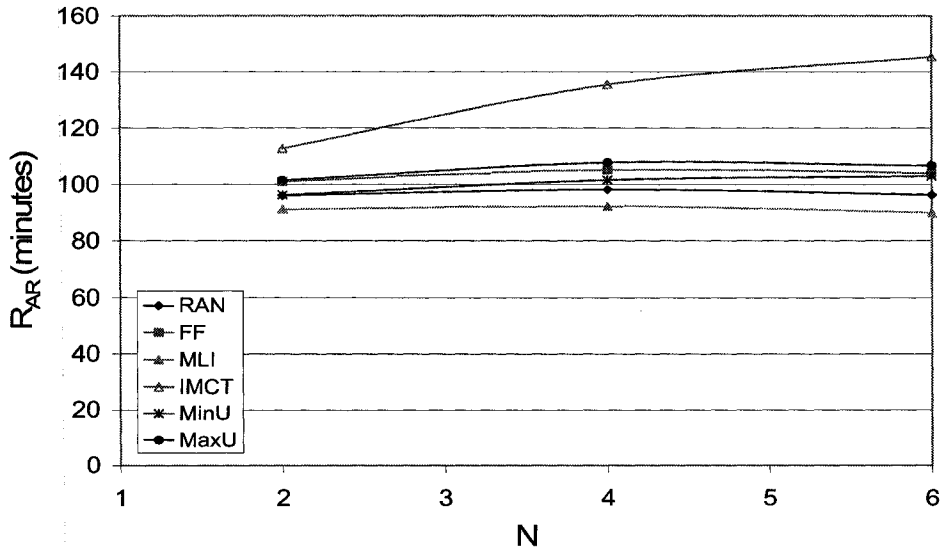


Figure 8.12: Impact of Size of Candidate set on Response Time of Advance Reservations

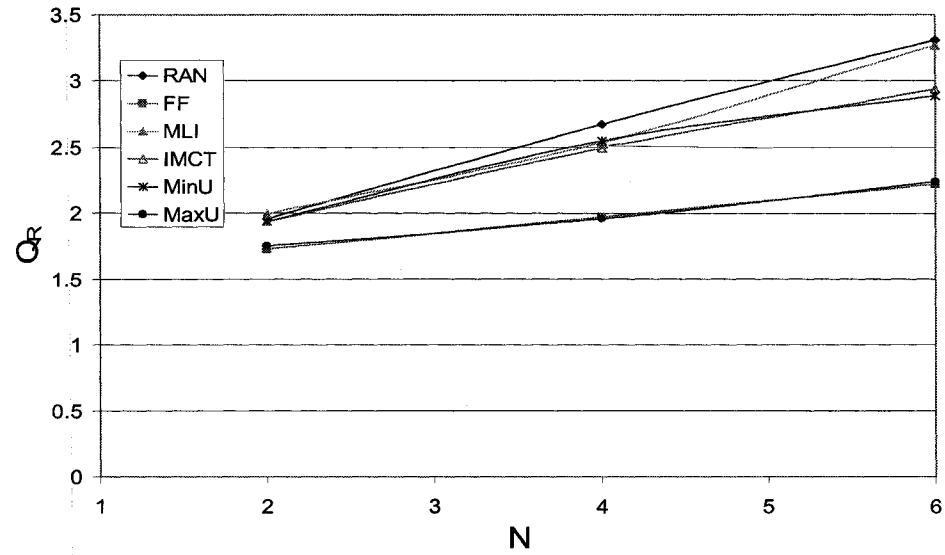


Figure 8.13: Impact of Size of Candidate set on Fairness

number of jobs in each cluster competing for the resource. With IMCT, R_{AR} increases at a higher rate with the increase in N. As explained in Section 8.5.1, this is because IMCT often ends up selecting a cluster in which in order to accommodate the new job in the schedule the scheduler pushes jobs already in schedule towards their deadlines. Thus as U increases with the increase in N, the phenomenon becomes more pronounced and R_{AR} for IMCT increases at a very high rate.

Figure 8.13 shows an increase in Q_R with the increase in N. As N increases the probability of rejecting a small job decreases because there are more options to fit the job in. This increases U and decreases spare capacity for accommodating a large job. Hence, as N increases the system becomes biased towards small jobs irrespective of the algorithm used.

8.5.4 Impact of Arrival Rate

This section studies the impact of arrival rate on system performance. The results presented in this section correspond to Case MM.4 in Table 8.1. In these experiments, the arrival rate is varied to vary the maximum possible utilization of the system. A higher arrival rate corresponds to a higher maximum possible utilization and vice versa. System behavior is investigated at different maximum possible utilization values.

Figure 8.14 shows that when arrival rate is increased, W_R increases. A higher arrival rate means that the system is subjected to a higher total work and since the system has only a limited capacity, an increase in arrival rate results in an increase in W_R. The lower than 1 slopes of W_R curves in Figure 8.14 indicate that as the number of arrival increases not all the new jobs are rejected by the system. This result is confirmed by Figure 8.15 which shows an increase in U with an increase in arrival rate. As arrival rate increases, the system is able to accept more jobs. This is because given a higher number of job arrivals there is a higher probability of fitting one of the jobs in one of the idle segments of one of the resources. However, after a certain point the system tends to saturate and the rate of increase in U with an increase in the arrival rate decreases. The results also show that irrespective of the maximum possible utilization of the system, MLI results in the lowest W_R values and the highest U values.

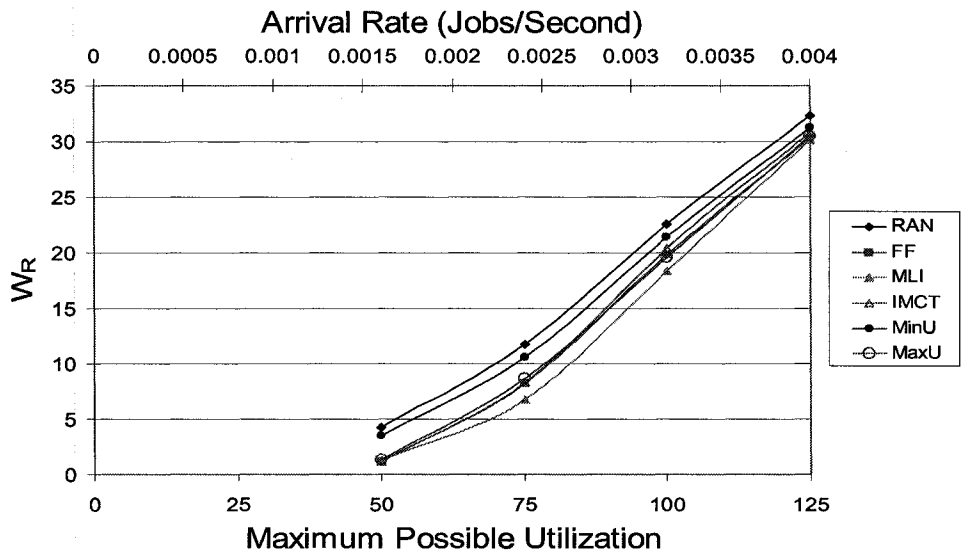


Figure 8.14: Impact of Arrival Rate on Work Rejected for Case MM.4

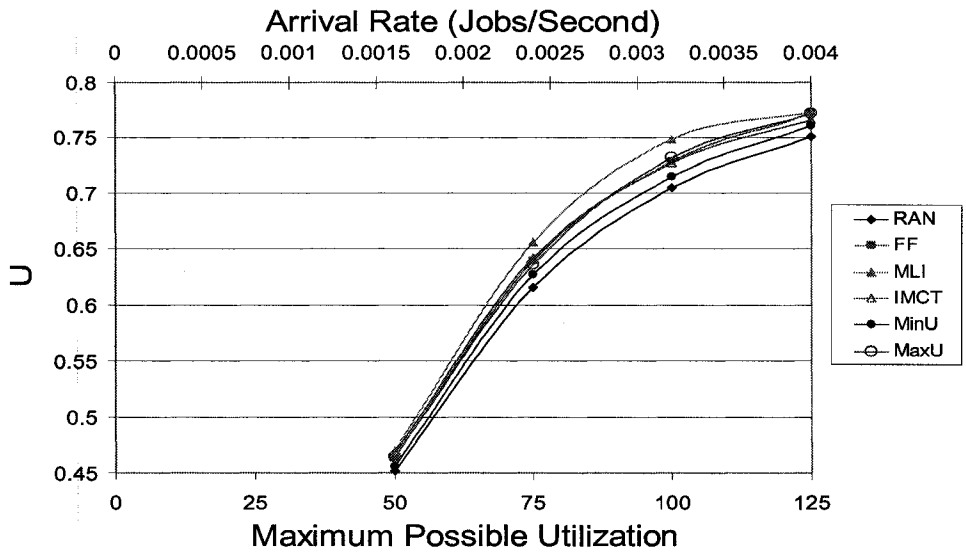


Figure 8.15: Impact of Arrival Rate on Utilization for Case MM.4

As the system utilization increases with an increase in the arrival rate so does the contention for the resources. Figure 8.16 thus shows an increase in R_{AR} with an increase in arrival rate. As in Section 8.5.1 and Section 8.5.3, by economically using laxity of jobs, MLI results in the lowest R_{AR} values irrespective of the job arrival rate.

Figure 8.17 shows that with an increase in arrival rate Q_R values decrease and approach 1. The reason for this decrease has been discussed in Section 6.3.3.

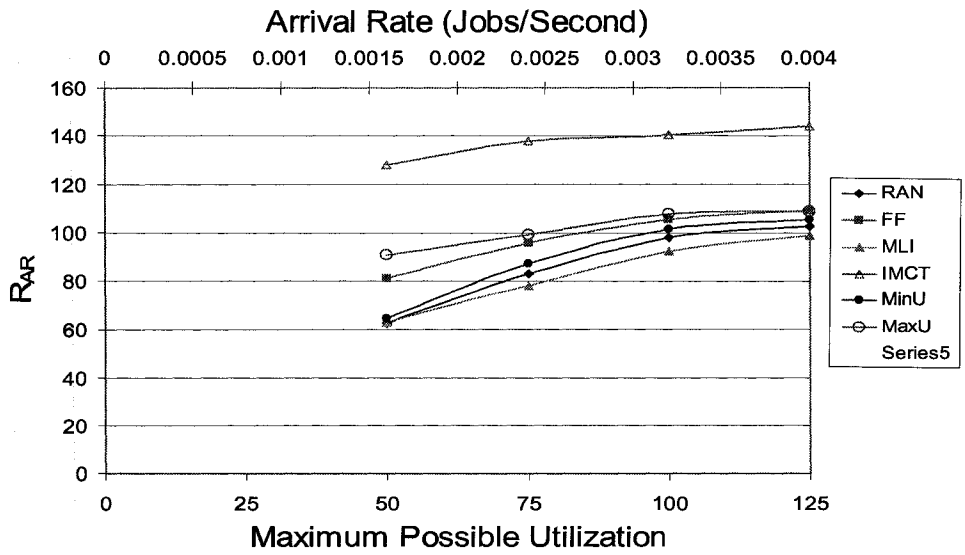


Figure 8.16: Impact of Arrival Rate on Response Time of Advance Reservations for Case MM.4

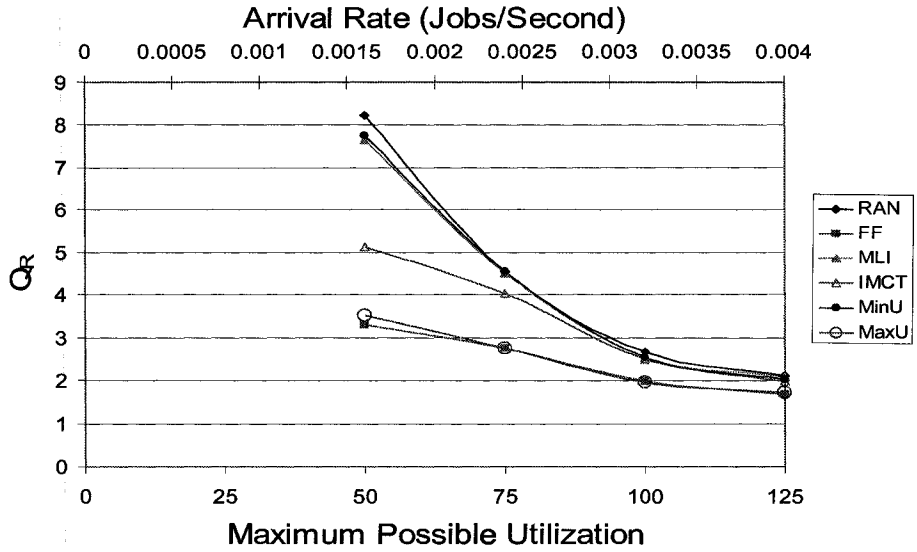


Figure 8.17: Impact of Arrival Rate on Fairness for Case MM.4

Results for Case MM.5 are presented in Appendix D.1. They are similar to those obtained for Case MM.4.

8.5.5. Impact of Error in User-Estimated Runtimes

This section investigates the impact of error in estimated runtimes on performance. The results correspond to Case MM.6 in Table 8.1. The results in this section are compared with Case MM.7 in Table 8.1. For a fair comparison, the results for

these cases (MM.6 and MM.7) were obtained using exactly the same workload including the same job arrival times, earliest start times, estimated runtimes, actual runtimes and laxities. The only difference is that for Case MM.6 SEM was not activated while for Case MM.7 SEM was plugged in to adapt the resource schedule to the changing conditions.

Figure 8.18 and Figure 8.19 shows that W_R and U for Case MM.6 have same trends as those of W_R and U obtained with complete *a priori* knowledge of job runtimes

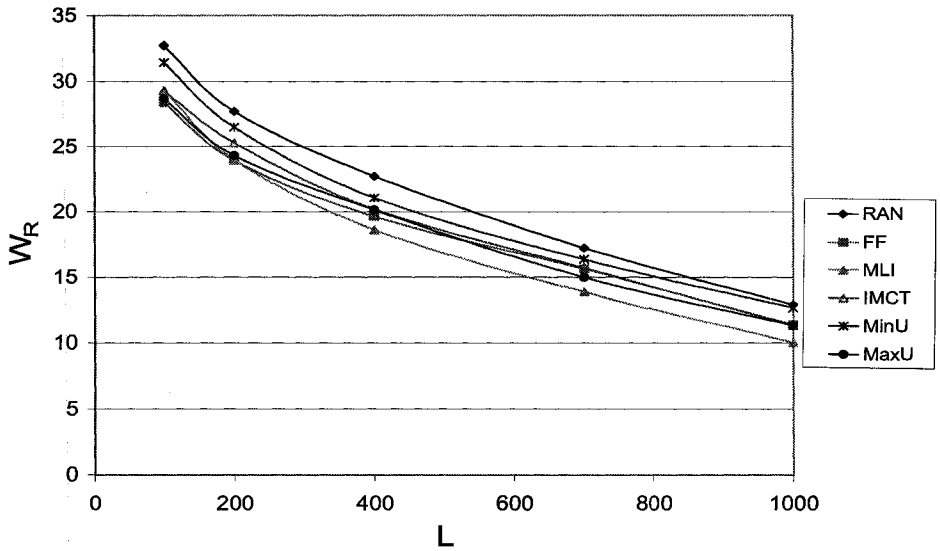


Figure 8.18: Work Rejected for Case MM.6

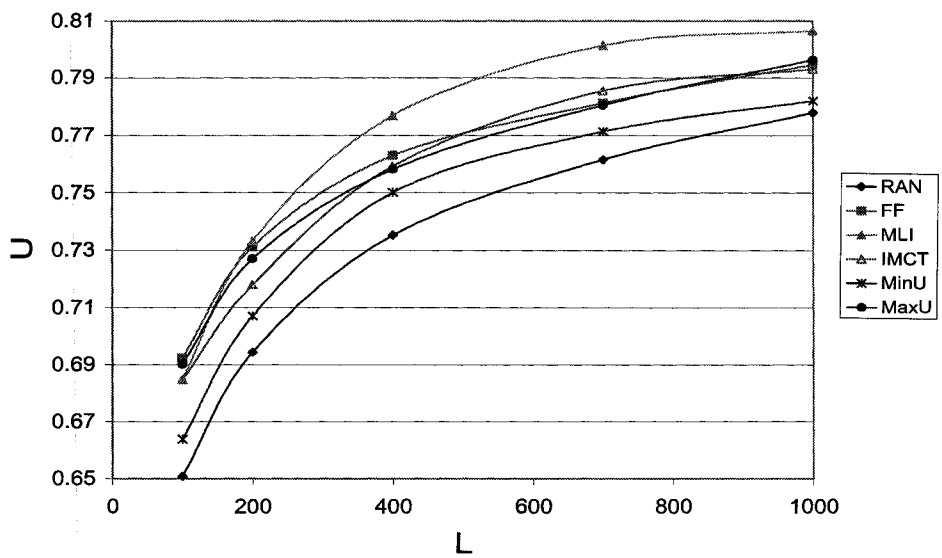


Figure 8.19: Utilization for Case MM.6

discussed in detail in Section 8.5.1. However, since in this case a number of jobs accepted by the system are aborted, the useful utilization UU of the system in Figure 8.21 is substantially lower than that in Figure 8.2. For example with MLI, for any given value of L, UU in Figure 8.21 is approximately 45% lower than that in Figure 8.2. This shows that if not handled properly, even small errors in user-estimated runtimes can severely degrade system performance.

The results also show that even in the presence of errors in estimated runtimes, MLI outperforms all other algorithms and results in lower W_R and higher U and UU.

Figure 8.20 shows that with the increase in L, W_A increases. The reason is that with the increase in L the percentage of work accepted increases and so does the amount of work with underestimated runtimes. Since all jobs with underestimated runtimes are aborted, W_A increases with L.

The results show that R_{AR} in Figure 8.22 display the same trend as R_{AR} in Figure 8.3 while Q_R in Figure 8.23 is similar to Q_R in Figure 8.4.

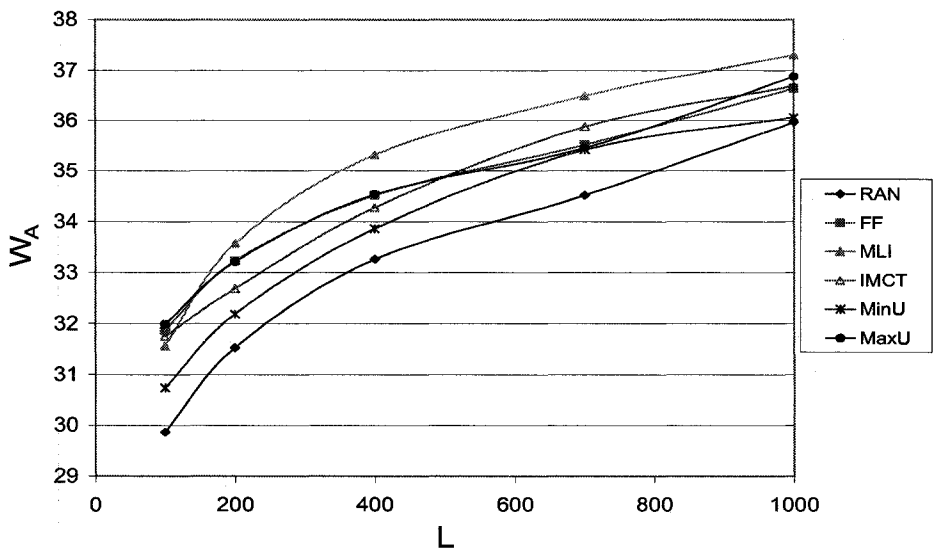


Figure 8.20: Work Aborted for Case MM.6

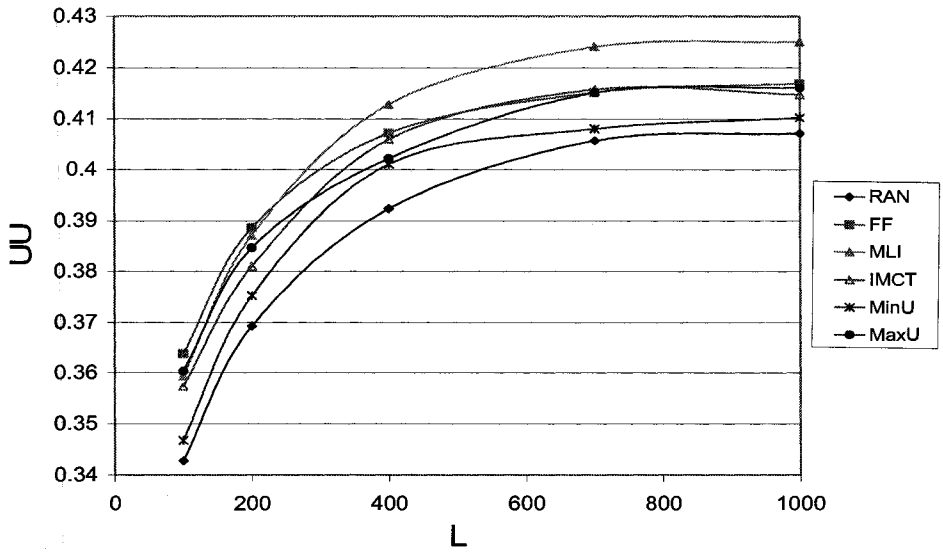


Figure 8.21: Useful Utilization for Case MM.6

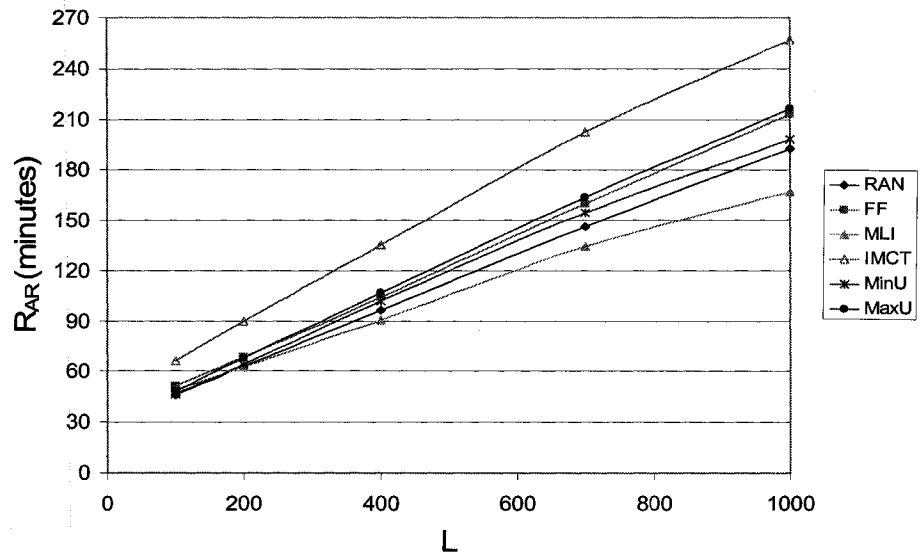


Figure 8.22: Response Time of Advance Reservations for Case MM.6

8.5.6. Performance of Schedule Exceptions Manager

This section presents the results for Case MM.7 in Table 8.1. These results are obtained with exactly the same workload and same values of all parameters as in Section 8.5.5 but by activating SEM that monitors the resource schedule and adapts it to the changing conditions.

