Scheduling Advance Reservations with Priorities in Grid Computing Systems

by

Rui Min

A thesis

Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

SCHEDULING ADVANCE RESERVATIONS WITH PRIORITIES IN GRID COMPUTING SYSTEMS

BY

RUI MIN

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree of MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Rui Min

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Rui Min

Abstract

Grid computing systems utilize distributively owned and geographically dispersed resources for providing a wide variety of services for various applications. One of the key considerations in Grid computing systems is resource management with quality of service constraints. The quality of service constraints dictate that submitted tasks should be completed by the Grid in a timely fashion while delivering at least a certain level of service for the duration of execution. Because the Grid is a highly "dynamic" system due to the arrival and departure of tasks and resources, it is necessary to perform advance reservations of resources to ensure their availability, and to meet the requirements of the different tasks.

This thesis introduces two new scheduling algorithms for advance reservations including co-reservations, namely, *Reservation Scheduler with Priorities and Benefit Functions* (RSPB) and *Co-Reservation Scheduler with Priorities and Benefit Functions* (Co-RSPB). The algorithms consider the relative priorities of various reservation requests while scheduling reservations. The benefit function is used to quantify the "profit" for the client in order to remove the re-negotiation overhead in case of resource scarcity. Simulations are performed to compare proposed algorithms with an existing approach or with some comparison algorithms developed as basic comparison line in this thesis. The results indicate that the proposed algorithms can improve the overall the performance by satisfying larger number of reservation requests.

Acknowledgements

I would like to express my sincere gratitude to Dr. M. Maheswaran for his valuable guidance, encouragement, and his patience throughout the research project. I have greatly benefited from his expertise and constant help and advice, without which this thesis would not have been possible. I am also thankful to the thesis committee members, Dr. R. McLeod and Dr. A. Rueda, for being on my thesis committee. Also, I would like to thank Dr. R. McLeod for his valuable suggestions and help throughout the course of this thesis..

I express my acknowledgement to Kumaran Subramoniam for carefully reading early draft of this thesis and giving a number of useful suggestions.

I am deeply indebted to my parents for their support and sacrifices without which I certainly would not have reached where I have. I'd especially like to thank my husband, Ming-Dong, for his love, encouragement, and support. Finally, I express my hearty gratitude to my son Jesse and my daughter Cathy for their patience, understanding and for being constant source of inspiration.

Table of Contents

List of Figures

1 INTRODUCTION

1.1 Preliminary Remarks

The rapid advancements in microprocessor technologies and computer communications have facilitated the emergence of a new class of network-based applications. These applications are different from the current ubiquitous WWW and WWW-based applications. They require functionality that extends beyond the coordinated use of the network to encompass end systems, data repositories, sensors, visualization devices, and advanced human computer interfaces. The current Internet is not geared towards supporting such applications. Therefore, researchers have proposed a generalized, large-scale computing and data handling infrastructure called the *Computational Grid* (referred to as Grid in the following context) [FoK99a, Fos99, and JoG99].

1.2 What is Grid

The "Grid" in dictionaries has some concept of "network" or "mesh". The term "Grid" for network computing is analogous to the power grid. A power grid links source of electrical power together, and provides for widespread access to power with certain services. Similarly, a "Computational Grid" is "a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities" [FoK99a]. In other words, a "Grid" is an Internet sized network computing system with millions of machines distributed across multiple organizations and administrative domains to provide dependable, consistent, pervasive, and cost-effective access to diverse services. To achieve this

goal, a Grid needs to provide several services including: resource description and discovery mechanisms, reliable multi-party communications, resource management with quality of service (QoS), access control, data location, etc. [Man99].

Five major application classes are identified for computational Grids [Man99]:

- Distributed supercomputing;
- High-throughput computing;
- On-demand computing;
- Data-intensive computing;
- Collaborative computing.

Although the tremendous advancements in computer communications and distributed computing have enabled constructing and experimenting with several Grid prototypes (such as Globus [FoK97], Legion [Legion], and PVM [PVM] etc.) and experiments based on these prototypes, the Grid technology is very much in its infancy. Several key issues need to be investigated before the Grid technology can see widespread deployment. Some of these issues include developing:

- Efficient mechanisms for location independent use of distributed components;
- Efficient and highly scalable resource discovery schemes;
- Mechanisms for efficient resource allocation and reservation;
- Quality of service brokering.