The results presented in Figure 8.24, Figure 8.25, Figure 8.28 and Figure 8.29 show that W_R, U, R_{AR} and Q_R in this case are almost the same as obtained without activating SEM in the MM.6 case discussed in Section 8.5.5. However, for any given value of L, W_A in Figure 8.26 is substantially lower than that in Figure 8.20. For example, with MLI the work aborted in Figure 8.26 is only 41% to 59% of that aborted in Figure 8.20. As a result, useful utilization UU of the system in Figure 8.27 is significantly

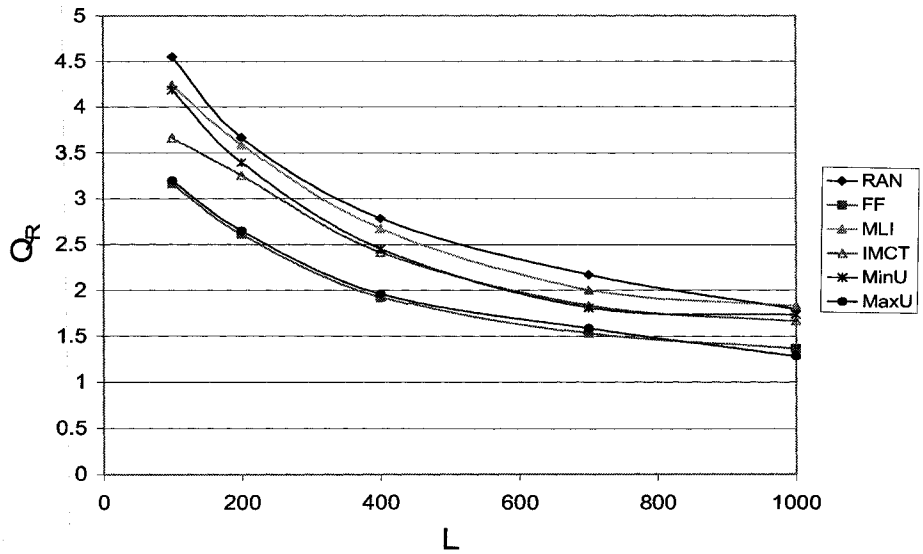


Figure 8.23: Fairness for Case MM.6

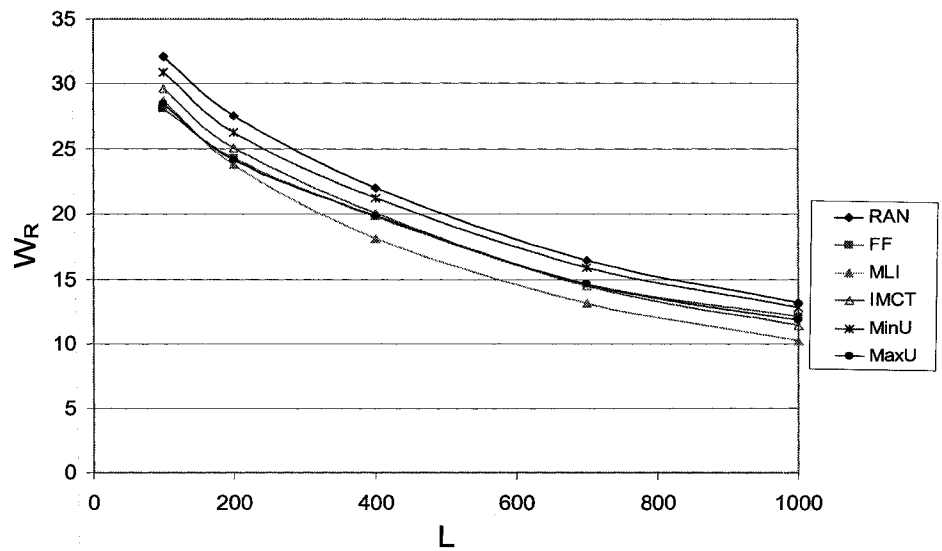


Figure 8.24: Work Rejected for Case MM.7

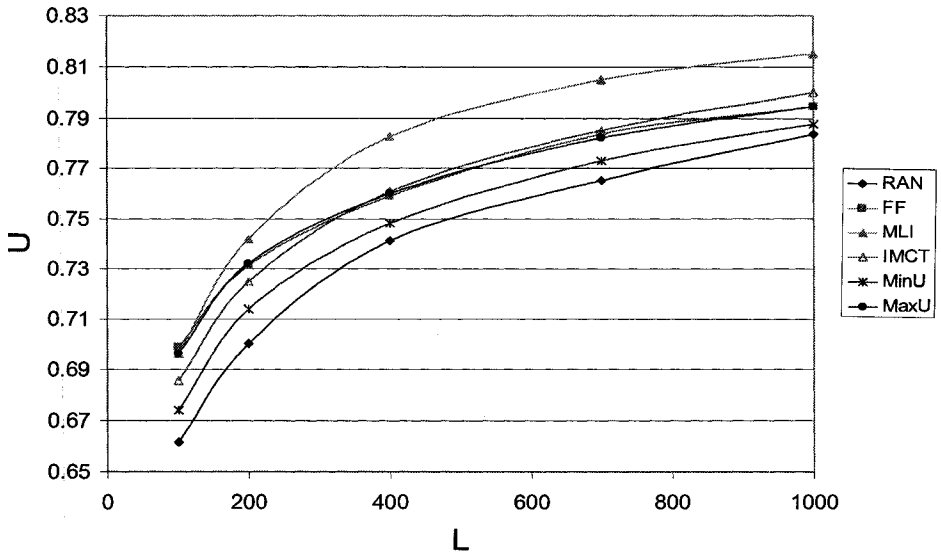


Figure 8.25: Utilization for Case MM.7

higher than that obtained in Figure 8.21. With MLI for example, UU of resource with SEM is 1.38 to 1.56 times of that obtained without SEM. This shows that SEM is effective in bringing UU of the resource in the presence of errors in user-estimated runtimes close to that obtained when the estimated runtimes of the jobs are exactly equal to their actual runtimes.

The results also show that for any given value of L, MLI results in the highest UU among all other algorithms. By effectively utilizing laxity of jobs, MLI not only results in the acceptance of a larger number of jobs but also allows the extension of the runtimes of underestimated jobs and hence results in the highest UU.

Figure 8.26 shows that with the increase in L, W_A increases. The reason for an increase in W_A with an increase in L has been discussed in detail in Section 7.5.2.

8.5.7. Impact of Heterogeneous Environments

This section studies the performance of matchmaking algorithms in heterogeneous environments consisting of different speed resources. The results in this section correspond to Case MM.8 and Case MM.9 in Table 8.1.Results for Case MM.8

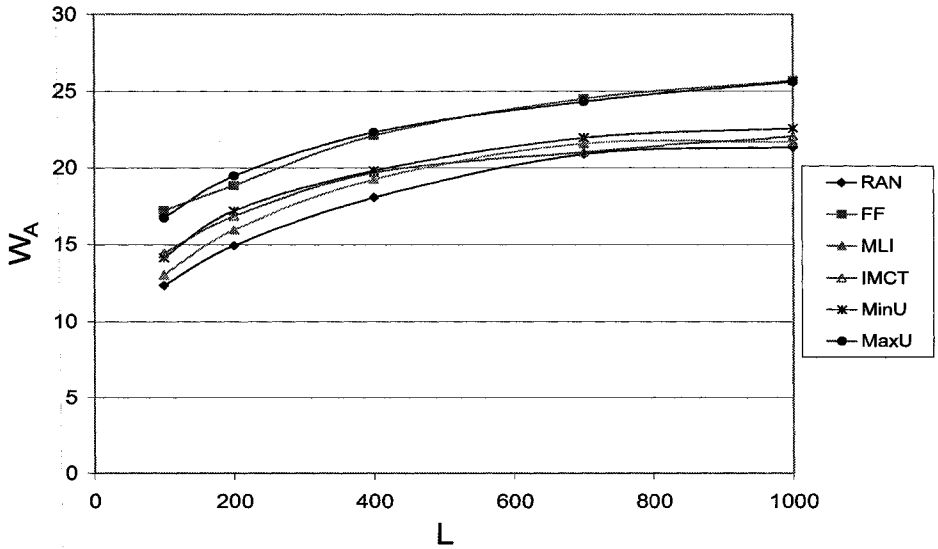


Figure 8.26: Work Aborted for Case MM.7

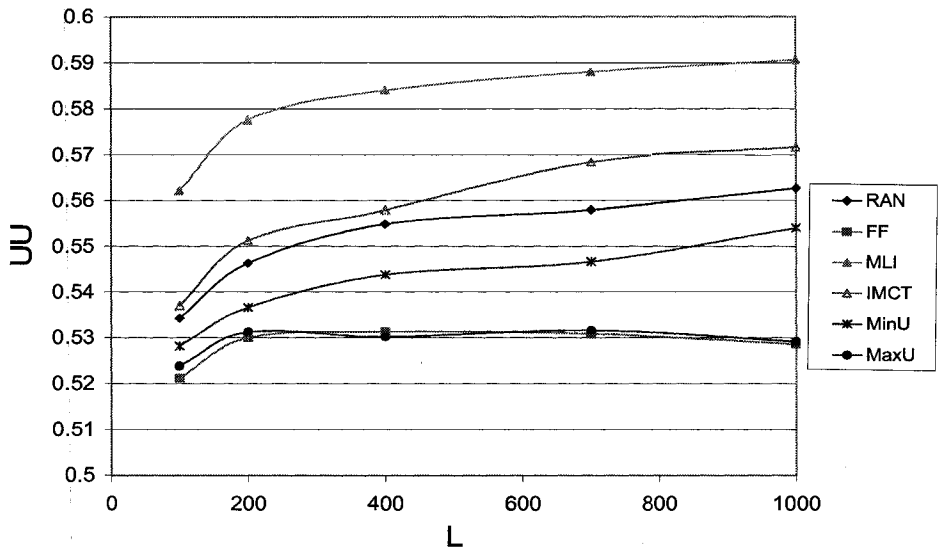


Figure 8.27: Useful Utilization for Case MM.7

are presented in Figure 8.30, Figure 8.31, Figure 8.32 and Figure 8.33 while those for Case MM.9 are presented in Figure 8.34, Figure 8.35 and Figure 8.36.

Figure 8.30 shows that even in a heterogeneous environment, W_R shows the same trend as was observed in a homogeneous environment in Section 8.5.2. Figure 8.30 also shows that MLI and its variants result in the lowest W_R values.

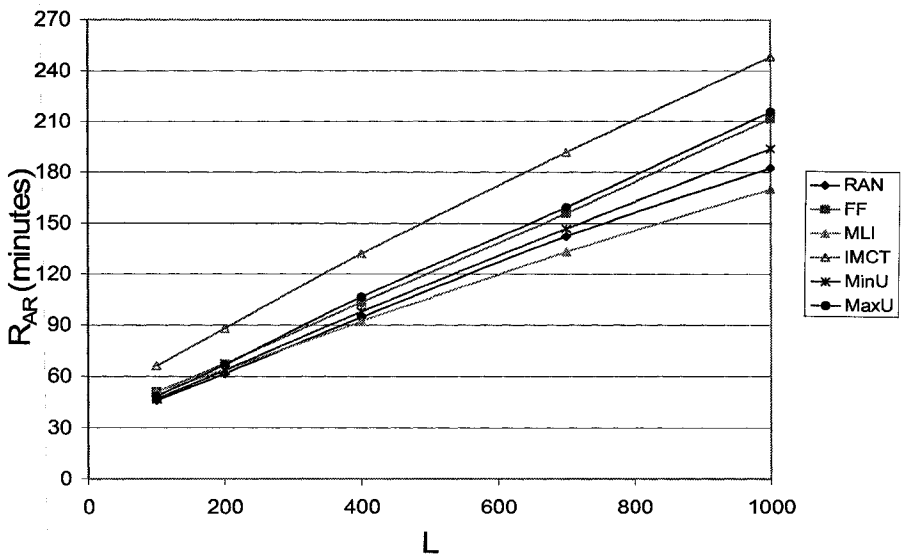


Figure 8.28: Response Time of Advance Reservations for Case MM.7

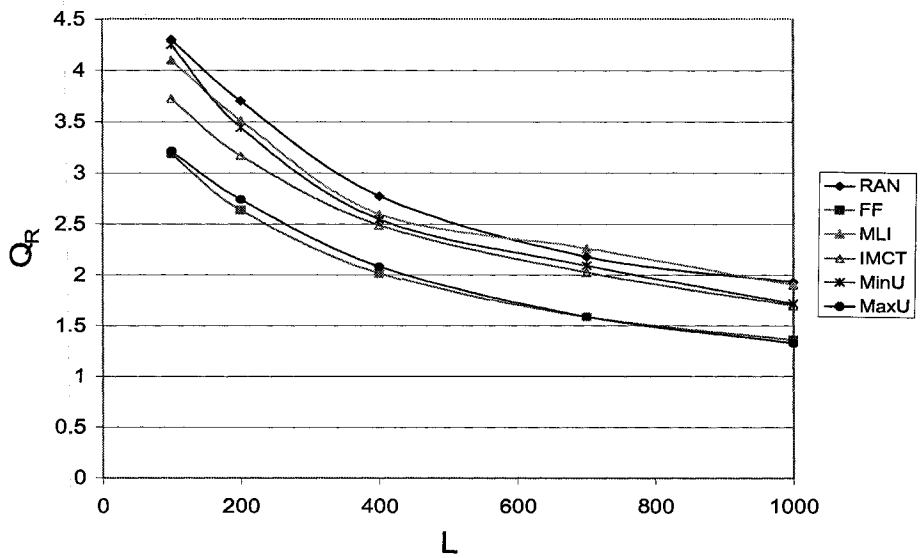


Figure 8.29: Fairness for Case MM.7

Figure 8.31 which is a zoomed version of Figure 8.30, compares the performance of five best algorithms for L greater than 400%. Although not shown in Figure 8.30 and Figure 8.31 to avoid cluttering, the performance of MLI-SB is close to that of MLI-SA. Figure 8.31 shows that performance of MLI-SA is comparable to that of MLI. This is because factors A and B presented in Section 8.4, tend to cancel each other out and hence scaled versions of MLI result in the same performance as MLI. This experimentally

verifies the conclusion in Section 8.4 that contrary to intuition no scaling is required in MLI.

The results show MLI-SDP outperforms all other algorithms and results in a substantially lower W_R. For example, for L equal to 700%, W_R with MLI-SDP is merely 13% of that of MLI and MLI-SA (the next best algorithms). The high performance of MLI-SDP can be attributed to effective dynamic partitioning of resources and intelligent

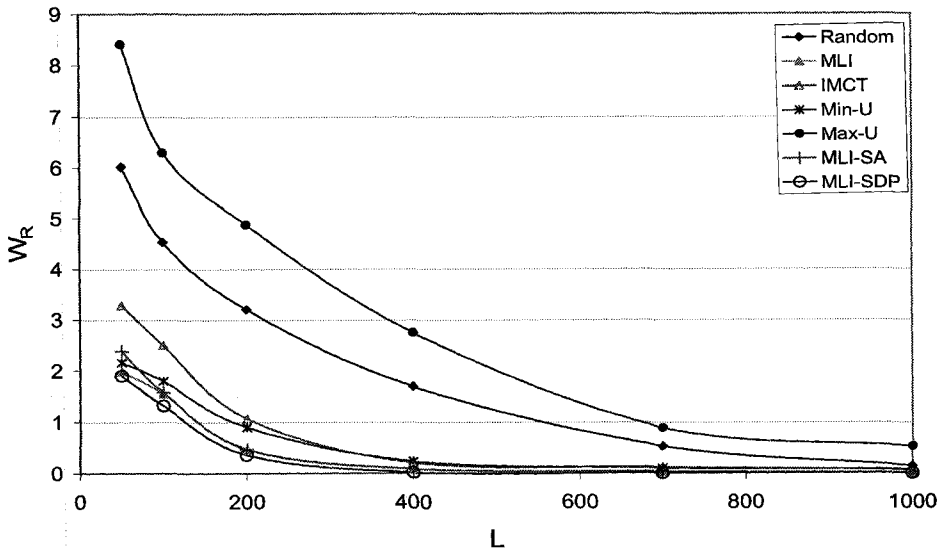


Figure 8.30: Work Rejected for Case MM.8

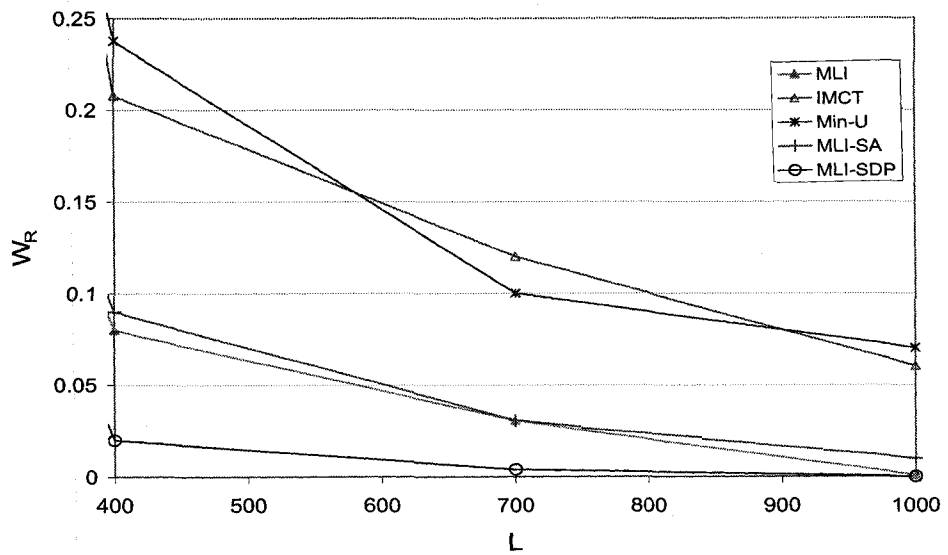


Figure 8.31: A Closer Look at Work Rejected for Case MM.8

use of laxity of jobs as discussed in Section 8.4.3.

Figure 8.32 shows that MLI-SA and IMCT result in the lowest R_{AR} values. MLI-SDP results in slightly higher R_{AR} values. This is because MLI-SDP results in the lowest W_R and hence the highest utilization. Higher utilization increases contention for the resources and hence R_{AR} increases.

Figure 8.33 shows that IMCT, MLI and its variants result in the lowest R_{OD}

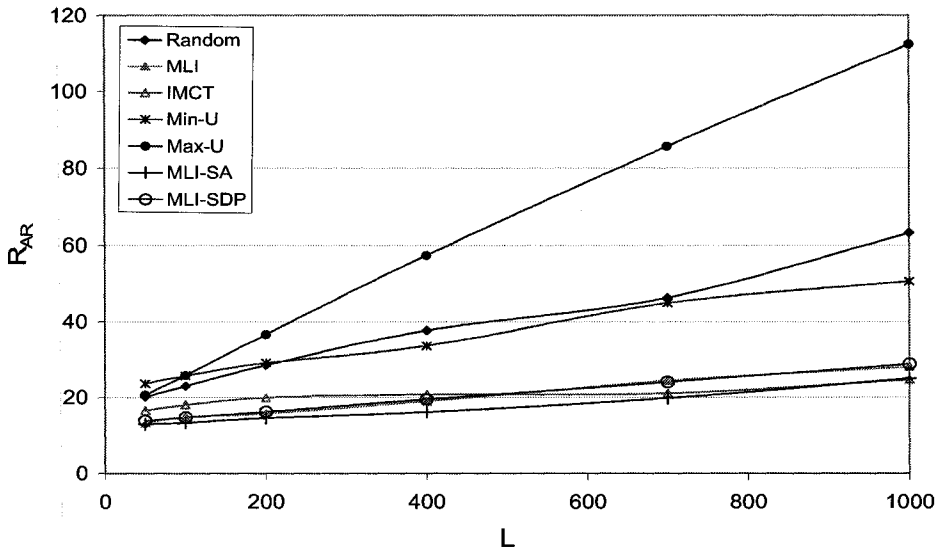


Figure 8.32: Response Time of Advance Reservations for Case MM.8

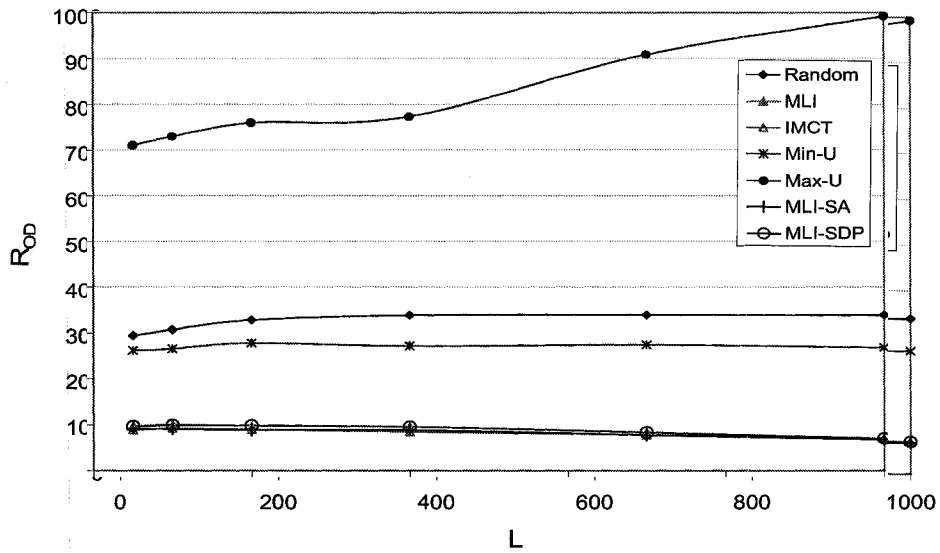


Figure 8.33: Response Time of On-Demand Requests for Case MM.8

values.

W_R in Figure 8.34 for Case MM.9, where PAR is equal to 1, is similar to that in Figure 8.30. However, the results show that the spread in W_R in Figure 8.34 is smaller than that was observed in Figure 8.30 and Figure 8.1. This shows that for high PAR values in a highly heterogeneous environment, intelligence in matchmaking has a smaller

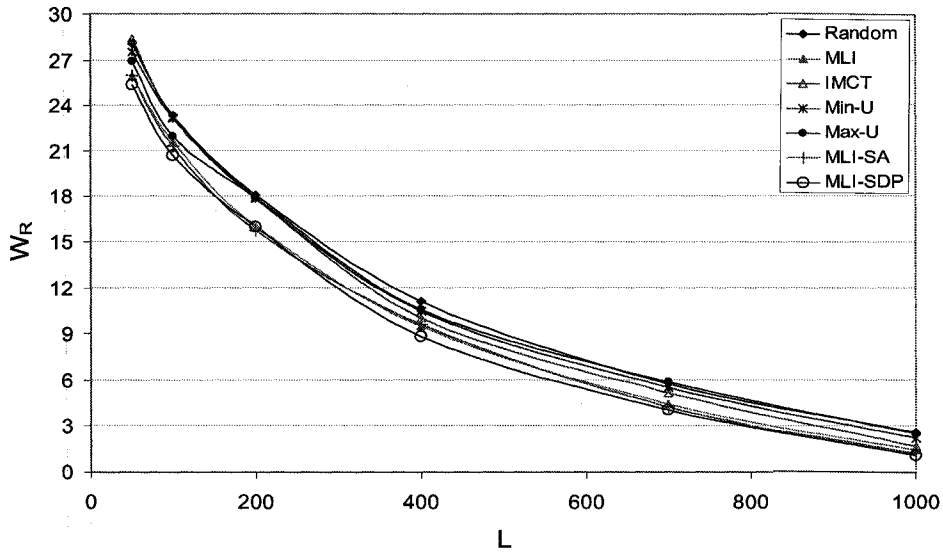


Figure 8.34: Work Rejected for Case MM.9

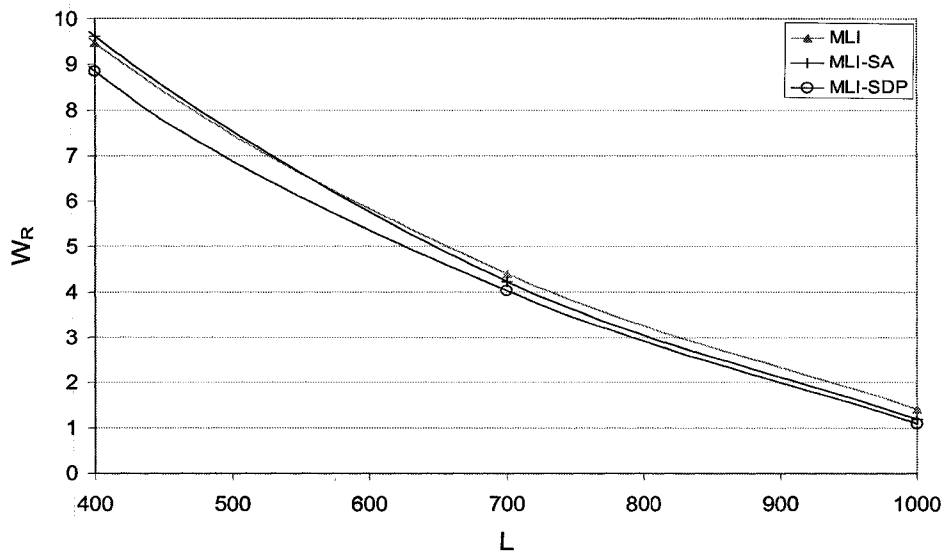


Figure 8.35: A Closer Look at Work Rejected for Case MM.9

effect on performance than in a homogenous environment. Figure 8.35, a zoomed version of Figure 8.34, attests to the conclusion that scaled versions of MLI result in performance close to that of MLI while MLI-SDP that dynamically partitions the resources results in a better performance.

Figure 8.36 shows that MLI and its variants result in the lowest R_{AR} values.

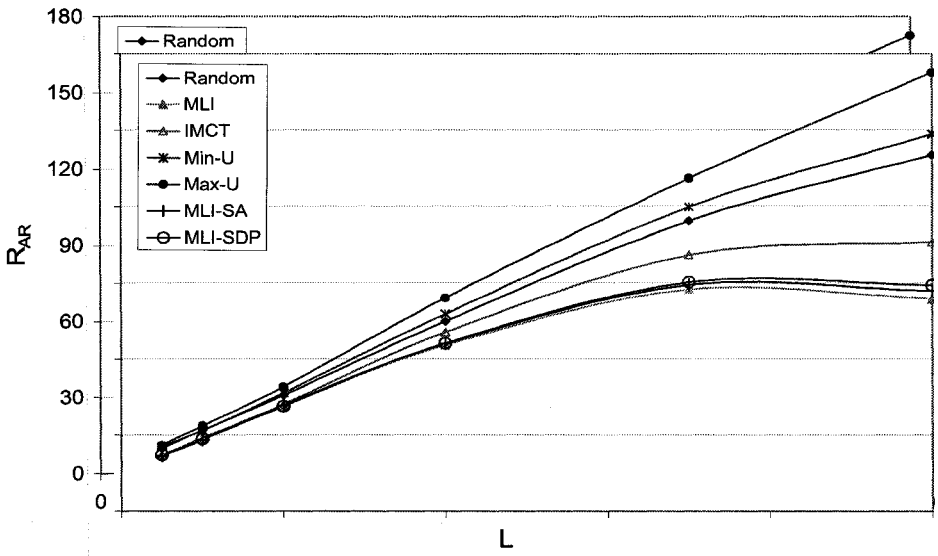


Figure 8.36: Response Time of Advance Reservations for Case MM.9

Results for Case MM.10 and Case MM.11, where H is equal to 4, are presented in Appendix D.2. The results show similar trends as are observed in this section and yields similar conclusions.

8.6. Summary

Effective application-to-resource mapping and load balancing are important to achieve the best possible performance. The results in this chapter show that traditional matchmaking algorithms used in Grids and distributed computing in general do not perform well when applied in multi-institutional settings particularly for workloads consisting of both advance reservations and best-effort jobs. This chapter investigated several matchmaking algorithms and presented a novel algorithm, MLI, for matchmaking

that outperforms every other algorithm investigated in the thesis in almost every respect for a wide range of workload parameters. Performance analysis of the matchmaking algorithms brings important insights into the system behavior.

The results in Section 8.5.1 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%. The results also show that traditional matchmaking algorithms, such as MinU, do not perform well when ARs are introduced in the system. MLI performs the best among all matchmaking algorithms. For L = 1000%, MLI results in a W_R which is only 89.2% of the next best algorithm, it increases U of each cluster by 2% over that of the next best algorithm and still results in the lowest R_{AR} values. MLI gives the best performance even when a significant number of OD requests are introduced in the system. The high performance of MLI can be attributed to the intelligent use of laxity of jobs. Another application of MLI is as an effective meta-scheduler within Platform Computing's Community Scheduler Framework (CSF) [PLA05c] that is used to dispatch jobs to resource managers in multiple domains.

This chapter extends matchmaking algorithms to cater to the needs of heterogeneous environments and shows that contrary to intuition no scaling is required in MLI for such environments. The chapter presented MLI-SDP, a variant of MLI, which accommodates a self dynamic partitioning mechanism. The results show MLI-SDP outperforms all other algorithms investigated in heterogeneous environments. For example, in Figure 8.32, for L equal to 700%, W_R with MLI-SDP is merely 13% of that of the next best algorithm.

Chapter 9

Resource Co-Allocation in Multiple Domains

9.1. Co-Allocation Problem in Multiple Domains

As discussed in Section 3.4.3, a matchmaker can be effectively used for scenarios where an application requires a single resource or where different jobs of an application are independent of each other such that they do not require simultaneous allocation of resources. However, many Grid applications, such as those in data mining and computational-steering, and workflows may benefit from coordinated resource usage at multiple sites. In situations where an application requires simultaneous allocation of multiple resources from possibly different domains, the Multiple-Resource Coordinator is required. MRC ensures that different jobs of the Grid application are scheduled to execute at exactly the same time on different resources in potentially different domains. The efficiency of the system depends on the co-allocation algorithm used with MRC. The co-allocation algorithm determines a suitable scheduled-time for the application and suitably selects the resources that can start the application at the determined scheduled-time. This thesis presents and analyzes novel co-allocation algorithms that seek to minimize work rejected by the system while meeting the QoS and co-allocation constraints of the applications.

9.2. Resource Co-Allocation with Multi-Resource Coordinator

The most effective way to ensure simultaneous availability of multiple resources in multiple domains is to reserve the resources in advance. This is the approach that was

presented in GARA [FOS99b] and has been adopted in this thesis. In all previous works however, all advance reservation requests including those for co-allocation do not have any laxity. The problem thus is to find and reserve resources that can start the application at its earliest start time. As the results in Chapter 5 and Chapter 6 have shown, advance reservations without laxity result in low utilization of resources and hence do not meet the performance and cost objectives. The results later in this chapter will also show that when co-allocation is done with the aid of advance reservations without laxity, system performance degrades significantly. Just like applications consisting of single jobs or multiple independent jobs, termed as *non-co-allocation requests*, this thesis focuses on advance reservations with laxity for co-allocation requests, to help prevent performance deterioration. However, for non-co-allocation requests the domain scheduler is free to re-order and reschedule jobs to improve system utilization as long as it meets the deadline constraints of all jobs. Different resources required to satisfy a co-allocation request may be in multiple domains. Hence, if such a re-scheduling of co-allocation requests is allowed at an individual resource level without any co-ordination with other resources where other jobs of the co-allocated application are allocated, there is no guarantee that all resources will start the application at exactly the same time. The problem of co-allocation with the aid of under-constrained ARs is hence more complex in comparison to the problem of co-allocation with the aid of ARs without laxity. Approaches such as those where each resource seeks to co-ordinate with other resources each time it re-schedules a co-allocation request are not likely to be scalable. This is because re-scheduling of one co-allocation request at one resource may result in re-scheduling of

many co-allocation requests at another resource and re-scheduling of each such request may result in re-scheduling of many other requests at some other resource and so on.

The Multi-Resource Coordinator (MRC) component of this thesis solves the problem of co-allocation with ARs with laxity by dividing the co-allocation process into three phases. The co-allocation requests are submitted to MRC and at the arrival of every such request, MRC first queries all resources in the candidate set to determine all feasible scheduled-times at each resource. This constitutes the first phase. Note that during the first phase the co-allocation request is treated as a non-co-allocation request and each resource obtains all feasible scheduled-times for the given request based on the earliest start time, estimated execution time and deadline of the request. During the second phase, MRC selects a suitable scheduled-time for the co-allocation request based on the feasible scheduled-times of each resource obtained during the first phase. It is assured that at least the required number of resources can start the application at the determined scheduled-time. MRC then selects suitable resources from the candidate set that can start the application at the determined scheduled-time. During the third phase, reservations are made on the resources selected at the end of the second phase. The start time of these reservations is selected as the scheduled-time determined during the second phase. The deadline is selected as the scheduled-time plus the estimated execution time. Hence, during the third phase laxity is taken out of the reservation window to assure that the local schedulers do not reschedule a co-allocation request. As the results in Section 9.4 will show, such a three phase approach is effective in maintaining high system performance and meeting co-allocation and QoS constraints of the applications.

9.3. Algorithms for Resource Co-Allocation in Multiple Domains

As discussed in Section 9.2, the second phase of co-allocation with MRC consists of determining a suitable scheduled-time for the application and selection of resources that can start the application at the determined scheduled-time. The goal of a co-allocation algorithm is to determine the suitable scheduled-time and resources for each co-allocation request in such a way that system performance is maximized. This section proposes a number of co-scheduling algorithms. As we can expect a typical Grid workload to consist of non-co-allocation and co-allocation requests, the performance of the system is investigated by using different combinations of matchmaking and co-allocation algorithms.

9.3.1. Random/Random (RAN/RAN)

In RAN/RAN, the Random algorithm presented in Section 8.2.1 is used for matchmaking non-co-allocation requests. For co-allocation requests, the earliest possible scheduled-time is determined such that the number of resources that can start the application at that time is greater than or equal to the number of resources required by the co-allocation request. Resources required by the co-allocation request are randomly chosen from the set that can start the application at the determined scheduled-time.

9.3.2. Minimum Laxity Impact with Self Dynamic Partitioning/Random (MLI-SDP/RAN)

In MLI-SDP/RAN, the MLI-SDP algorithm presented in Section 8.4.3 is used for matchmaking non-co-allocation requests. For co-allocation requests, the Random co-allocation algorithm outlined in Section 9.3.1 is used. Note that for scenarios where

heterogeneity of the system is equal to 1, MLI-SDP automatically reduces to MLI (presented in Section 8.3).

9.3.3. Minimum Laxity Impact with Self Dynamic Partitioning/Cost Based – Low End (MLI-SDP/CB-LE)

In MLI-SDP/CB-LE, the MLI-SDP algorithm presented in Section 8.4.3 is used for matchmaking non-co-allocation requests. For co-allocation request, a cost is assigned to each combination of the required number of resources in the candidate set and the algorithm selects the lowest cost combination for co-allocation. This assignment of cost is based on the relative speed of resources. Consider for example a scenario in which a co-allocation request requires two resources. If the two resources selected from the candidate set for this co-allocation request have the same speeds, then the time required to execute each job of the co-allocation request on each of the resources is the same. On the other hand, if the speed of one of the resources is 10 times the speed of the other resource, then the higher speed resource can finish the job in $1/10^{th}$ of the time than the lower speed resource. However, since both of these resources are to be co-allocated for the complete execution of the application, to allow for inter-job communication for instance, resource power will be unnecessarily wasted on the higher speed resource. The cost based co-allocation algorithm tries to minimize such unnecessary wastage of resource power resulting from the co-allocation of jobs on resources with mismatching speeds.

For this purpose, all the resources in the candidate set are assigned a speed factor r based on their relative speeds. The speed factor r is assigned to a resource such a way that for any two resources b_y and b_z the ratio of the speed of b_y and b_z is equal to the ratio of r_y and r_z. This can be achieved by assigning a speed factor of 1 to the standard resource and

assigning every other resource a speed factor equal to the ratio of the speed of the resource and the speed of the standard resource. Thus, resources with same speeds are assigned the same speed factor. The number of resources required by the application is chosen from the candidate set. Since there can be different combination of such resources, each combination is assigned a cost. The cost for each combination consists of a primary cost, a secondary cost and a tertiary cost. The lowest speed resource in the combination is called the base resource and its speed factor is represented by r_b. The primary cost is calculated as follows:

$$Primary\ Cost = \sum_{(For\ all\ resources\ bz\ in\ the\ combination\ excluding\ the\ base\ resource)} r_z / r_b \qquad \textbf{(9.1)}$$

The secondary cost is calculated as follows:

$$Secondary\ Cost = r_b \qquad \textbf{(9.2)}$$

If the combination with the lowest primary cost is selected for co-allocation, wastage of resource power can be minimized. This is because the execution time of the co-allocation request is equal to the execution time of the request on the base resource r_b. From equation 9.1, the higher the primary cost the greater the difference between the speeds of the base resource and the other resources selected for co-allocation. As discussed earlier in this section, a high difference between the speeds of the resources selected for co-allocation results in an unnecessary wastage of resource power.

The algorithm determines the earliest scheduled-time for the co-allocation request on the combination with the lowest primary cost such that each resource in the combination can start the application at exactly the same time. If no such time exists, the combination with the next lowest primary cost is selected and so on. If two combinations have the same primary cost, the combination with the lower secondary cost is selected. In CB-LE since secondary cost is equal to r_b, combinations involving lower speed resources

are given preference over those involving higher speed resources. If both the primary cost and secondary cost for two combinations happen to be the same, tertiary cost is used to break the tie. The tertiary cost is equal to the sum of impact of laxity ζ of the co-allocation request on each of the resources in the combination. The impact of laxity ζ on each resource is calculated with the help of Equation 8.4. The combination with the lowest tertiary cost is selected.

In a homogeneous environment, with heterogeneity H equal to 1, all the resources will have equal speeds and thus the primary cost and secondary cost will always be the same. Hence, in such environments it is the tertiary cost that decides the selection of resources for co-allocation.

Note that it is assumed that each job of the co-allocation request requires the same amount of execution. For scenarios where different jobs of a co-allocation request have different resource demands, the cost-based algorithm presented can be adapted. Such an adaptation is proposed as a future work.

9.3.4. Minimum Laxity Impact with Self Dynamic Partitioning/Cost Based – High End (MLI-SDP/CB-HE)

MLI-SDP/CB-HE is similar to MLI-SDP/CB-LE presented in Section 9.3.3. The only difference is that if two combinations have the same primary cost, MLI-SDP/CB-HE gives preference to the combination involving the higher speed resource. For this purpose, the secondary cost of a combination in MLI-SDP/CB-HE is calculated by taking the reciprocal of the speed factor of the highest speed resource of the combination. The rest of the algorithm is the same as MLI-SDP/CB-LE.

9.3.5. Minimum Laxity Impact with Self Dynamic Partitioning/Cost Based with Dynamic Timing– Low End (MLI-SDP/CB-DT-LE)

MLI-SDP/CB-DT-LE is similar to MLI-SDP/CB-LE presented in Section 9.3.3. The only difference is that once the combination of resources has been selected for co-allocation, instead of scheduling the job at the earliest feasible scheduled-time on the combination, MLI-SDP/CB-DT-LE determines the cost of scheduling a job at different feasible times. This cost is calculated for each feasible scheduled-time where all the resources in the combination can start the application. The cost is calculated by adding the impact of laxity ζ of the co-allocation request on each of the resources in the combination. The impact of laxity ζ is calculated with the help of Equation 8.4. It has already been shown in Chapter 8 that no scaling of ζ is required for heterogeneous environments.

The feasible scheduled-time that results in the lowest cost is selected as the scheduled-time of the co-allocation request. CB-DT-LE thus seeks to further lower the performance cost of co-scheduling in comparison to CB-LE.

9.4. Performance Analysis of Multi-Resource Coordinator

Table 9.1 summarizes the sets of experiments conducted with MMC for studying the performance of the Multi-Resource Coordinator. The setup for these experiments has been discussed in detail in Section 4.1.2 and Section 4.4.7, while the workload model has been discussed in Section 4.3.2. The results in Section 6.3.1 show that the EDF-BF combination of heuristics for the GSD algorithm results in the lowest W_R and the highest U. Thus, for the experiments in this Chapter, the EDF-BF combination of heuristics was used with GSD for scheduling jobs on individual clusters.

For these experiments, PAR = 1, N = 4, B = 0% and L for non-co-allocation requests is equal to 400%. For the cases with H equal to 8, relative speeds of the

resources are 1, 2, 4 and 8. The resource with speed 1 is the standard resource. The arrival rate is chosen such that if all requests are non-co-allocation requests, the maximum possible utilization of the system is 100%.

Case MR.1 and Case MR.3 in Table 9.1 investigate, for different values of PCR, the performance of the co-allocation algorithms for scenarios involving co-allocation requests with laxity while Case MR.2 and Case MR.4 investigate the performance of the co-allocation algorithms for scenarios involving co-allocation requests without laxity. Case MR.5 studies a special case of a homogenous environment.

Table 9.1: Cases Studied with MMC for Studying the Performance of the Multi-Resource Coordinator

Case No.	L for Co-Allocation Requests	PCR	H
MR.1	400%	0.2	8
MR.2	0%	0.2	8
MR.3	400%	0.4	8
MR.4	0%	0.4	8
MR.5	400%	0.2	1

9.4.1. Impact of Co-Allocation Algorithms

This section studies the impact of co-allocation algorithms on performance when co-allocation requests have laxity in their reservation windows. The results in this section correspond to Case MR.1 in Table 9.1.

Figure 9.1 shows that co-allocation algorithms have a significant impact on work rejected. In Figure 9.1, the spread in W_R is 17.4% of W_R obtained with RAN/RAN. This justifies the use of an intelligent co-allocation mechanism at MRC. W_R obtained with MLI-SDP/CB-DT-LE is the lowest among all other algorithms and is only 82.6% of W_R obtained with RAN/RAN. However, W_R obtained with MLI-SDP/CB-LE is only 1.007

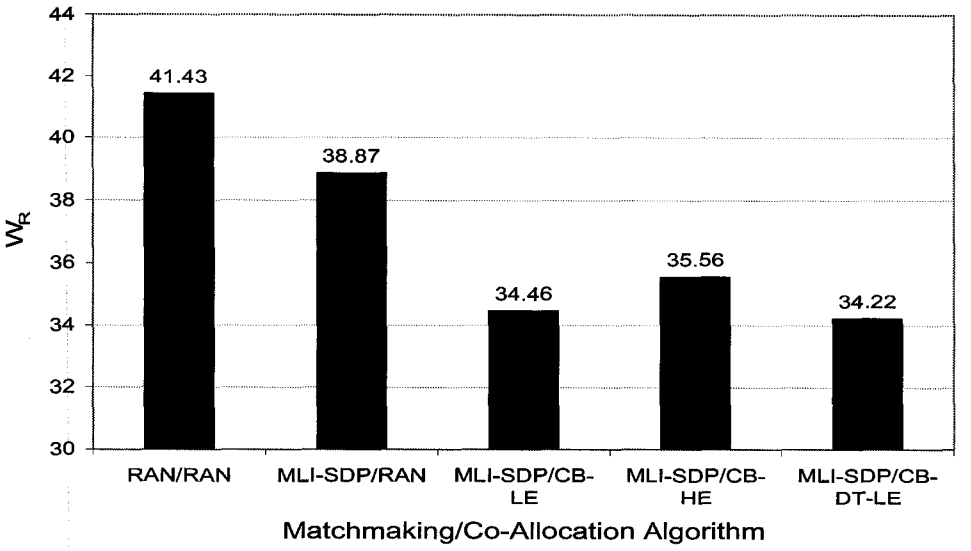


Figure 9.1: Impact of Co-Allocation Algorithms on Work Rejected

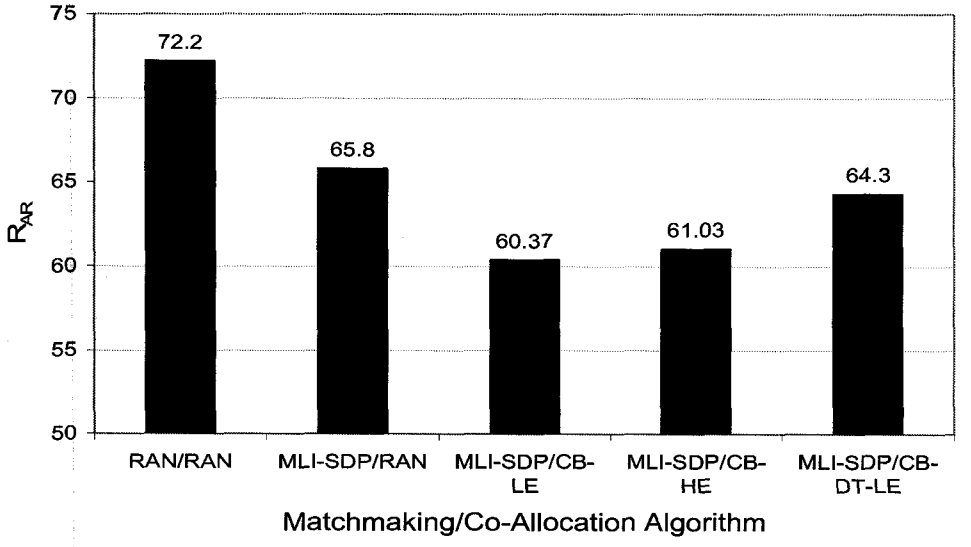


Figure 9.2: Impact of Co-Allocation Algorithms on Response Time of Advance Reservations

times that obtained with MLI-SDP/CB-DT-LE. MLI-SDP/CB-DT-LE is a more expensive algorithm in terms of overheads as it needs to compute cost of multiple feasible scheduled-times before selecting one. This seems to imply that MLI-SDP/CB-LE is the best choice among all other algorithms. In general, the results show that cost based algorithms result in much better performance than others as they seek to reduce performance costs by preventing job co-allocation on mismatching resources and by

economically using laxity of jobs. CB-LE performs better than CB-HE in terms of lower W_R. This can be attributed to the fact that by giving preference to lower speed resources for co-allocation, CB-LE keeps spare capacity on high end resources. That spare capacity is utilized whenever a request cannot be accommodated in lower speed resources. CB-HE on the other hand may result in keeping spare capacity on lower end resources but for jobs with large execution times, lower speed resources may not be enough to meet their deadlines.

The results in Figure 9.1 also show that co-allocation has a tremendous impact on system performance. Case MM.9 in Table 8.1 and Case MR.1 in Table 9.1 are almost identical to each other and the only difference is that in Case MR.1 20% of the requests are co-allocation requests. At L = 400%, W_R obtained with MLI-SDP/CB-DT-LE for Case MR.1 in Figure 9.1 is 3.87 times that obtained for Case MM.9 with MLI-SDP in Figure 8.34 for the same value of L. This shows that even a small percentage of co-allocation requests have substantial performance costs.

Figure 9.2 shows that cost based algorithms results in lower R_{AR} values than other

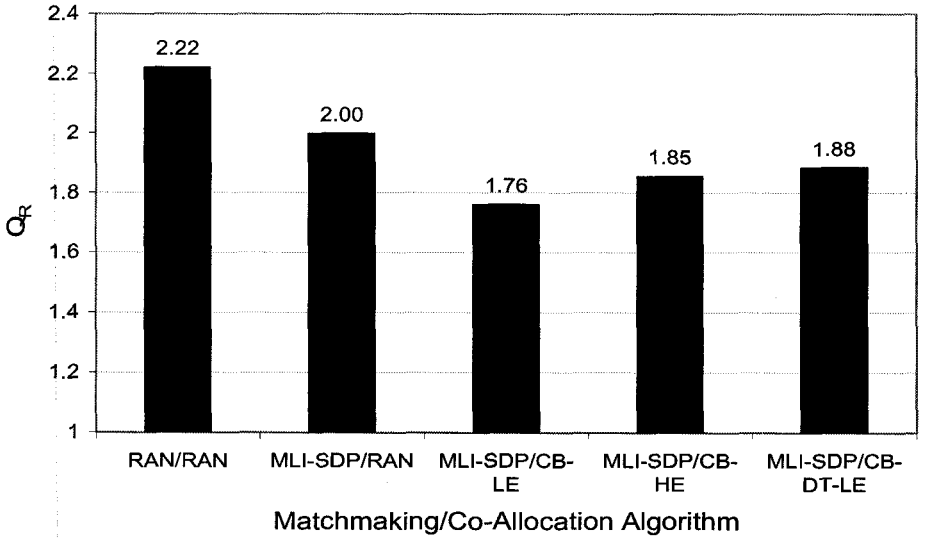


Figure 9.3: Impact of Co-Allocation Algorithms on Fairness

algorithms. MLI-SDP/CB-DT-LE however, results in a higher R_{AR} as it often delays co-allocation requests to schedule them at such a feasible scheduled-time where the impact on laxity is minimum. Figure 9.3 shows that the values of Q_R obtained with different algorithms are greater than 1. This shows that more large jobs are rejected than small ones.

9.4.2. Impact of Co-Allocation Requests with No Laxity

This section studies the impact of co-allocation requests having no laxity on performance. The results in this section correspond to Case MR.2 in Table 9.1.

Figure 9.4 shows that for any given matchmaking/co-allocation algorithm, W_R obtained for this case is much higher than that obtained for Case MR.1 in Figure 9.1. For the values of the parameters used in the experiments, W_R for Case MR.2 is 1.4 to 1.46 times W_R for Case MR.1 depending on the algorithm used for co-allocation. Such a substantial performance cost incurred by co-allocation requests without laxity, for even a small PCR value of 0.2, clearly demonstrates that co-allocation requests with laxity are a better alternative and that the three phase co-allocation procedure presented in this thesis is effective in meeting performance and cost objectives.

Figure 9.4 shows that cost based algorithms result in the lowest W_R values. As co-allocation requests have no laxity in Case MR.2, the behaviors of MLI-SDP/CB-LE and MLI-SDP/CB-DT-LE are identical. This is because the earliest start time of the co-allocation request in the only feasible scheduled-time. W_R obtained with MLI-SDP/CB-LE and MLI-SDP/CB-DT-LE is the lowest and is only 86.2% of that obtained with RAN/RAN.

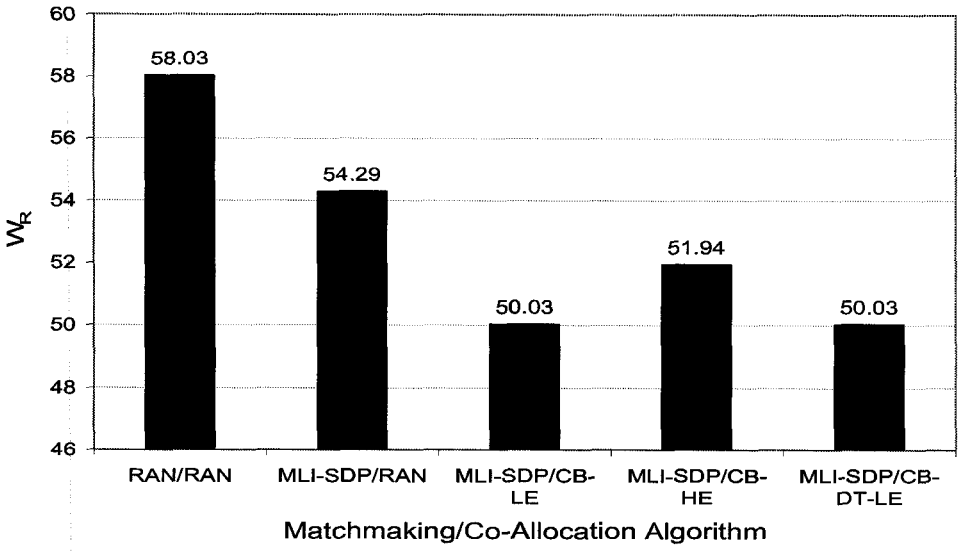


Figure 9.4: Work Rejected for Case MR.2

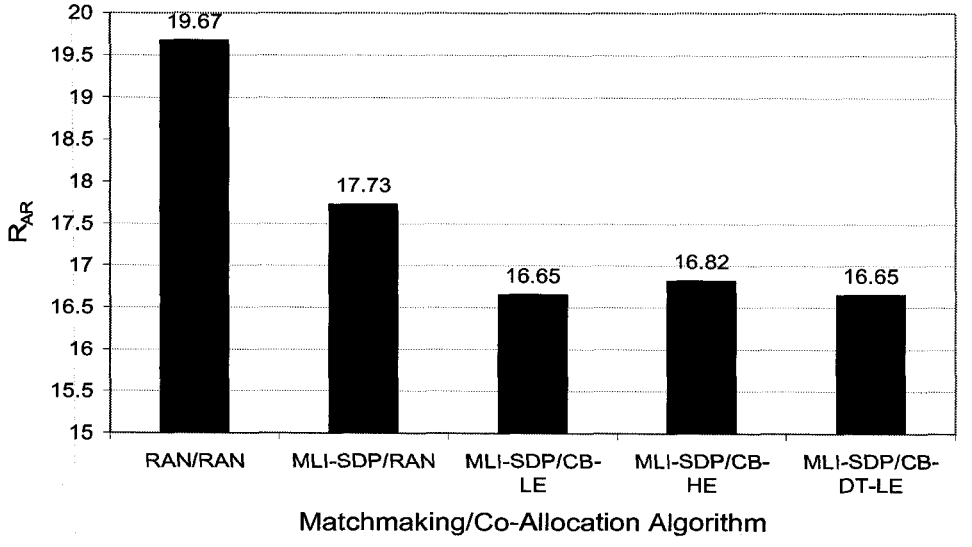


Figure 9.5: Response Time of Advance Reservations for Case MR.2

Figure 9.5 shows that cost based co-allocation algorithms result in the lowest R_{AR} values. Comparison of Figure 9.5 and Figure 9.2 shows that R_{AR} obtained in this case for any given matchmaking/co-allocation algorithm is only a fraction of that obtained in Case MR.1. This is because of two reasons. First, more work is rejected in Case MR.2 and hence there is less contention for the resources. The second reason is that more co-allocation requests are rejected in Case MR.2. The analysis of the simulation logs reveals

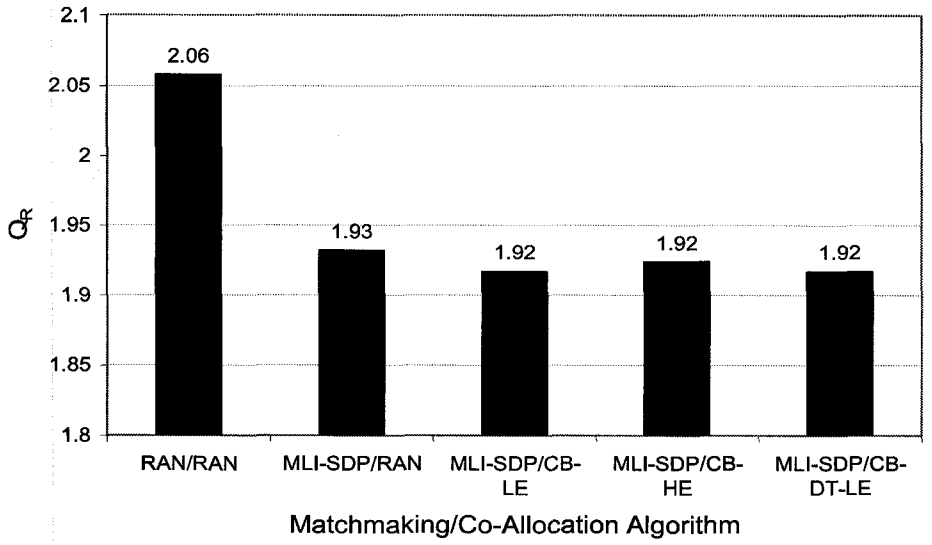


Figure 9.6: Fairness for Case MR.2

that a high proportion of co-allocation requests is rejected in Case MR.2 as they do not have any laxity. Since the execution times of co-allocation requests are decided by the lowest speed resources allocated to them, they result in significantly higher response times compared to non-co-allocation requests where jobs running on high speed resources have very small execution times.

Q_R obtained in Figure 9.6 is similar to that obtained in Figure 9.3.

Results for Case MR.3 and Case MR.4 in Table 9.1 are presented in Appendix E.1 and Appendix E.2, respectively. They display trends similar to those displayed by Case MR.1 and Case MR.2 respectively, and yield the same conclusions.

9.4.3. Impact of Homogeneous Environments on Co-Allocation

This section studies the impact of co-allocation algorithms on system performance for homogeneous environments with H equal to 1. The results in this section correspond to Case MR.5 in Table 9.1. Note that for homogeneous environments, MLI-SDP reduces to MLI and CB-LE and CB-HE become identical.

Figure 9.7 shows that for homogeneous environments MLI-SDP/CB-DT-LE results in the lowest W_R with a value which is only 74.5% of W_R obtained with RAN/RAN. W_R values obtained with MLI-SDP/CB-LE and MLI-SDP/CB-HE are approximately equal the W_R values obtained with MLI-SDP/CB-DT-LE. Since MLI-SDP/CB-DT-LE incurs more overheads, MLI-SDP/CB-LE seems to be the best choice for homogeneous environments as well. Comparison of Figure 9.7 and Figure 9.1 shows

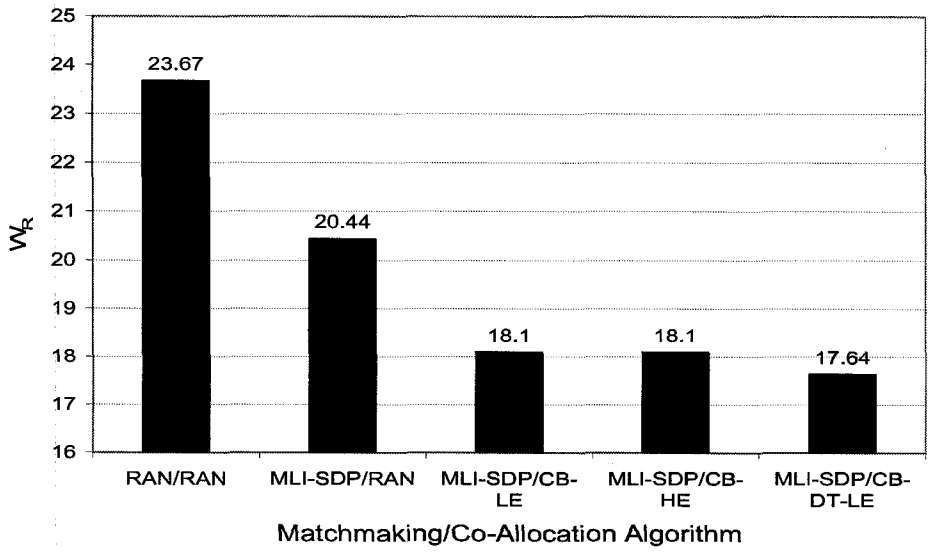


Figure 9.7: Work Rejected for Case MR.5

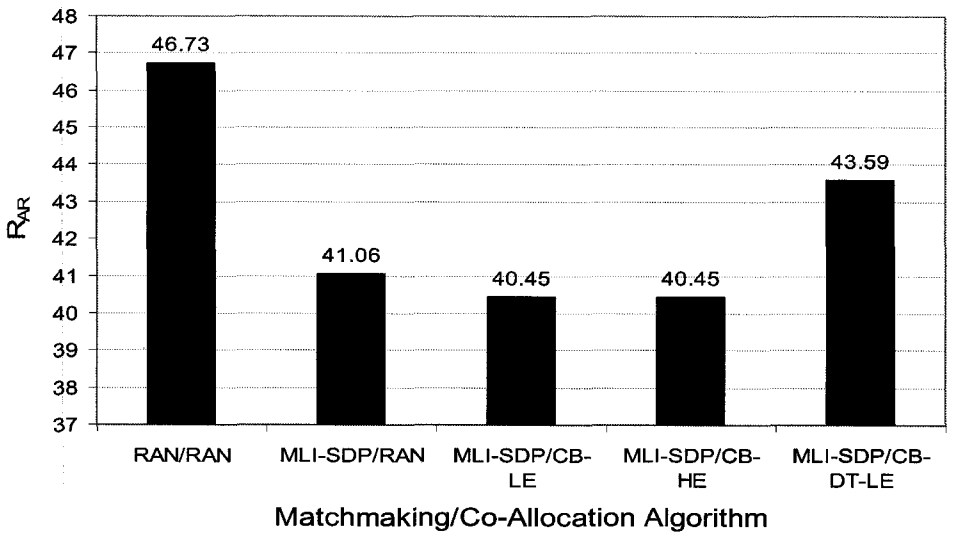


Figure 9.8: Response Time of Advance Reservations for Case MR.5

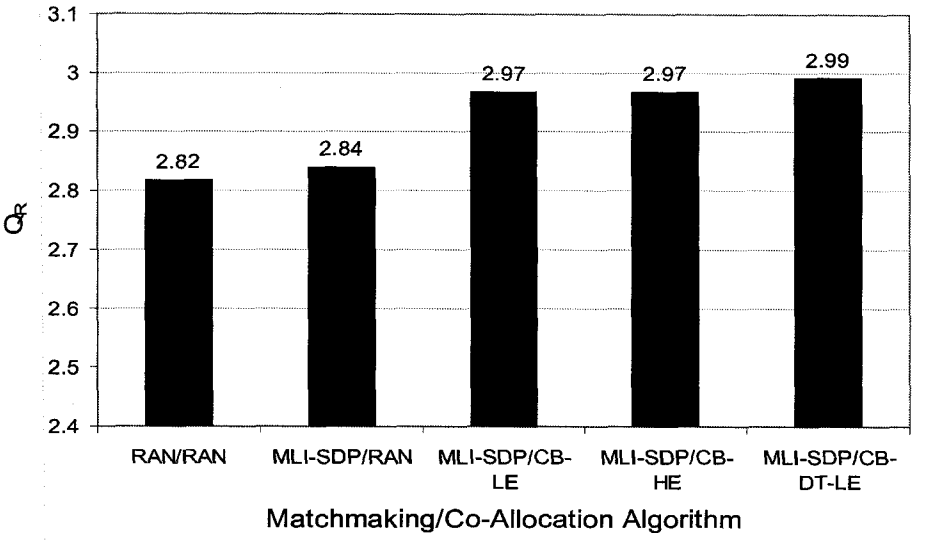


Figure 9.9: Fairness for Case MR.5

that significantly less work is rejected in homogeneous environments. This is because in heterogeneous environments, resource power is wasted on high speed resources when they are co-allocated with low speed resources (as discussed in Section 9.3.3). Hence in Figure 9.7 W_R is only 50% to 57% of that in Figure 9.1 depending on the co-allocation algorithm used.

Figure 9.8 shows that MLI-SDP/CB-LE and MLI-SDP/CB-HE result in the lowest R_{AR} values. This is because for a homogenous environment, primary and secondary costs in CB-LE and CB-HE are equal (as discussed in Section 9.3.3). The resource selection is thus done based on the tertiary cost that is calculated by measuring the impact of laxity of a job on the resource. Results in Chapter 8 has shown that by selecting the resource on which the impact of laxity is minimum, we get the lowest R_{AR} compared to other algorithms.

Figure 9.9 shows that for all algorithms Q_R is significantly larger than 1. This shows that more large jobs are rejected by the system as they are difficult to accommodate in a deadline constrained environment.

9.5. Summary

Simultaneous allocation of multiple resources in multiple domains is a challenging task but it is necessary to meet the demand patterns and QoS constraints of some of the Grid applications. This chapter presents a three phase co-allocation approach, which with the aid of novel co-scheduling algorithms outperforms traditional techniques for co-allocation. Performance analysis of co-allocation algorithms brings important insights into system dynamics.

Results in Section 9.4.2 show that even for a small percentage of co-allocation requests, traditional co-allocation mechanisms result in a substantially higher W_R than that obtained with Multi-Resource Coordinator. The results show that co-allocation algorithms can have a significant impact on system performance. The thesis presents several cost based algorithms and the results show that cost based algorithms result in a much better performance than others as they seek to reduce performance costs by preventing job co-allocation on mismatching resources and by economically using laxity of jobs. Results in Section 9.4.1 and Section 9.4.3 show that CB-LE is the best choice for both heterogeneous and homogeneous environments.

Chapter 10

Conclusions

10.1. Summary

Scalability, flexibility, quality of service provisioning, efficiency and robustness are the desired characteristics of most computing systems. Grids are scalable, encompassing multiple organizations in multiple geographies, and flexible owing to their inherent dynamic nature that allows even runtime integration of heterogeneous resources from multiple domains. The need for QoS provisioning in Grids is well-established [FOS02b]. Many Grid applications require response time guarantees and often guaranteed co-allocation of resources in multiple domains. Efficiency of Grids depends on the effective utilization of Grid resources. High efficiency results in cost-effectiveness and hence justifies the use of Grids. Since QoS objectives of resource consumers in Grids are often in conflict with efficiency goals of resource providers and since Grid resources are shared by multiple applications that belong to different administrative domains, it is challenging to meet the QoS objectives of applications while maintaining high system efficiency. Grid systems must also be robust enough to accommodate uncertainties such as those in user-estimated runtimes while meeting QoS and efficiency goals. This thesis presents a middleware framework for Grids that achieves user satisfaction by providing QoS guarantees for Grid applications, cost effectiveness by efficiently utilizing resources and robustness by intelligently handling uncertain runtimes of applications. To the best of our knowledge, this is the first framework that comprehensively tackles the problem of

QoS provisioning in Grids and allows efficient use of resources even in the presence of uncertainties.

The framework described in the thesis presents components for each of the fabric, resource and collective layer of the Grid layered architecture and aligns well with other Grid technologies. The framework is not limited to any particular type of Grid resources or applications. For example, it can be applied for computational jobs requesting resources on a cluster, a data transfer application requesting lightpaths or jobs that require both computational and network resources. A simulation based investigation of resource management strategies devised for the framework was carried out. The results show that the framework and the strategies are effective in meeting their objectives. Some of the notable features of the system are summarized.

10.1.1. QoS Aware Scheduling on Grid Resources

The thesis presents the Scaling through Subset Scheduling algorithm for an NP-Complete problem of scheduling advance reservation and on-demand requests on non-shared Grid resources. This algorithm supports both preemptable and non-preemptable jobs. Performance results obtained with the algorithm brings important insights into the dynamics of the system. The results demonstrate that under-constraining advance reservations through the introduction of laxity is effective in maintaining high resource utilizations while meeting QoS constraints of applications. The results show that segmentation is effective in achieving high system performance for scenarios where the service times of the jobs have high variability and/or proportion of advance reservations in the workload is high. The results also show that laxity can be exchanged for data segmentation to achieve high utilization.

The thesis presents a novel heuristic algorithm, Grid Scheduling with Deadlines referred to as GSD in the thesis, for an NP-Complete problem of scheduling advance reservations with laxities on a shared resource. GSD can be configured with the help of pluggable strategies to adapt to various workload conditions and needs of the system. The simulation results show that for a wide range of workload parameters the combination of Earliest Deadline First and Best Fit heuristics in GSD demonstrates the best performance. GSD is scalable and supports gang scheduling.

10.1.2. QoS Provisioning Under Uncertain Runtimes

Providing QoS guarantees under uncertain runtimes of the jobs is a challenging task. Simulation results show that error in user-estimated runtimes can significantly affect system performance. The thesis introduced the Schedule Exceptions Manager module of the framework to deal with runtime exceptions and uncertain runtimes and the results show that SEM is highly effective. Results show that for a given percentage error in user-estimated runtimes, overestimation and underestimation can have a significantly different impact on system performance. With the aid of experiments, the thesis shows that for the error handling policies used in the thesis, the runtime-predicting-algorithms should always try to avoid underestimation even if that leads to a higher percentage error. The thesis develops two pre-scheduling policies that have been observed to improve performance under extremely large errors in runtime estimation.

10.1.3. Matchmaking in Multi-Institutional Environments

Effective application-to-resource mapping and load balancing are important to achieve the best possible performance. The results show that traditional matchmaking algorithms, such as Minimum Utilization Algorithm (presented in Section 8.2.5), do not

perform well when applied in multi-institutional settings particularly for workloads consisting of both advance reservations and best-effort jobs. The thesis investigates several matchmaking algorithms and presents a novel algorithm for matchmaking, Minimum Laxity Impact referred to as MLI in the thesis, which outperforms every other algorithm investigated in almost every respect for a wide range of workload parameters. The high performance of MLI can be attributed to the intelligent use of laxity of jobs. Another application of MLI is as an effective meta-scheduler within Platform Computing's Community Scheduler Framework [PLA05c]. The thesis extends matchmaking algorithms to cater to the needs of heterogeneous environments and shows that contrary to intuition no scaling is required in MLI for such environments. The thesis presents Minimum Laxity Impact with Self Dynamic Partitioning algorithm, a variant of MLI, which accommodates a self dynamic partitioning mechanism. The results show that this variant of MLI results in the best performance in heterogeneous environments.

10.1.4. Resource Co-Allocation in Multiple Domains

Simultaneous allocation of multiple resources in multiple domains is a challenging task but it is necessary to meet the demand patterns and QoS constraints of some of the Grid applications. To the best of our knowledge, no previous work on co-allocation on Grids has considered QoS guarantees. The thesis presents a three phase co-allocation approach that is observed to be effective in scheduling requests that require resources from multiple domains. The thesis also presents a number of cost based algorithms for co-allocation that have been observed to result in a much better performance than the other algorithms investigated in the thesis.

10.1.5. Dynamic Configurability

Different components of the framework are configurable; these components can also be plugged in and plugged out in accordance with the needs of the system. The thesis argues that this configurability of the modern systems, which may comprise thousands of resources and consumers, is extremely important for achieving high efficiency.

10.2. Future Work

A number of future research directions resulting from this research are identified.

- The thesis shows that dynamic matchmaking based on system state can significantly improve overall system utilization. Hence, the cost associated with running a particular application on a Grid depends on the overall system state. Based on this observation, it seems likely that a dynamic cost model for resources in an economy-based Grid can help improve overall system utilization and hence reduce costs associated with using Grid resources. Sophisticated dynamic cost-based strategies that can stabilize economy-based Grids by satisfying the well-known John Nash's equilibrium need further investigation.
- The thesis has discussed how network resources, in dynamic optical networks for example, can be scheduled just like other Grid resources to meet the QoS demands of applications. Combining the routing phase in networks with the scheduling and matchmaking phase to optimize network resource usage is an interesting direction for future research.
- The thesis presents the Grid Scheduling with Deadlines algorithm for scheduling advance reservations and on-demand requests on homogeneous shared Grid resources. An extension of Grid Scheduling with Deadlines algorithm for heterogeneous shared Grid resources is an important direction for future research.

- The thesis shows how components of the framework such as the Grid Scheduling with Deadlines scheduler and Schedule Exceptions Manger can be integrated with Globus Toolkit and how the integration can be deployed on a compute cluster. Implementing other components of the framework on a real system and investigation of their performance are proposed as future work.
- Results show cost based algorithms can effectively schedule co-allocation requests while seeking to minimize performance related costs. However, the cost based algorithms presented in the thesis assume that each job of the co-allocation request has the same execution time. Adapting the cost based algorithm presented in this thesis for scenarios where different jobs of a co-allocation request have different resource demands warrants further investigation.
- On some Grid systems, a high proportion of advance reservations may not be desirable. This is because for such systems even a moderate increase in the response times of on-demand jobs may not be tolerable. Techniques for capacity planning on such systems, with either limiting the proportion of advance reservations and/or appropriately partitioning the resources between on-demand and advance reservation requests, form an interesting direction for future research.

References:

[ALL01a] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, "Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing," in the *Proceedings of the 18th IEEE Symposium on Mass Storage Systems*, pp. 13-28, San Diego, CA, USA, April 2001.

[ALL01b] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, B. Toonen, "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus," in the *Proceedings of the Supercomputing Conference*, pp. 52-76, Denver, CO, USA, November 2001.

[ALT04] Altair Grid Technologies, *Portable Batch System.* http://www.openpbs.org. Accessed 2004.

[APG06] ApGrid. http://www.apgrid.org/. Accessed 2006.

[BUY00] R. Buyya, D. Abramson, J. Giddy, "Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid," in the *Proceedings of the 4th International Conference and Exhibition on High Performance Computing in Asia-Pacific Region*, pp. 283-289, Beijing, China, May 2000.

[BUY02] R. Buyya, D. Abramson, J. Giddy, H. Stockinger, "Economic Models for Resource Management and Scheduling in Grid Computing," in *Journal of Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing Environments*, 14(13-15): 1507-1542, 2002.

[CAM06] S. Camorlinga, B. Schofield, "Modeling of Workflow-Engaged Networks on Radiology Transfers Across a Metro Network," in *IEEE Transactions on Information Technology in Biomedicine*, 10(2): 275-281, April 2006.

[CAS00] H. Casanova, G. Obertelli, F. Berman, R. Wolski, "The Apples Parameter Sweep Template: User-Level Middleware for the Grid," in the *Proceedings of the AMC International Conference on Supercomputing*, pp. 60-78, Santa Fe, NM, USA, May 2000.

[CZA98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, 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*, pp. 62-82, Orlando, FL, USA, March 1998.

[CZA04] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, W. Vambenepe, "The WS-Resource Framework," March 2004. http://www.globus.org/wsrf/specs/ws-wsrf.pdf

[CZY96] J. Czyzyk, M. P. Mesnier, J. J. More, "The Network-Enabled Optimization System (NEOS) Server," *Preprint MCS-P615-0996*, Argonne National Laboratory, Argonne, Illinois, 1996.

[DAN03] S. Dandamudi, *Hierarchical Scheduling in Parallel and Cluster Systems*, Kluwer Academic Press, 2003.

[DAT05] DataGRID. http://www.cern.ch/grid/. Accessed 2005.

[DIN01] P. Dinda, "Online Prediction of Runtime of Tasks," in the *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, CA, USA, August 2001.

[DOE05] DOE Science Grid. http://doesciencegrid.org/. Accessed 2005.

[DON06] Y. Dong, J. Yang, Z. Wu, "ODSG: An Architecture of Ontology-based Distributed Simulation on Grid," in the *Proceedings of the International Multi-Symposiums on Computer and Computational Sciences*, pp. 759-765, Hangzhou, China, June 2006.

[DOW97] A. Downey, "Predicting Queue Times on Space-Sharing Parallel Computers," in the *Proceedings of the 11th International Parallel Processing Symposium*, pp. 209-218, Geneva, Switzerland, April 1997.

[DUT06] Dutch Grid. http://hepwww.rl.ac.uk/htasc/jun00/bos/index.htm. Accessed 2006.

[ENT06] Entropia Inc., *Entropia PC-Grid Computing*, http://www.entropia.com/. Accessed 2006.

[EUR05] EUROGrid. http://www.eurogrid.org/. Accessed 2005.

[FAR05a] U. Farooq, S. Majumdar, E. W. Parsons, "Efficiently Scheduling Advance Reservations in Grids," *Technical Report SCE-05-14*, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, August 2005.

[FAR05b] U. Farooq, S. Majumdar, E. W. Parsons, "Impact of Laxity on Scheduling with Advance Reservations in Grids", in the *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems,* pp. 319-324, Atlanta, GA, USA, September 2005.

[FAR05c] U. Farooq, S. Majumdar, E. W. Parsons, "Dynamic Scheduling of Lightpaths in Lambda Grids", in the *Proceedings of the 2nd IEEE International Workshop on Networks for Grid Applications*, pp. 540-549, Boston, MA, USA, October 2005.

[FAR06a] U. Farooq, S. Majumdar, E. W. Parsons, "Scheduling Advance Reservations on Shared Grid Resources", *Technical Report SCE-06-13*, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, July 2006.

[FAR06b] U. Farooq, S. Majumdar, E. W. Parsons, "A Framework to Achieve Guaranteed QoS for Applications and High System Performance in Multi-Institutional Grid Computing", in the *Proceedings of the 35th International Conference on Parallel Processing*, pp. 373-380, Columbus, OH, USA, August 2006.

[FAR06c] U. Farooq, O. M. Kanwar, S. Majumdar, E.W. Parsons, "Building Advance Reservations Enabled Grid Computing Systems," Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, September 2006.

[FAR07] U. Farooq, S. Majumdar, E. W. Parsons, "Engineering Grids Applications and Middleware for High Performance", in the *Proceedings of the 6th ACM International Workshop on Software and Performance,* pp. 141-152, Buenos Aires, Argentina, February 5-8, 2007.

[FARira] U. Farooq, S. Majumdar, E. W. Parsons, "Providing QoS Guarantees under Uncertain Runtimes of Jobs in Multi-Institutional Grid Computing," *Journal Paper in review.*

[FARirb] U. Farooq, S. Majumdar, E. W. Parsons, "QoS Aware Matchmaking in Dynamic Heterogeneous Environments," *Journal Paper in preparation.*

[FARirc] U. Farooq, S. Majumdar, E.W. Parsons, "Achieving Efficiency, Quality of Service and Robustness in Multi-Organizational Grids," *Journal Paper in review.*

[FIG03] R. Figueiredo, P. Dinda, J. Fortes, "A Case for Grid Computing on Virtual Machines," in the *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pp. 550-559, Providence, RI, USA, May 2003.

[FIG04a] 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 Applications,* San Jose, CA, USA, October 2004.

[FIG04b] 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*, pp. 707-714, Chicago, IL, USA, April 2004.

[FOH95] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems," in the *Proceeding of the 16th IEEE Real-Time Systems Symposium*, pp. 152-161, Pisa, Italy, December 1995.

[FOS99a] I. Foster, C. Kesselman, "Computational Grids," in *The Grid: Blueprint for a New Computing Infrastructure*, pp. 15-51, 1999.

[FOS99b] 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*, pp. 27-36, London, United Kingdom, May 1999.

[FOS00] 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*, pp. 181-188, Pittsburgh, PA, USA, June 2000.

[FOS01] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," in *International Journal of Supercomputer Applications and High Performance Computing*, 15(3): 200-222, 2001.

[FOS02a] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," *Technical Document*, Open Grid Service Infrastructure Working Group, Global Grid Forum, June 2002.

[FOS02b] I. Foster, "What is the Grid? A Three Point Checklist," (published in *GRID Today*, July 2002). http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf. Accessed 2004.

[FOS03] 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.

[FOS05] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, J. Von Reich, "The Open Grid Services Architecture, Version 1.0," *Informational Document*, Global Grid Forum, January 2005.

[FUN01] S. Funk, J. Goossens, S. Baruah, "On-Line Scheduling on Uniform Multiprocessors," in the *Proceedings of the 22nd IEEE Real-Time Systems Symposium,* pp. 183-192, London, United Kingdom, December 2001.

[GAR79] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman Press, 1979.

[GHO04] P. Ghosh, N. Roy, S. K. Das, K. Basu, "A Game Theory Based Pricing Strategy for Job Allocation in Mobile Grids," in the *Proceedings of the 18th International Parallel & Distributed Processing Symposium,* pp. 82-91, Santa Fe, NM, USA, April 2004.

[GIA01] F. Giacomini, F. Prelz, "Definition of Architecture, Technical Plan and Evaluation Criteria for Scheduling, Resource Management, Security and Job Description," *Technical Report DataGrid-01-D1.4-0127-1_0*, European DataGrid Project, 2001.

[GLO05a] Global Grid Forum. http://www.ggf.org/. Accessed 2005.

[GLO05b] Globus Alliance. http://www.globus.org/alliance/. Accessed 2005.

[GLO05c] The Globus Toolkit. http://www.globus.org/toolkit/. Accessed 2005.

[GLO05d] Globus Alliance, *GT Information Services: Monitoring & Discovery System.* http://www.globus.org/toolkit/mds/. Accessed 2005.

[GLO05e] Global Grid Forum Performance Working Group, *A Grid Monitoring Architecture.* http://www-didc.lbl.gov/GGF-PERF/GMA-WG/. Accessed 2005.

[GLO05f] Global Grid Forum CIM-based Grid Schema Working Group. http://www.daasi.de/wgs/CIMGIS/. Accessed 2005.

[GLO05g] Global Grid Forum Discovery and Monitoring Event Data Working Group. http://www-didc.lbl.gov/damed/. Accessed 2005.

[GLO05h] Globus Alliance, *GT 4.0 Security.* http://www.globus.org/toolkit/docs/4.0/security/. Accessed 2005.

[GLO05i] Global Grid Forum Distributed Resource Management Application API Working Group. http://www.drmaa.org/. Accessed 2005.

[GLO05j] Global Grid Forum Scheduling Dictionary Working Group, http://www.fz-juelich. de/ zam/RD/coop/ggf/sd-wg.html. Accessed 2005.

[GLO05k] Global Grid Forum Grid Resource Allocation Agreement Protocol Working Group, http://www.fz-juelich.de/zam/RD/coop/ggf/graap/graap-wg.html. Accessed 2005.

[GLO05l] Globus Alliance, *GridFTP*. http://www.globus.org/grid_software/data/gridftp.php. Accessed 2005.

[GLO05m] Globus Alliance, *GT 4.0 WS-GRAM Approach*. http://www.globus.org/toolkit/docs/4.0/execution/key/WS_GRAM_Approach.html. Accessed 2005.

[GLU05] GLUE Information Model, http://infnforge.cnaf.infn.it/glueinfomodel/. Accessed 2005.

[GNU04] Gnutella Inc., *Gnutella File Sharing System*. http://www.gnutella.com/. Accessed 2004.

[GRI06] Grid-Ireland. http://www.grid.ie/. Accessed 2006.

[HIN03] V. Hingne, A. Joshi, T. Finin, H. Kargupta, E. Houstis, "Towards A Pervasive Grid", in the *Proceedings of the 17th IEEE International Parallel and Distributed Processing Symposium*, pp. 207-214, Nice, France, April 2003.

[HOB05] P. R Hobson, "GRIDCC - providing a real-time Grid for distributed instrumentation," White Paper, 2005. http://www.gridcc.org/

[JOS03] J. Joseph, C. Fellenstein, *Grid Computing*, Prentice Hall Press, December 2003.

[KAP07] N. K. Kapoor, S. Majumdar, B. Nandy, "Matching of Independent Jobs on a Computing Grid," in the *Proceedings of the 2007 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pp. 537-546, San Diego, CA, USA, July 2007.

[KAR05a] Karlsruhe Research Centre, Institute for Data Processing and Electronics, *Ultrasound Computer Tomography*, http://www.ipe.fzk.de/projekt/med/usct/d_index.html. Accessed 2005.

[KAR05b] Karlsruhe Research Centre, Institute for Data Processing and Electronics, *Pierre Auger Observatory, http://www.ipe.fzk.de/projekt/auger/e_index.html.* Accessed 2005.

[KAR05c] Karlsruhe Research Centre, Institute for Data Processing and Electronics, *KArlsruhe TRItium Neutrino experiment. http://www.ipe.fzk.de/fachgruppe software/schwerpunkte/katrin/d_index.html.* Accessed 2005.

[KIM04] K. H. Kim, "Wide-Area Real-Time Distributed Computing in a Tightly Managed Optical Grid – An Optiputer Vision," in the *Proceedings of the 18th International Conference on Advanced Information Networking and Applications,* pp. 2-11, Fukuoka, Japan, March 2004.

[KOS04] T. Kosar, 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*, pp. 342-349, Tokyo, Japan, March 2004.

[KUR03] K. Kurowski, J. Nabrzyski, A. Oleksiak. J. Weglarz, "Multicriteria Aspects of Grid Resource Management," in *Grid Resource Management, State of the Art and Future Trends,* pp. 271-293, 2003.

[LAVir] 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.*

[LEE96] C. Lee, C. Kesselman, S. Schwab, "Near-Realtime Satellite Image Processing: Meta-Computing in CC++," in *IEEE Computer Graphics and Applications*, 16(4):79-84, 1996.

[LIT98] M. J. Litzkow, M. Livny, M. W. Mutka, "Condor: A Hunter of Idle Workstations," in the *Proceedings of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, Amsterdam, The Netherlands, June 1988.

[LIU73] C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," in *Journal of the ACM*, 20(1): 46-61, January 1973.

[LUB03] U. Lublin, D. G. Feitelson, "The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs," in *Journal of Parallel and Distributed Computing*, 63(11): 1105-1122, November 2003.

[MAJ07] S. Majumdar, U. Farooq, E.W. Parsons, "Schedulability Analysis and Performance Bounds for Advance Reservations Requests on Grids," Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, January 2007.

[MAM03] 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.

[MAN98] G. Manimaran, C. S. R. Murthy, "A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis," in *IEEE Transactions on Parallel and Distributed Systems*, 9(11): 1137-1152. November 1998.

[MAR96] 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*, pp. 49-54, Renvale, Ireland, September 1996.

[MAU04] The Maui Scheduling System. http://www.mhpcc.edu/maui. Accessed 2004.

[MCM75] G. McMahon, M. Florian, "On Scheduling with Ready Times and Due Dates To Minimize Maximum Lateness," in *Operations Research*, 23(3): 475-482, May-June, 1975.

[MEC93] C. Mechoso, C. Ma, J. Farrara, J. Spahr, R. Moore, "Parallelization and Distribution of a Coupled Atmosphere-Ocean General Circulation Model," in *Monthly Weather Review*, 121:2062-2076, 1993.

[MIK03] M. Mika, G. Waligora, J. Weglarz, "A Metaheuristic Approach to Scheduling Workflow Jobs on a Grid," in *Grid Resource Management, State of the Art and Future Trends,* pp. 295-318, 2003.

[MUA01] A. W. Mualem, D.G. Feitelson, "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling," in *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529-543, June 2001.

[NIE96] 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*, pp. 432-441, Syracuse, NY, USA, August 1996.

[NOR96] 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 *International Journal of Supercomputer Applications and High Performance Computing*, 10(2):131-140, 1996.

[NOR04] Nortel Networks, *Dynamic Resource Allocation Controller*. http://www.nortel.com. Accessed 2004.

[PHA02] T. Phan, L. Huang, C. Dulan, "Challenge: Integrating Mobile Wireless Devices into the Computational Grid," in the *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pp. 271-278, Atlanta, GA, USA, September 2002.

[PLA05a] Platform Computing Inc., *Load Sharing Facility*. http://www.platform.com. Accessed 2005.

[PLA05b] Platform Computing Inc., *Platform Multi-Cluster*. http://www.platform.com. Accessed 2005.

[PLA05c] Platform Computing Inc., "Open Source Meta-Scheduling for Virtual Organizations with the Community Scheduler Framework (CSF)," *Technical Whitepaper*. http://www.platform.com. Accessed 2005.

[POT96] C. Potter, R. Brady, P. Moran, C. Gregory, B. Carragher, N. Kisseberth, J. Lyding, J. Lindquist, "EVAC: A Virtual Environment for Control of Remote Imaging Instrumentation," in *IEEE Computer Graphics and Applications*, 16(4):62-66, 1996.

[RAM90] K. Ramamritham, J.A. Stankovic, P.F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," in *IEEE Transactions on Parallel and Distributed Systems*, 1(2): 184-194, April 1990.

[RAM98] R. Raman, M. Livny, M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," in the *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pp. 140-146, Chicago, IL, USA, July 1998.

[REA06] RealityGrid. http://www.realitygrid.org/. Accessed 2006.

[ROU97] M. Roussos, A. Johnson, J. Leigh, C. Valsilakis, C. Barnes, T. Moher, "NICE: Combining Constructionism, Narrative, and Collaboration in a Virtual Learning Environment," in *Computer Graphics*, 31(3):62-63, 1997.

[SAW03] A.A. Sawchuk, E. Chew, R. Zimmermann, C. Papadopoulos, C. Kyriakakis, "From Remote Media Immersion to Distributed Immersive Performance," in the *Proceedings of the ACM SIGMM 2003 Workshop on Experiential Telepresence*, Berkeley, CA, USA, November 2003.

[SCH02] J. M. Schopf, B. Nitzberg, "Grids: The Top Ten Questions," in *Scientific Programming, Special Issue on Grid computing*, 10(2):103-111, August 2002.

[SCH03] J. M. Schopf, L. Yang, "Using Predicted Variance for Conservative Scheduling on Shared Resoruces," in *Grid Resource Management, State of the Art and Future Trends,* pp. 215-236, 2003.

[SCH71] L. Schrage, "Obtaining Optimal Solutions to Resource Constrained Network Scheduling Problems," *Unpublished Manuscript*, March 1971.

[SCH98] O. Schelen, S. Pink, "Resource Sharing in Advance Reservation Agents," in *Journal of High Speed Networks, Special Issue on Multimedia Networking*, 7(3-4), 1998.

[SEC04] Secure File Transfer. http://acs.ucsd.edu/info/scp.php. Accessed 2004.

[SHA04] Sharman Networks, *Kazaa*, http://www.kazaa.com/. Accessed 2004.

[SHI02] Y. Shi, A. Shortridge, J. Bartholic, "Grid Computing for Real Time Distributed Collaborative Geo-Processing," in the *Proceedings of the Symposium on Geo-Spatial Theory, Processing and Applications*, Ottawa, ON, Canada, July 2002.

[SIL99] F. A. B. da Silva, I. D. Scherson, "Improvements in Parallel Job Scheduling Using Gang Service," in the *Proceedings of the 1999 International Symposium on Parallel Architectures*, pp. 268-273, Perth, Australia, June 1999.

[SMI00] W. Smith, I. Foster, V. Taylor, "Scheduling with Advanced Reservations," in the *Proceedings of the IEEE/ACM 14th International Parallel and Distributed Processing Symposium*, pp. 127-132, Cancun, Mexico, May 2000.

[SMI02] W. Smith, P. Wong, "Resource Selection Using Execution and Queue Wait Time Predictions," *Technical Report NAS02-003*, NASA Ames Research Center, 2002.

[SMI03] W. Smith, "Improving Resource Selection and Scheduling Using Predictions," in *Grid Resource Management, State of the Art and Future Trends,* pp. 237-253, 2003.

[SOB04] G. Sobin, G. Kochhar, P. Sadayappan, "Job Fairness in Non-Preemptive Job Scheduling," in the *Proceedings of the 33rd International Conference on Parallel Processing*, pp. 186-194, Montreal, QC, Canada, August 2004.

[SUD06] Sudo Utility, http://www.gratisoft.us/sudo/. Accessed 2006.

[SUL04] 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*, Boston, MA, USA, November 2004.

[SUN03] X. H. Sun and M. Wu, "Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling," in the *Proceedings of the 17th International Parallel & Distributed Processing Symposium,* Nice, France, April 2003.

[SUN05] Sun Microsystems Inc., *The Sun Grid Engine*, http://www.sun.com/software/gridware/index.xml. Accessed 2005.

[TAY01] V. Taylor, X. Wu, J. Geisler, X. Li, Z. Lan, M. Hereld, I. Judson, R. Stevens, "Prophesy: Automating the Modeling Process," in the *Proceedings of the 3rd International Workshop on Active Middleware Services*, pp. 3-11, San Francisco, CA, USA, August 2001.

[TER04] TeraGrid. http://www.teragrid.org/. Accessed 2004.

[UNI04] Unicore Forum. http://www.unicore.org/. Accessed 2004.

[W3C05a] W3C, *Web Services Description Language*. http://www.w3.org/TR/wsdl. Accessed 2005.

[W3C05b] W3C, *Web Services Addressing*. http://www.w3.org/2002/ws/addr/. Accessed 2005.

[WEI04] L. Weigang, M. V. P. Dib, D. A. Cardoso, "Grid Service Agents for Real Time Traffic Synchronization," in the *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence,* pp. 619-623, Beijing, China, September 2004.

[WHE96] G. H. Wheless, C. M. Lascara, A. Valle-Levinson, D. P. Brutzman, W. Sherman, W. L. Hibbard, B. E. Paul, "Virtual Chesapeake Bay: Interacting with a Coupled Physical/Biological Model," in *IEEE Computer Graphics and Applications*, 16(4):42-43, 1996.

[WIK05] Wikipedia, *Normal Distribution*, http://en.wikipedia.org/wiki/Normal_ distribution. Accessed 2005.

[WOL03] R. Wolski, L. J. Miller, G. Obertelli, M. Swany, "Performance Information Services for Computational Grids," in *Grid Resource Management, State of the Art and Future Trends,* pp. 193-213, 2003.

[XU90] J. Xu, D. Parnas, "Scheduling Processes With Release Times, Deadlines, Precedence and Exclusion Relations," in I*EEE Transactions on Software Engineering*, 16(3):360-369, 1990.

[ZHA01] Y. Zhang, A. Sivasubramaniam, "Scheduling Best-effort and Real-time Pipelined Applications on Time-Shared Clusters," in the *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures,* pp. 209-219, Crete Island, Greece, July 2001.

Appendix A

A.1. Preventing Large Scheduling Times with SSS

As discussed in Section 5.2, SSS prevents large scheduling times by suitably selecting and running the algorithm for only a subset of jobs S. Although unlikely in the Grid domain, it might happen that the size of S becomes very large. Since Problem 1 (defined in Section 3.3.2.1) is an NP-complete problem, such situations might result in extremely large completion time of the algorithm. To prevent such situations, the thesis defines a threshold size þ of the subset S, such that if the size of S is greater than þ SSS allows only a certain number of solution-nodes $\hat{H}$ to be included in vector v. If SSS cannot find a feasible solution within those nodes $\hat{H}$, the new job is rejected. The threshold size þ and number of nodes $\hat{H}$ can be chosen to suit the needs of the system. They can depend on a number of factors such as completion time of each iteration on the resource, average runtimes of the jobs and inter-arrival times of the jobs.

A.2. Results for Preemptable Jobs with Uniformly Distributed Service Times and No Starvation Prevention for On-Demand Requests

This section presents results for Case NS.2 described in Table 5.1. Figure A.1(a) presents the effect of PAR on U while Figure A.1(b) presents the effect of L on U. Effect of PAR on response time of on-demand requests and advance reservation requests for different values of L are presented in Figure A.2 and Figure A.3 respectively.

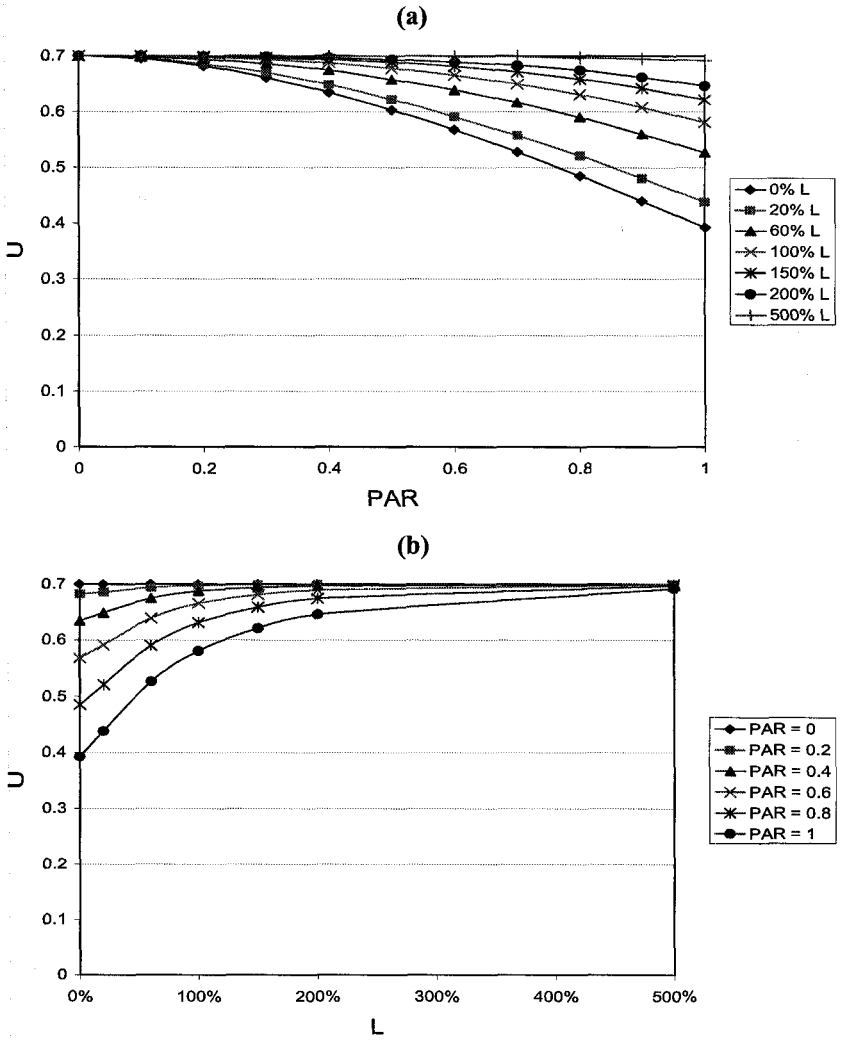


Figure A.1: Utilization for Case NS.2
(a) Effect of PAR on U (b) Effect of L on U

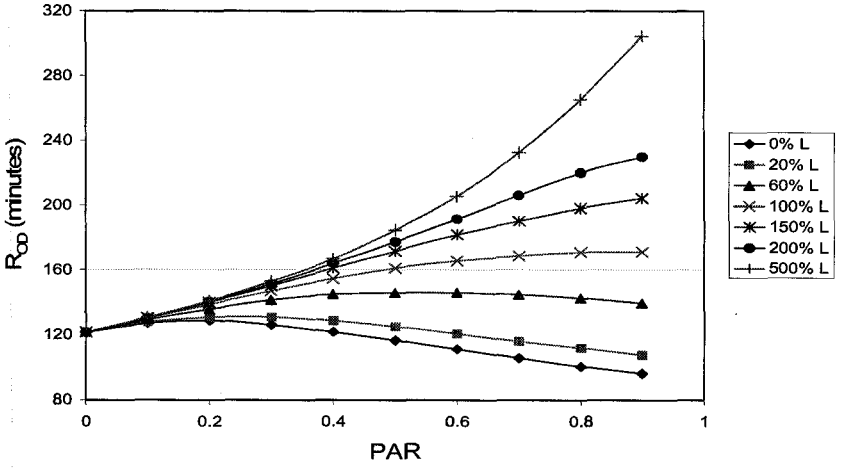


Figure A.2: Effect of PAR on R_{OD} for Case NS.2

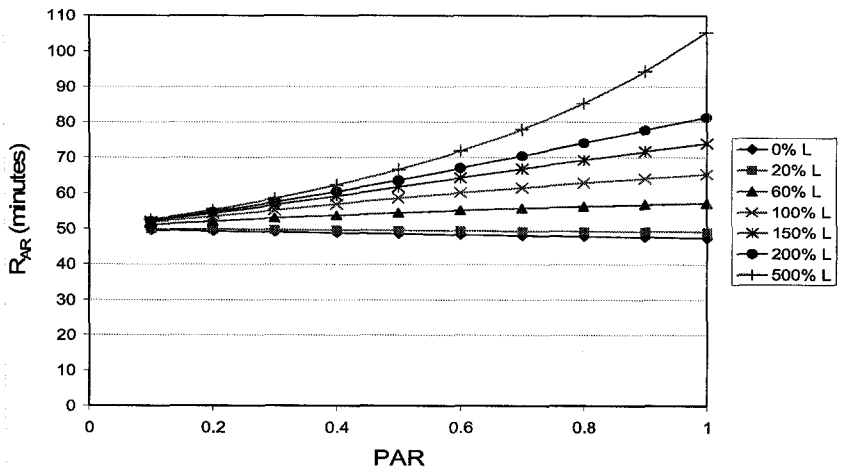


Figure A.3: Effect of PAR on R_{AR} for Case NS.2

A.3. Results for 50% Non-Preemptable and 50% Preemptable Jobs with Uniformly Distributed Service Times and No Starvation Prevention for On-Demand Requests

This section presents results for Case NS.4 in Table 5.1. Figure A.4 presents the effect of PAR on U while Figure A.5 presents the effect of L on U. Effect of PAR on response time of on-demand requests and advance reservation requests for different values of L are presented in Figure A.6 and Figure A.7 respectively.

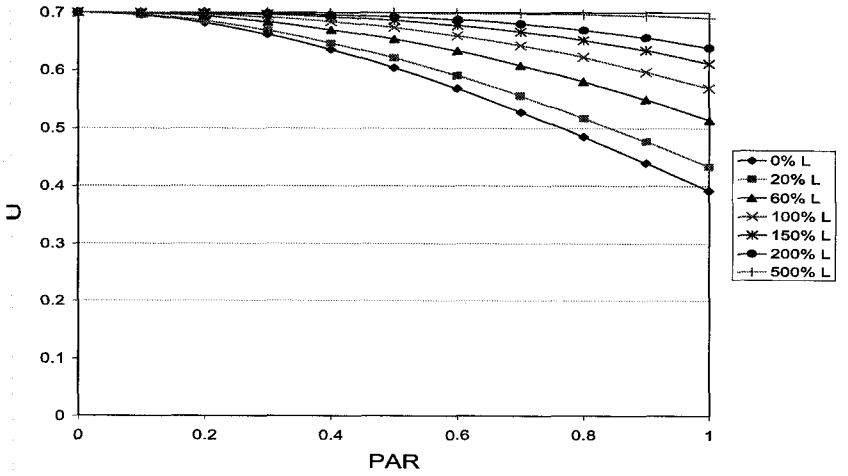


Figure A.4: Effect of PAR on U for Case NS.4

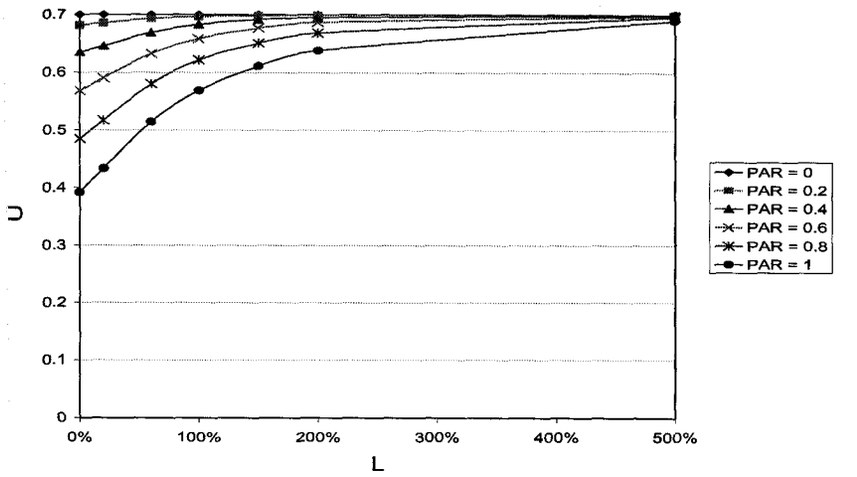


Figure A.5: Effect of L on Utilization for Case NS.4

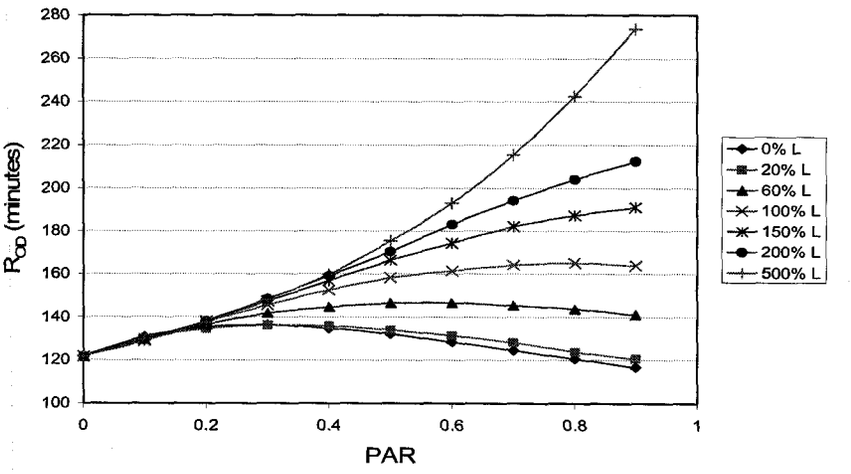


Figure A.6: Effect of PAR on R_{OD} for Case NS.4

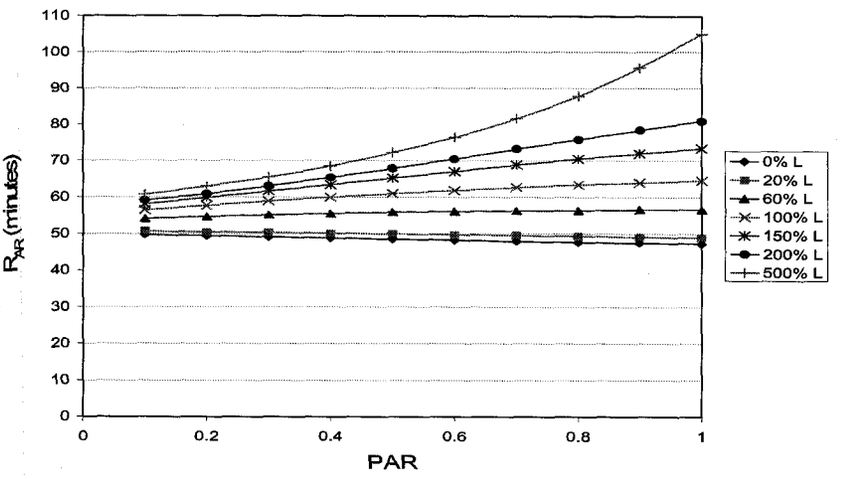


Figure A.7: Effect of PAR on R_{AR} for Case NS.4

A.4. Results for Preemptable Jobs with Hyper-Exponentially Distributed Service Times and No Starvation Prevention for On-Demand Requests

This section presents results for Case NS.6 in Table 5.1. Figure A.8(a) presents the effect of PAR on U while Figure A.8(b) presents the effect of L on U. Effect of PAR on response time of on-demand requests and advance reservation requests for different values of L are presented in Figure A.9 and Figure A.10 respectively.

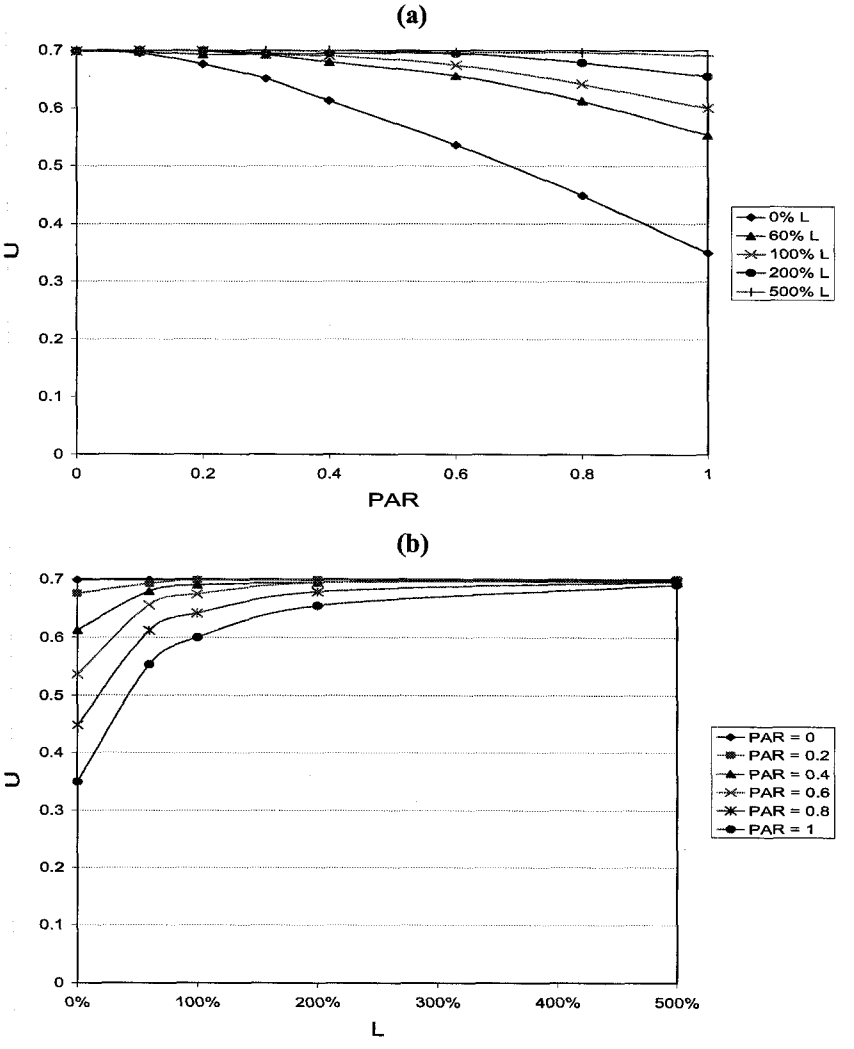


Figure A.8: Utilization for Case NS.6
(a) Effect of PAR on U (b) Effect of L on U

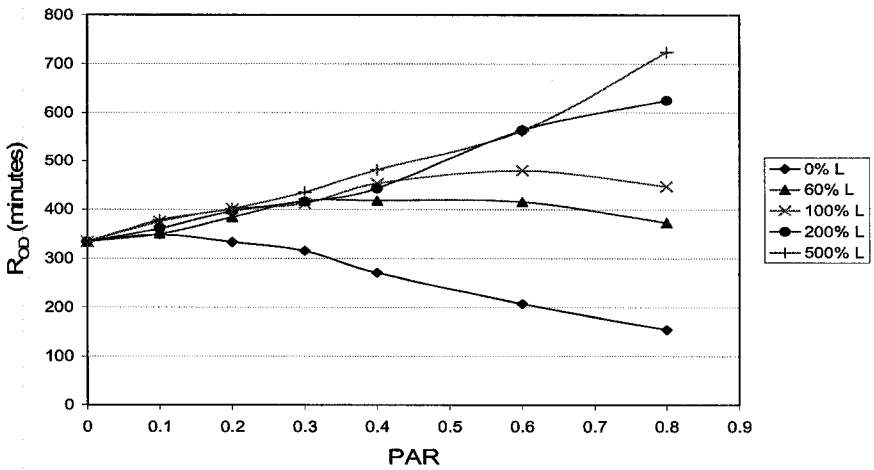


Figure A.9: Effect of PAR on R_{OD} for Case NS.6

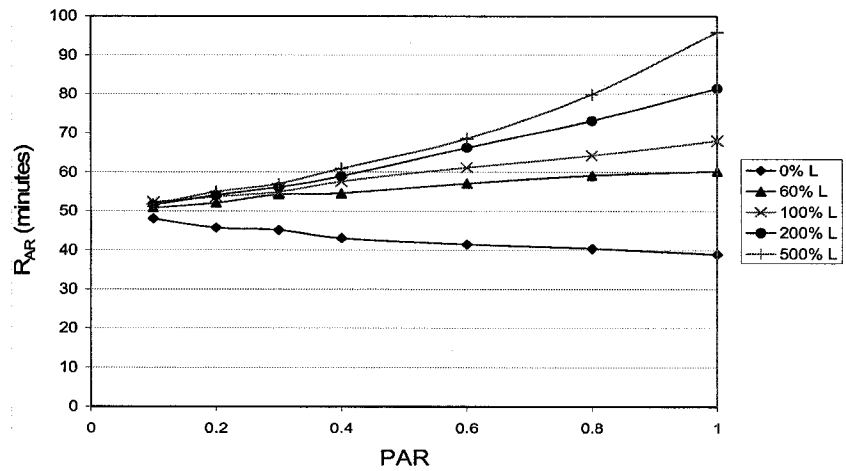


Figure A.10: Effect of PAR on R_{AR} for Case NS.6

A.5 Effect of Preemption at Different Values of PAR

This section presents the impact of preemption on performance at different values of PAR. Figure A.11 and Figure A.12 present the impact of preemption on utilization for PAR = 0.1 with uniform and hyper-exponentially distributed services times respectively. Similarly, Figure A.13 and Figure A.14 present the impact of preemption on R_{OD} for PAR = 0.1 with uniform and hyper-exponentially distributed service times respectively while Figure A.15 and Figure A.16 present the impact of preemption on R_{AR} for PAR = 0.1 with uniform and hyper-exponentially distributed service times respectively.

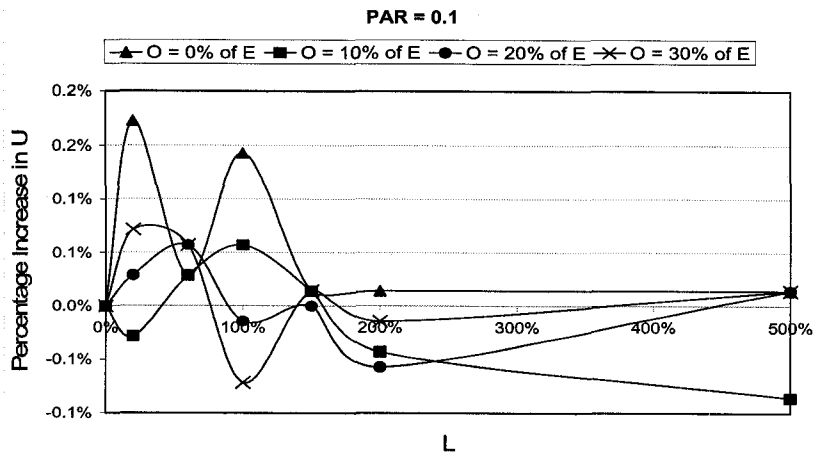


Figure A.11: Effect of Preemption on U for PAR = 0.1 for Uniformly Distributed Service Times

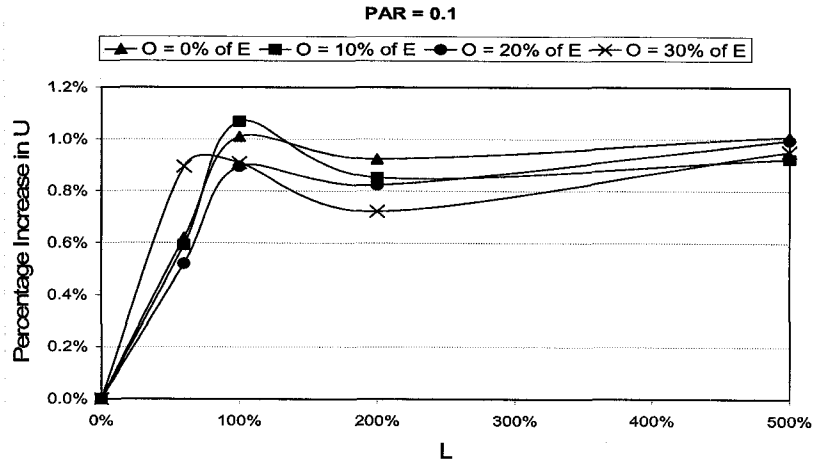


Figure A.12: Effect of Preemption on U for PAR = 0.1 for Hyper-Exponentially Distributed Service Times

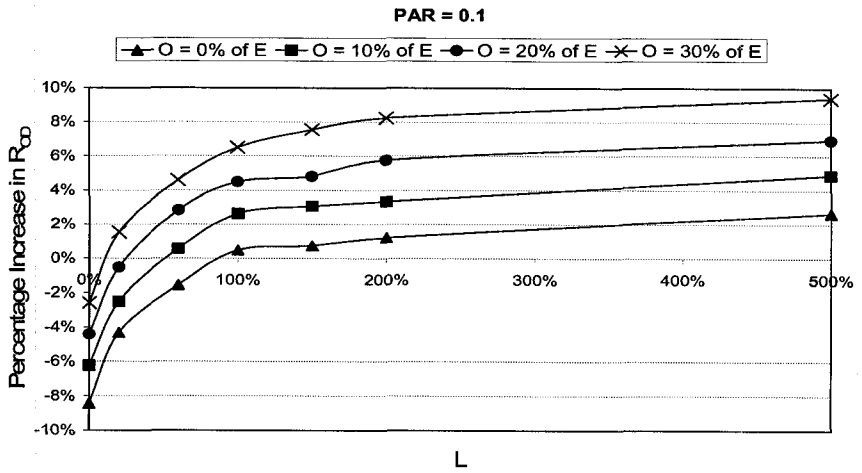


Figure A.13: Effect of Preemption on R_{OD} for PAR = 0.1 for Uniformly Distributed Service Times

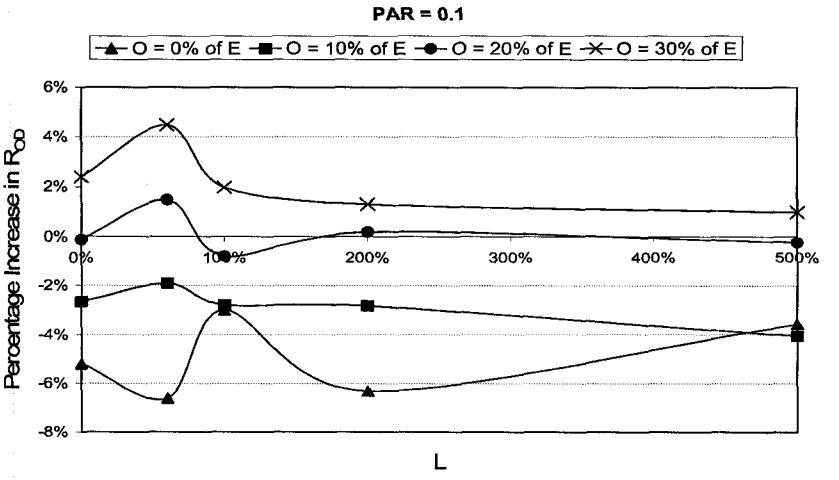


Figure A.14: Effect of Preemption on R_{OD} for PAR = 0.1 for Hyper-Exponentially Distributed Service Times

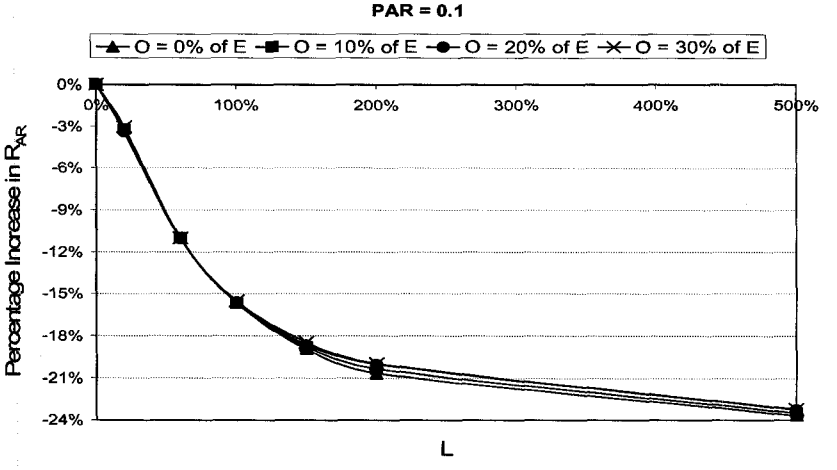


Figure A.15: Effect of Preemption on R_{AR} for PAR = 0.1 for Uniformly Distributed Service Times

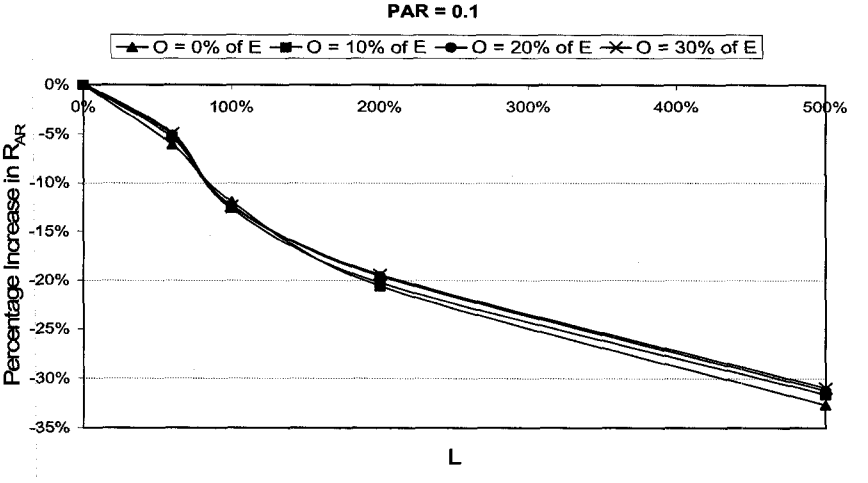


Figure A.16: Effect of Preemption on R_{AR} for PAR = 0.1 for Hyper-Exponentially Distributed Service Times

Figure A.17 and Figure A.18 present the impact of preemption on utilization for PAR = 1.0 with uniform and hyper-exponentially distributed services times respectively. Similarly, Figure A.19 and Figure A.20 present the impact of preemption on R_{AR} for PAR = 1.0 with uniform and hyper-exponentially distributed service times respectively.

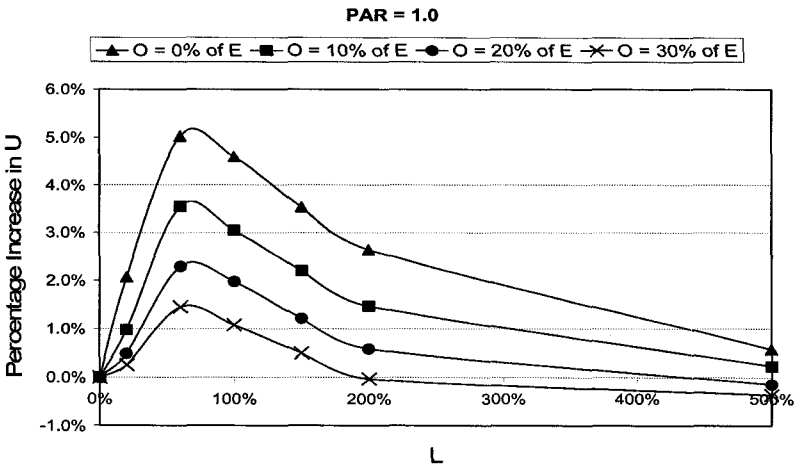


Figure A.17: Effect of Preemption on U for PAR = 1.0 for Uniformly Distributed Service Times

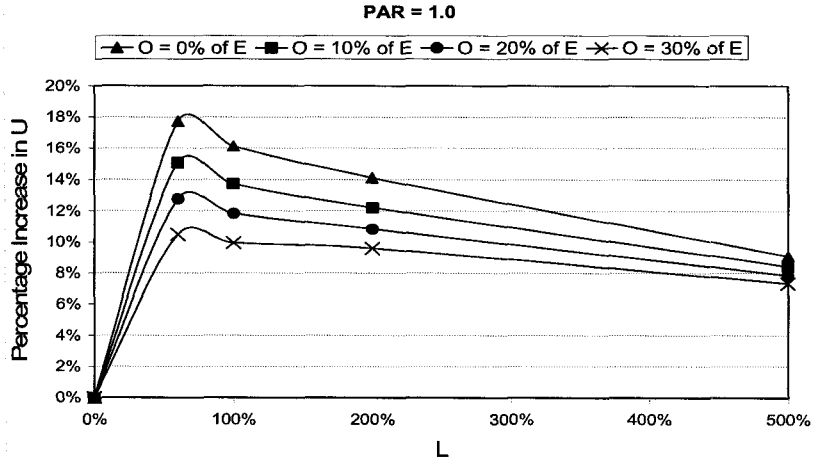


Figure A.18: Effect of Preemption on U for PAR = 1.0 for Hyper-Exponentially Distributed Service Times

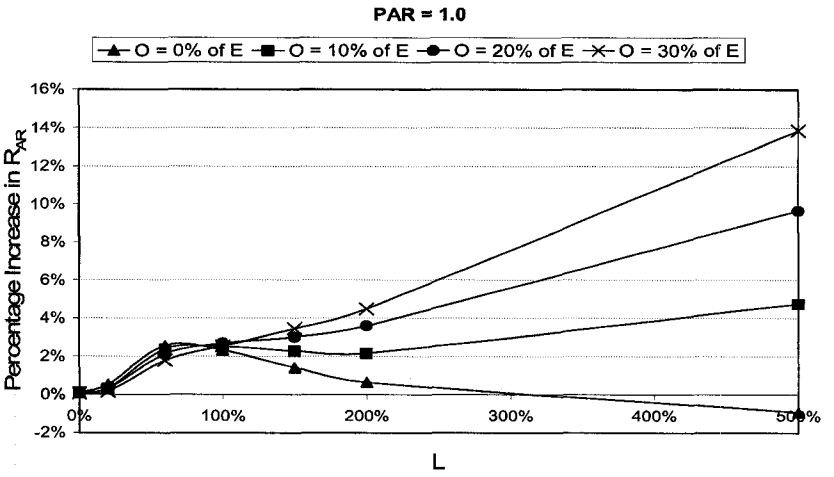


Figure A.19: Effect of Preemption on R_{AR} for PAR = 1.0 for Uniformly Distributed Service Times

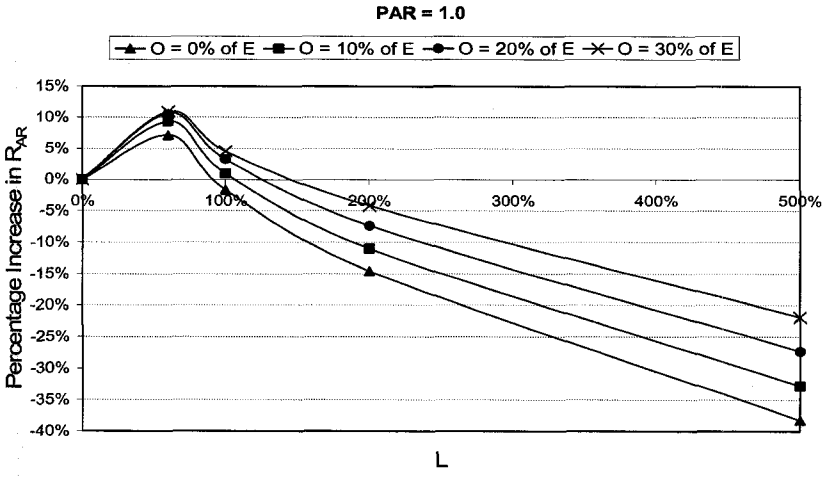
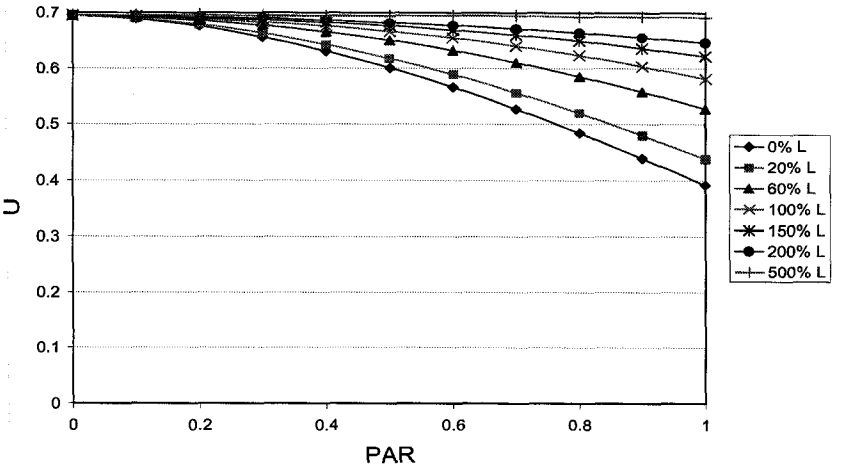


Figure A.20: Effect of Preemption on R_{AR} for PAR = 1.0 for Hyper-Exponentially Distributed Service Times

A.6. Results for Preemptable Jobs with Uniformly Distributed Service Times and Starvation Prevention for On-Demand Requests

This section presents results for Case NS.9 in Table 5.1. Figure A.21(a) presents the effect of PAR on U while Figure A.21(b) presents the percentage decrease in U in the NS.9 case compared to the NS.2 case. Effect of PAR on R_{AR} for different values of L is presented in Figure A.22. Figure A.23(a) presents the effect of PAR on R_{OD} while Figure A.23(b) presents the percentage decrease in R_{OD} in Case.NS.9 compared to Case NS.2.

(a)



(b)

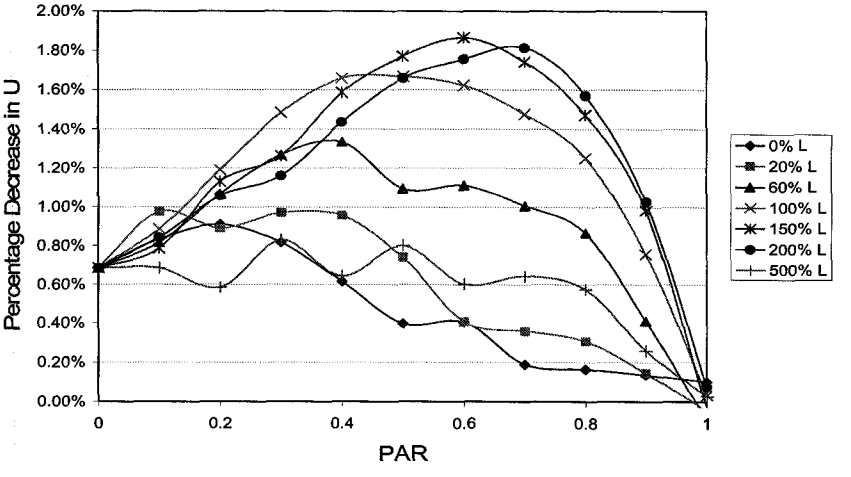


Figure A.21: Effect of Starvation Prevention Manager on U for Preemptable Jobs
(a) Effect of PAR on U for Case NS.9
(b) Percentage Decrease in U in Case NS.9 Compared to Case NS.2

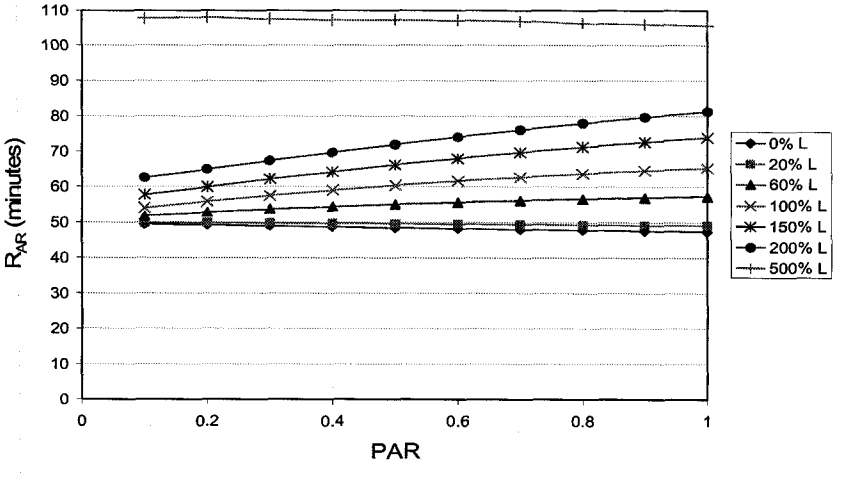
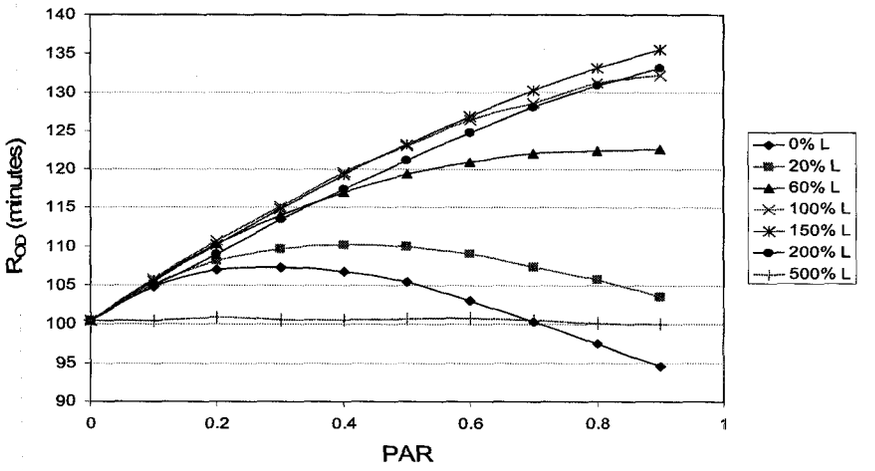


Figure A.22: Effect of PAR on R_{AR} for Case NS.9

(a)



(b)

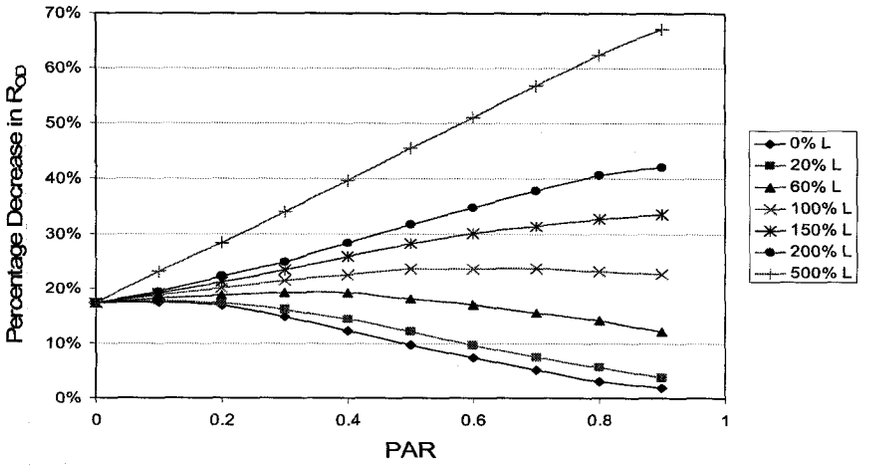


Figure A.23: Effect of Starvation Prevention Manager on R_{OD} for Preemptable Jobs
(a) Effect of PAR on R_{OD} for Case NS.9
(b) Percentage Decrease in R_{OD} in Case NS.9 Compared to Case NS.2

Appendix B

B.1. Scheduling Independent Tasks with GSD

If gang scheduling of all tasks of the jobs is not required, GSD can be adapted to schedule each task of the job independently. Independent scheduling allows for better utilization of the resource by providing more flexibility in task scheduling. However, with independent scheduling different tasks of the jobs may not be executing on the resource at the same time and hence it may lead to higher inter-task communication overheads. Mix of gang and independent scheduling is also possible with GSD.

If all or some of the tasks of a job j_i can be scheduled independent of the other, those tasks are treated as separate jobs by GSD. Those tasks get different values for their required number of nodes depending on their requirements. Thus, a single job j_i in list A in step 1 of GSD becomes multiple jobs corresponding to different independent tasks of job j_i. The rest of the algorithm remains the same. If some tasks of a particular job needs to be gang scheduled while other can be scheduled independently, the job is suitably split into sub-jobs. Each sub-job either represents an independent task or a combination of several tasks that need to be gang scheduled.

Appendix C

C.1. Results for Case SE.3

This section presents results for Case SE.3 in Table 7.1. W_R, U, W_A, UU, R_{AR} and Q_R for this case are shown in Figure C.1, Figure C.2, Figure C.3, Figure C.4, Figure C.5 and Figure C.6, respectively.

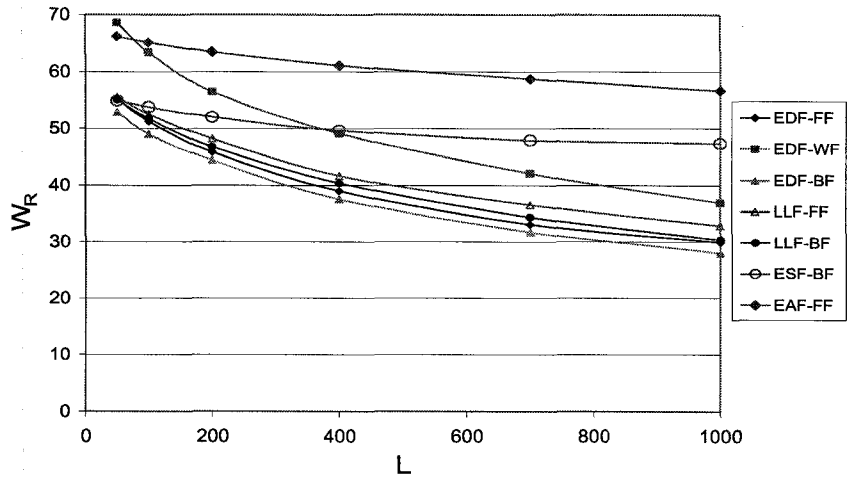


Figure C.1: Work Rejected for Case SE.3

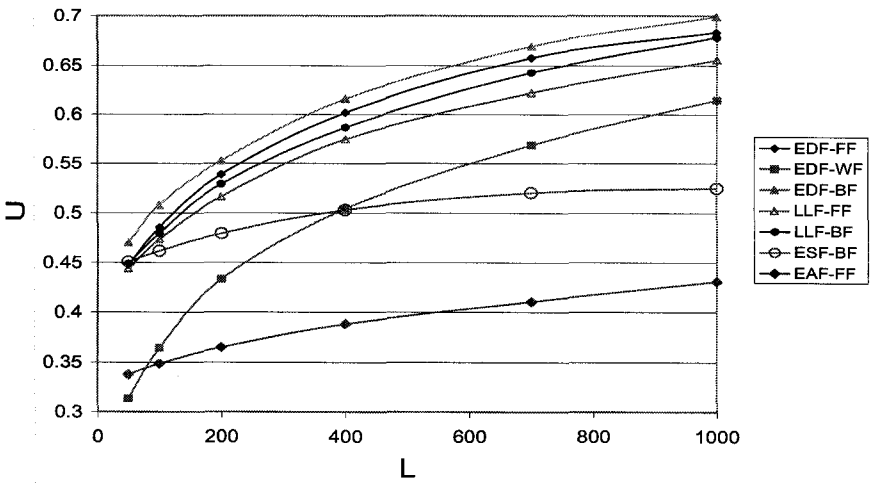


Figure C.2: Utilization for Case SE.3

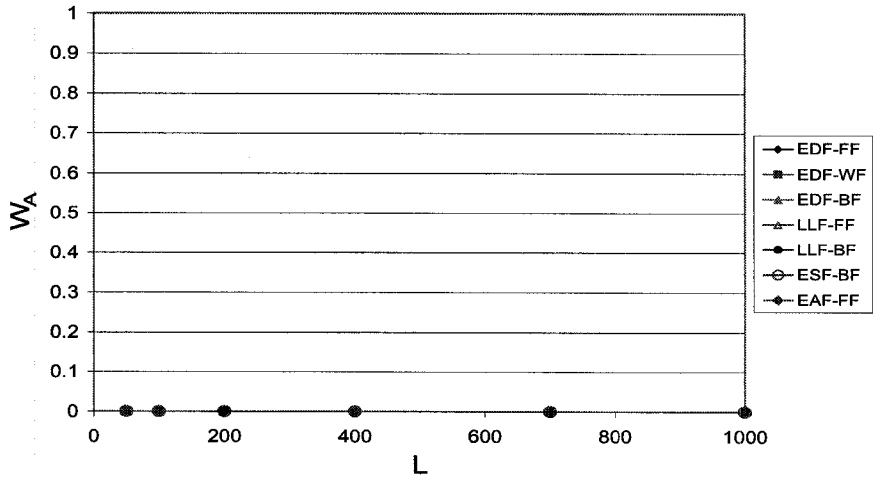


Figure C.3: Work Aborted for Case SE.3

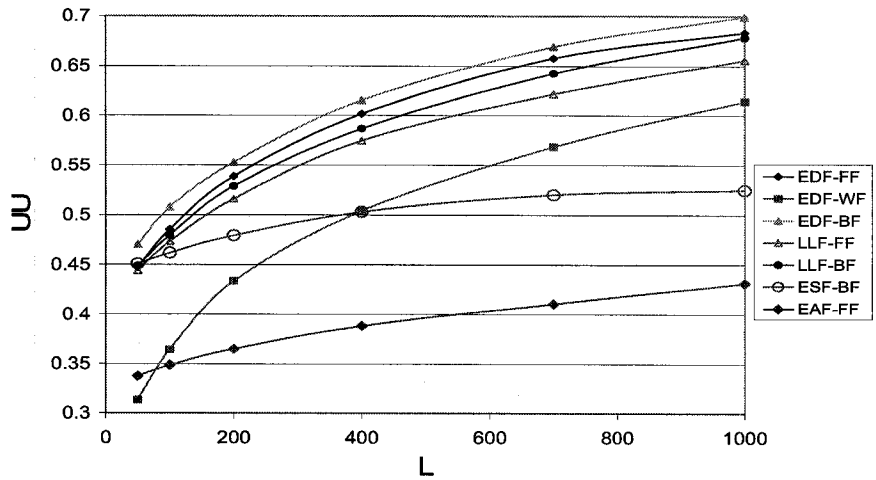


Figure C.4: Useful Utilization for Case SE.3

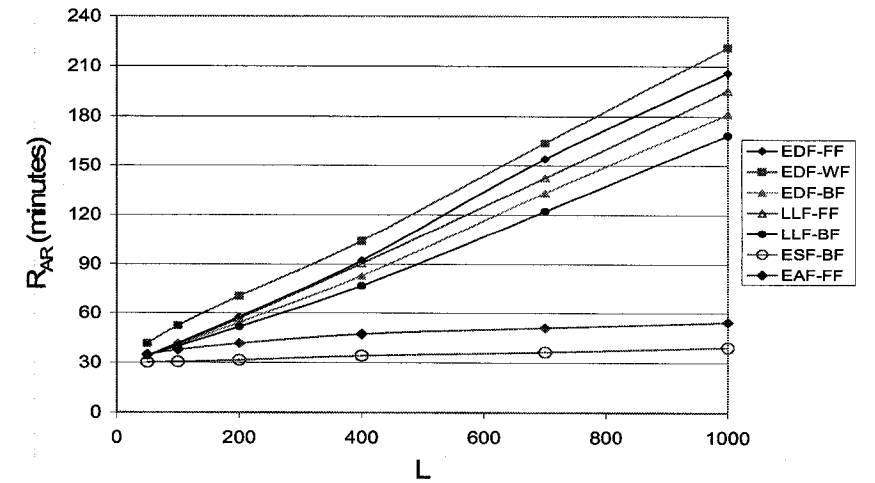


Figure C.5: Response Time of Advance Reservations for Case SE.3

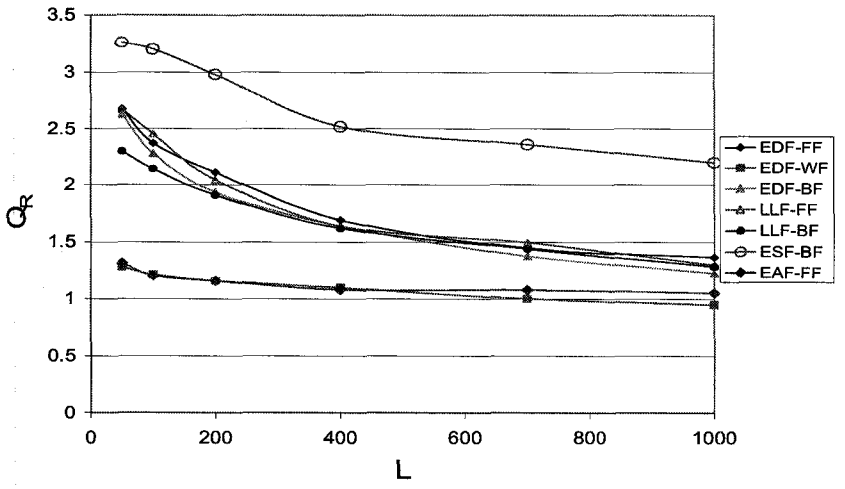


Figure C.6: Fairness for Case SE.3

C.2. Results for Case SE.4

This section presents results for Case SE.4 in Table 7.1. W_R, U, W_A, UU, R_{AR} and Q_R for this case are shown in Figure C.7, Figure C.8, Figure C.9, Figure C.10, Figure C.11 and Figure C.12, respectively.

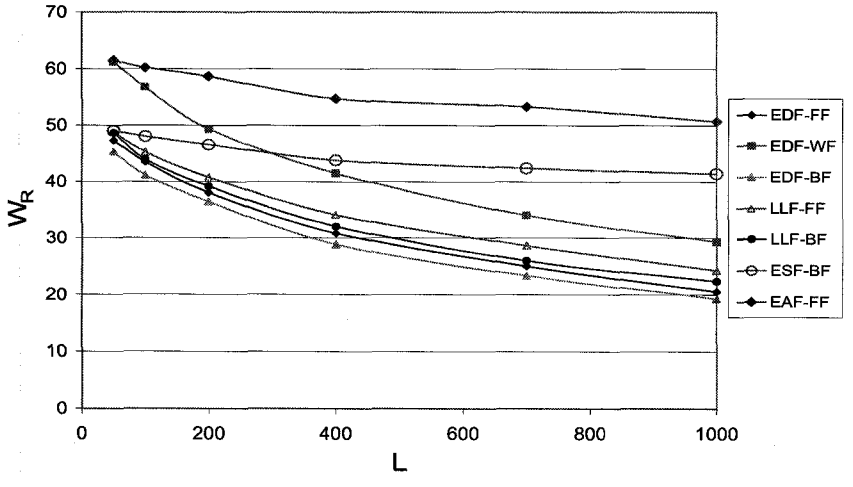


Figure C.7: Work Rejected for Case SE.4

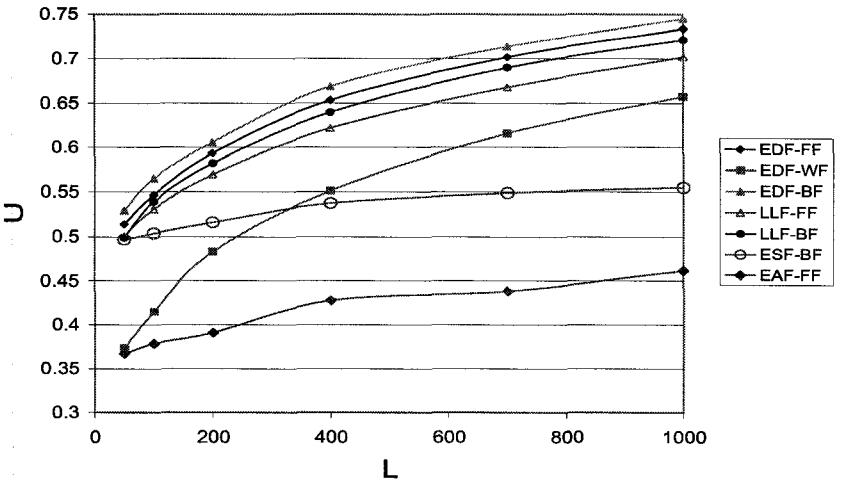


Figure C.8: Utilization for Case SE.4

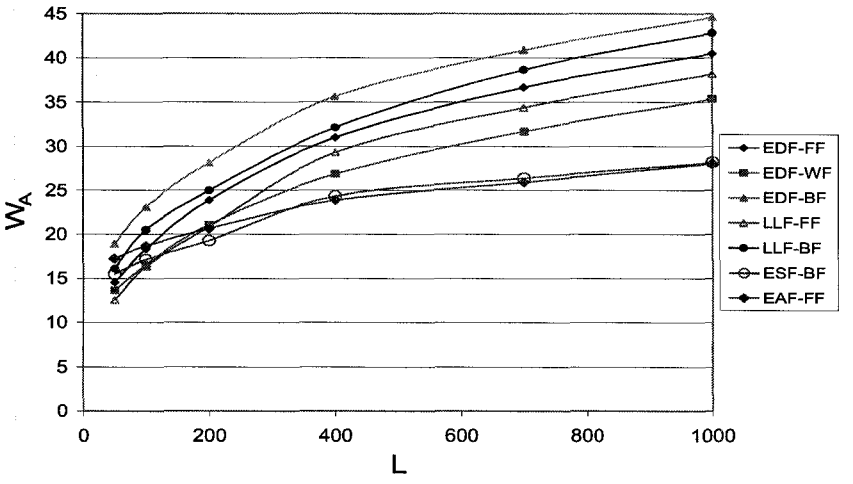


Figure C.9: Work Aborted for Case SE.4

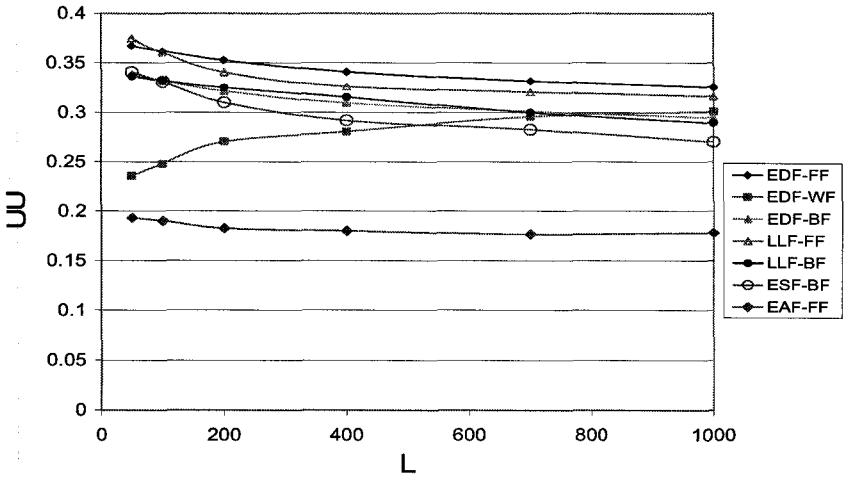


Figure C.10: Useful Utilization for Case SE.4

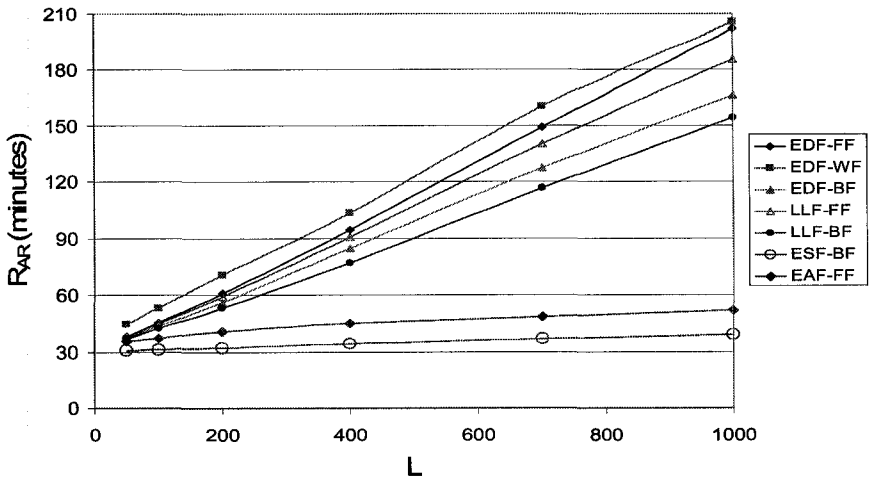


Figure C.11: Response Time of Advance Reservations for Case SE.4

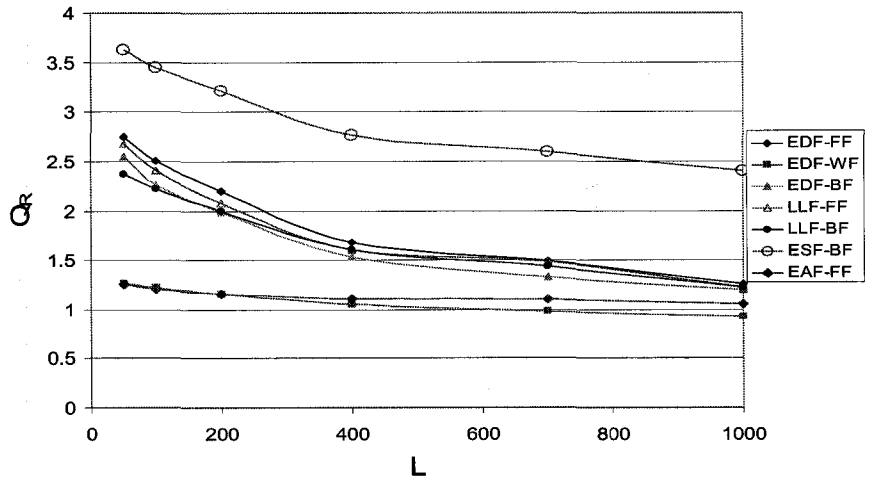


Figure C.12: Fairness for Case SE.4

Appendix D

D.1. Results for Case MM.5

This section presents results for Case MM.5 in Table 8.1. W_R, U, R_{OD}, R_{AR} and Q_R for this case are shown in Figure D.1, Figure D.2, Figure D.3, Figure D.4 and Figure D.5, respectively.

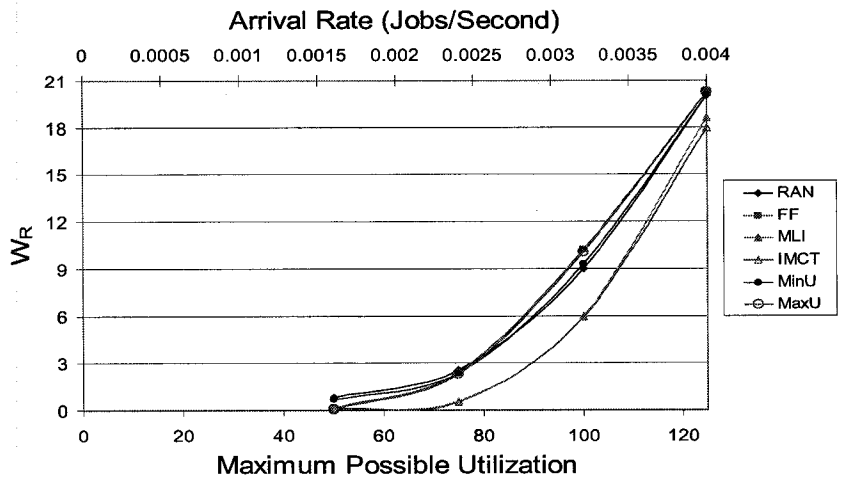


Figure D.1: Work Rejected for Case MM.5

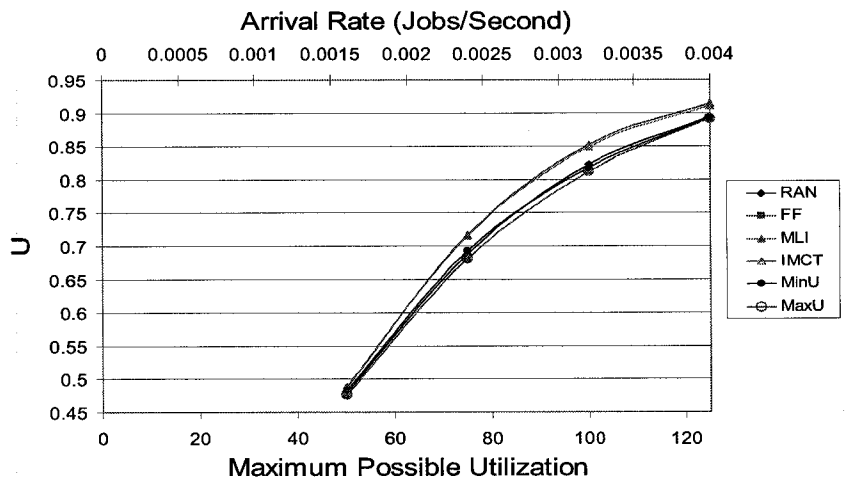


Figure D.2: Utilization for Case MM.5

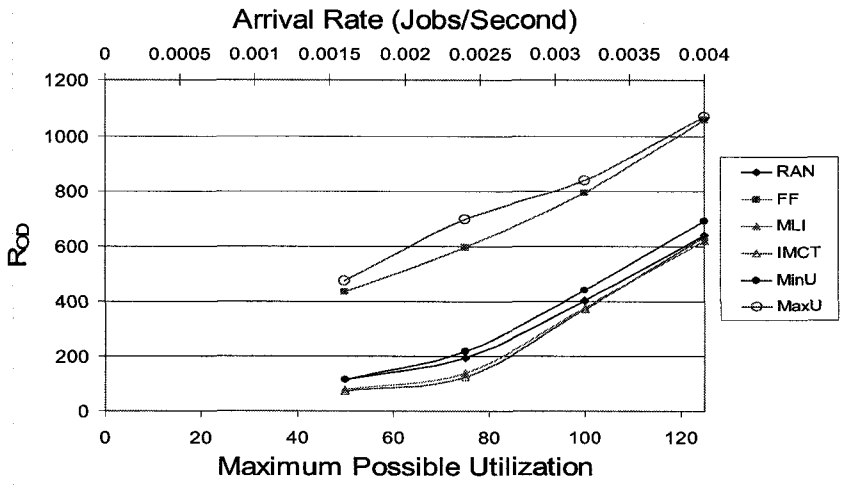


Figure D.3: Response Time of On-Demand Requests for Case MM.5

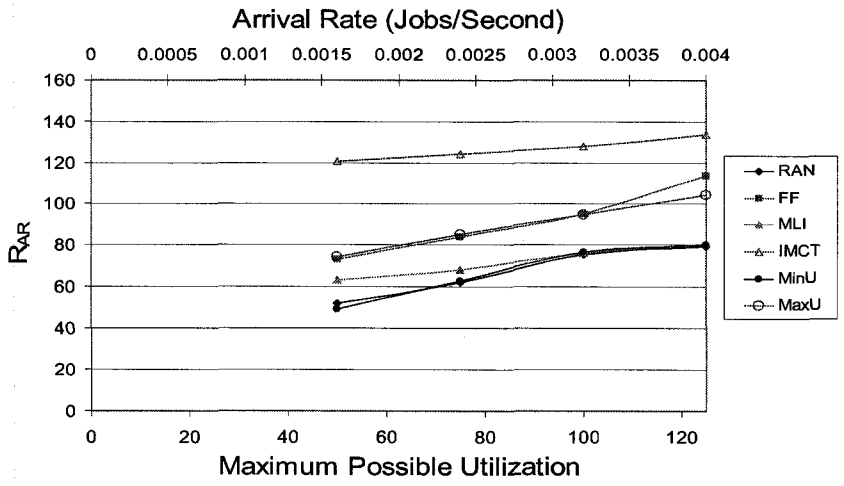


Figure D.4: Response Time of Advance Reservation Requests for Case MM.5

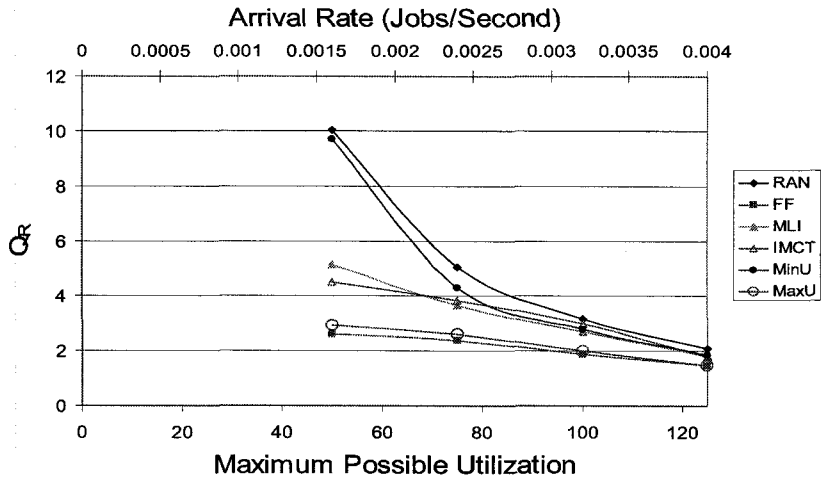


Figure D.5: Fairness for Case MM.5

D.2. Results for Case MM.10 and MM.11

This section presents results for Case MM.10 and Case MM.11 in Table 8.1. Figure D.6 and Figure D.7 show the work rejected for Case MM.10. R_{OD} and R_{AR} for Case MM.10 are shown in Figure D.8 and Figure D.9, respectively. Similarly, work rejected for Case MM.11 is shown in Figure D.10 and Figure D.11 while R_{AR} for this case is shown in Figure D.12.

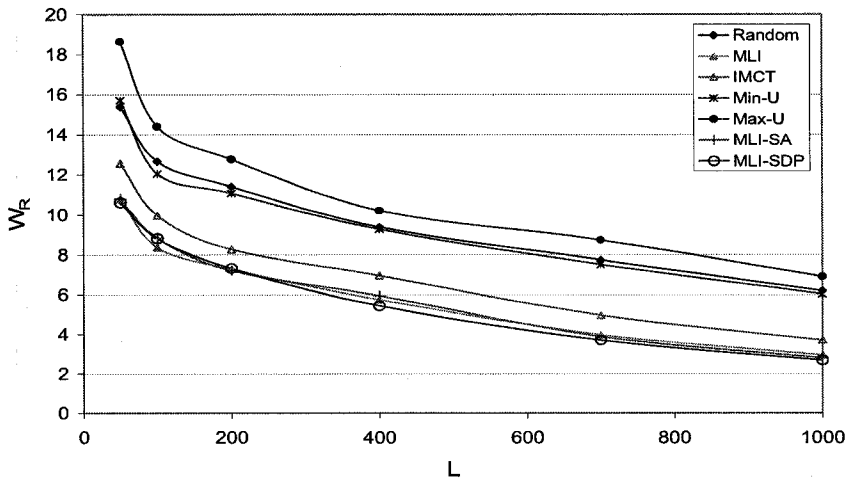


Figure D.6: Work Rejected for Case MM.10

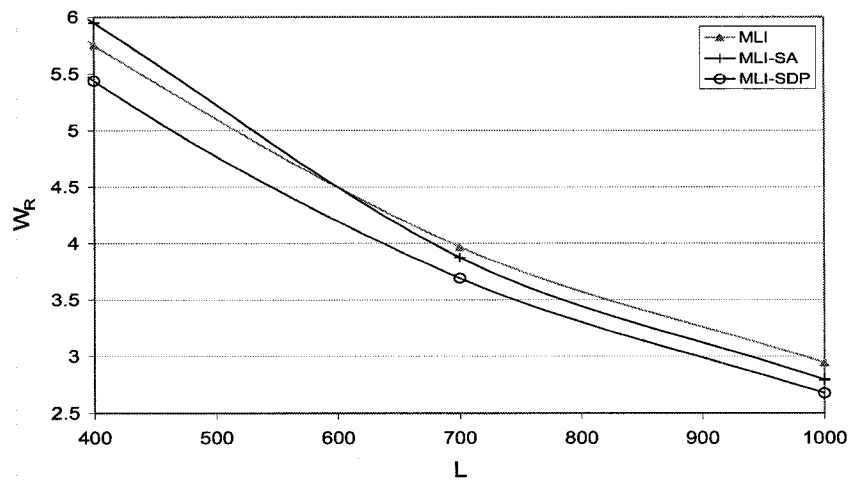


Figure D.7: A Closer Look at Work Rejected for Case MM.10

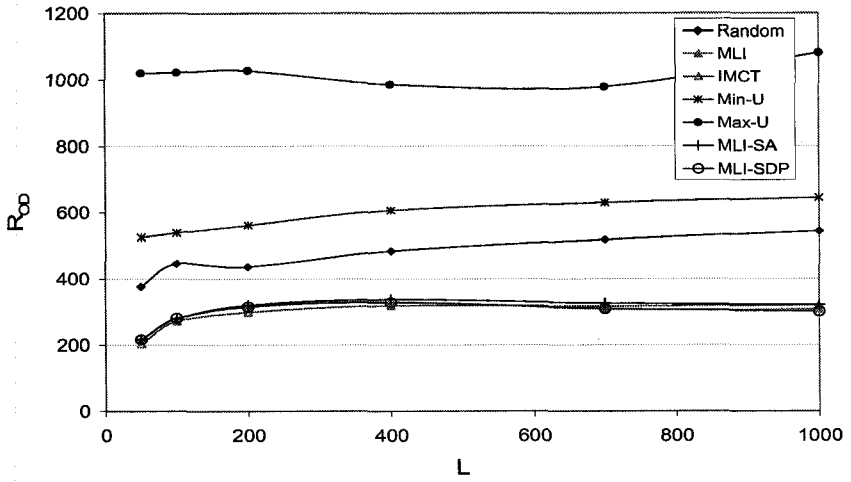


Figure D.8: Response Time of On-Demand Requests for Case MM.10

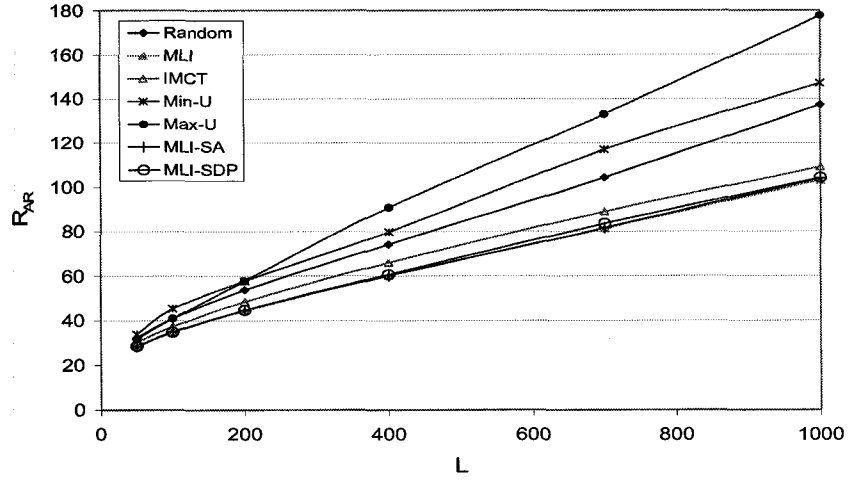


Figure D.9: Response Time of Advance Reservations for Case MM.10

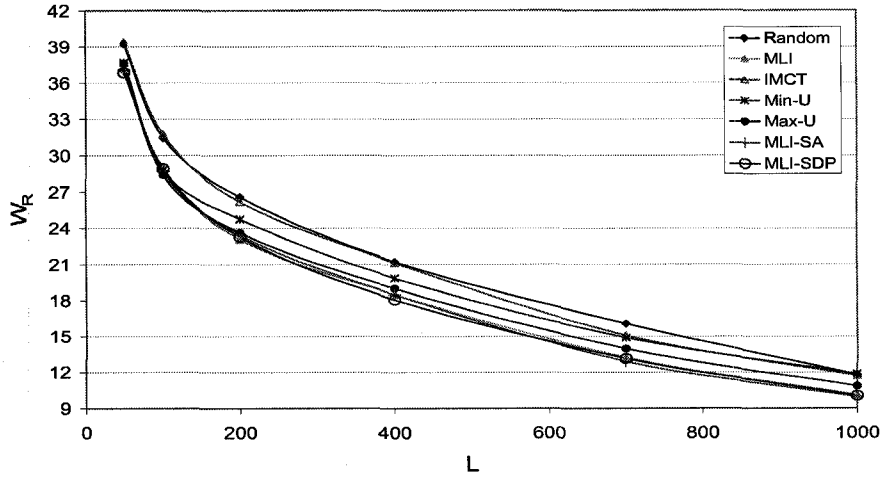


Figure D.10: Work Rejected for Case MM.11

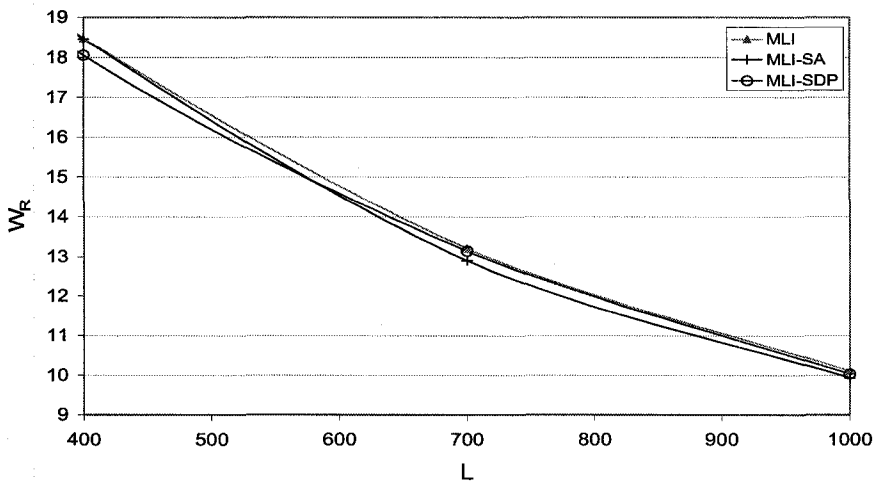


Figure D.11: A Closer Look at Work Rejected for Case MM.11

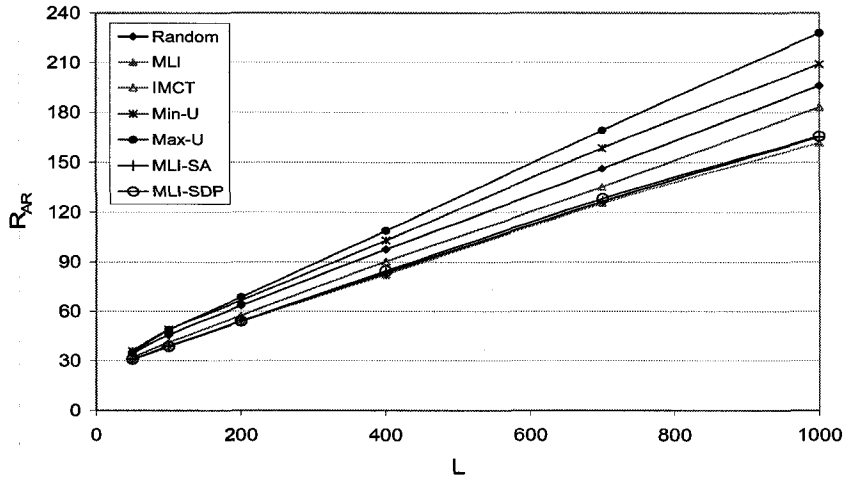


Figure D.12: Response Time of Advance Reservations for Case MM.11

Appendix E

E.1. Results for Case MR.3

This section presents results for Case MR.3 in Table 9.1. W_R, R_{AR} and Q_R for this case are shown in Figure E.1, Figure E.2 and Figure E.3, respectively.

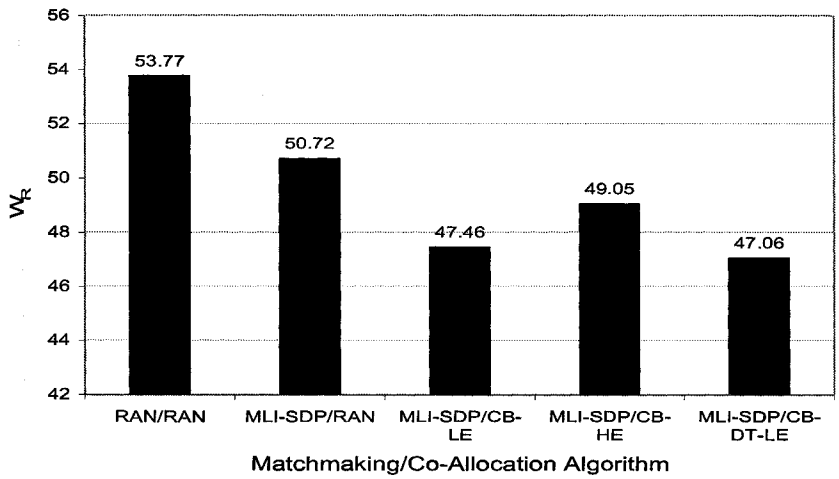


Figure E.1: Work Rejected for Case MR.3

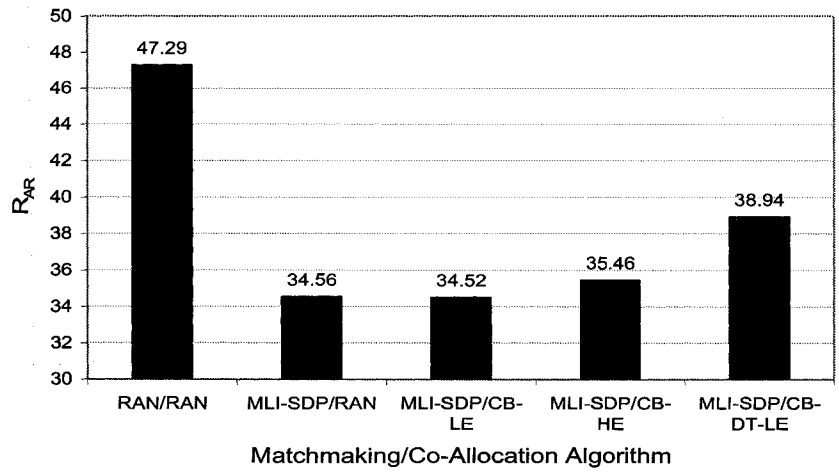


Figure E.2: Response Time of Advance Reservations for Case MR.3

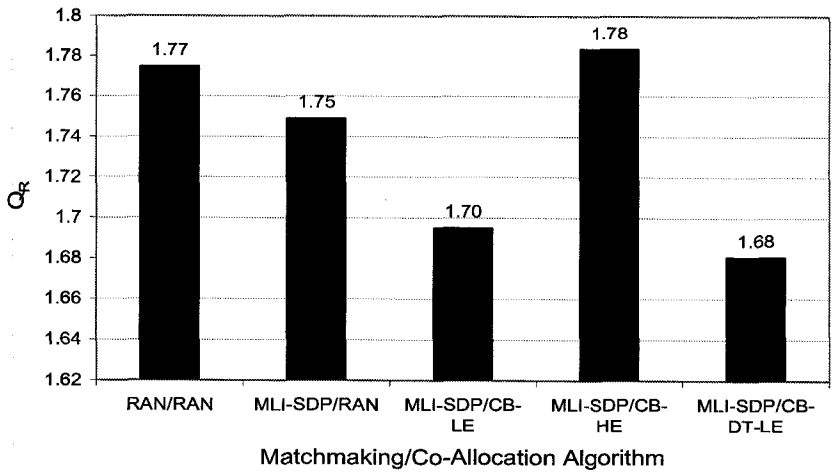


Figure E.3: Fairness for Case MR.3

E.2. Results for Case MR.4

This section presents results for Case MR.4 in Table 9.1. W_R, R_{AR} and Q_R for this case are shown in Figure E.4, Figure E.5 and Figure E.6, respectively.

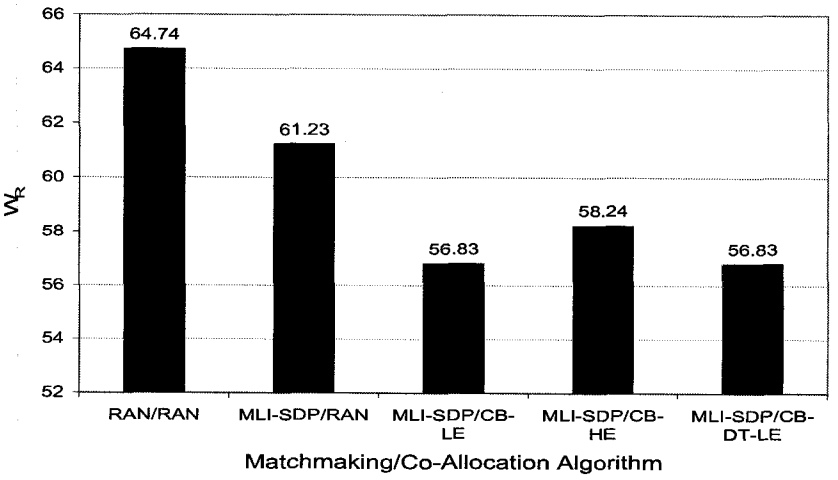


Figure E.4: Work Rejected for Case MR.4

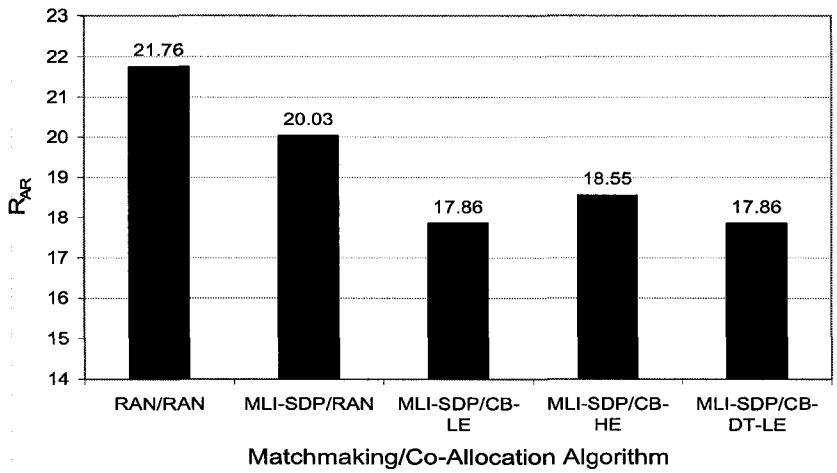


Figure E.5: Response Time of Advance Reservations for Case MR.4

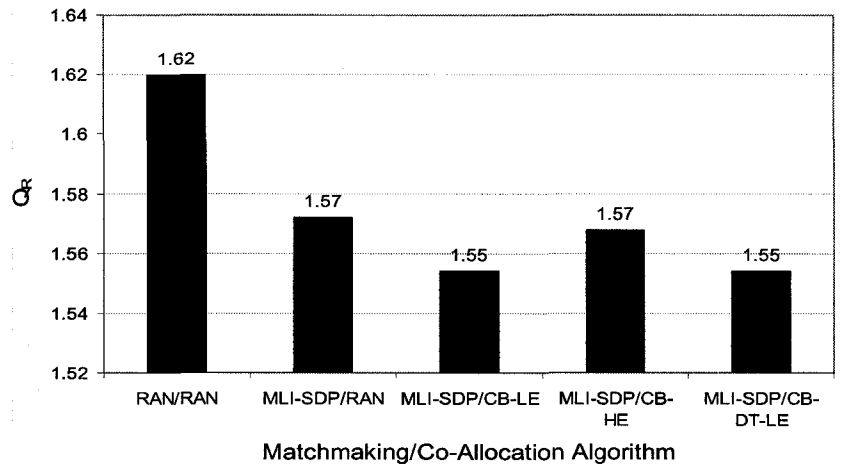


Figure E.6: Fairness for Case MR.4