1.3 QoS Requirements in Grid

The Grid is a highly dynamic system. The components of the Grid that support the services are referred to as the *resource providers*. Similarly, the components of a Grid that use the services for problem solving are called *resource consumers*. In a Grid system, the resource providers are

likely to be owned and administered by different organizations and possibly governed by different local policies. This means the resource providers will be committing varying amounts of the resources to provide the services to the Grid depending on the local policy and local demand for the resources. The resource consumers can also belong to different users with varying levels of subscription and privileges. With traditional resource allocation mechanisms, the fluctuations in the supply and demand situation in a Grid will impact the level of service delivered to the resource consumers. Depending on the criticality of the applications associated with a resource consumer, this may not be acceptable. To ensure that the sustained level of service delivered to an application is within its requirements, the application's *quality of service* (QoS) requirements should be considered while allocating the resources. The mechanisms involved in implementing the QoS requirements of an application vary with the specialization of the Grid. For example, in a high-throughput computational Grid, an application may be implemented by allocating time on a high-performance machine. Whereas, in a collaborative computing Grid, a session might need the co-allocation of several resources. In such a situation, the QoS requirements of an application should be mapped onto several resources. Due to the uncertainties of resource availability, it is necessary to support advance reservations to provide QoS guarantees in a Grid system.

1.4 Basic Concepts in QoS

Quality of service (QoS) represents the set of those quantitative and qualitative characteristics (referred to as QoS parameters) of a distributed system necessary to achieve the required functionality of an application. These QoS parameters are service specific. Different applications may have different subsets of QoS parameters with various values required. For example,

bandwidth, delay, throughput, jitter, etc. may be the relevant QoS parameters for a communication service. Whereas, CPU times and deadlines may be the relevant QoS parameters for an application in a computational Grid. For some applications, these parameters may be negotiable. For this kind of applications, the user may receive a certain degree of benefit if the system provides a certain level of service measured by required subsets of QoS parameters [ChS98] [VoK95].

QoS guarantees concerns user's benefit. It signifies that the QoS received by the user would not fluctuate with changes in resource usage by other applications and with changes in system state, such as servers coming on-line and going off-line. QoS guarantees are generally grouped into classes such as *hard QoS*, *soft QoS* and *best-effort QoS*. *Hard QoS* signifies that the user will receive required QoS every instance. *Soft QoS* signifies that the user will receive required QoS within a certain specified fraction of the instances. Best-effort QoS signifies that the user will not receive any guaranteed QoS [ChS98].

To provide an increased expectation of promised QoS guarantees while allocating resources, *reservation* concepts were first introduced in the area of communication network QoS [ZhD93]. There are two modes of reservation: *immediate reservations* and *advance reservations*. *Immediate reservations* are also referred to as *allocation* where reserved resources are allocated immediately. *Advance reservations* resource reservations in advance and the resources are used in the future. This increases the expectation that resources can be allocated when demanded. Without *advance reservations*, the user of the system may have more chance to encounter degraded service or even rejections.

In practice, many applications may have very large resource requirements and require multiple resources simultaneously. For example, in a collaborative Grid, an application may require resources from multiple computers and networks to execute. The mechanism that deals with allocating multiple resources simultaneously is called *co-allocation*. Advance reservation of multiple resources for a specific duration is referred to as *co-reservation*. Another frequently used mechanism in QoS-driven *resource management system* (RMS) is *admission control*. *Admission control* ensures that all applications accepted by the system will get guaranteed QoS service by admitting an application only if there are sufficient resources in the system [ChS98].

An admission control process can make a decision either in a simple yes-or-no form or a more sophisticated form that allows *negotiation* between the user and the system. A *negotiation* is a process of making an agreement between the user and the system about decreasing the value of a set of QoS parameters. The system tries to maximumings benefits to the user via the negotiation, therefore, making the system work in a most beneficial manner. The decrease in the level of service that is agreed upon during negotiation is referred to as *graceful degradation* and it is caused by scarcity of system resources [ChS98] [VoK95].

Negotiation is usually performed in following way: the system provides several QoS degradation options to the user, such as decreasing the reservation value for resources or delaying the starting time for advance reservations. The user selects an option that can give the user the most benefit, then, informs the system to make an agreement. Once the user gets a confirmation from the system, a contract between the system and the user is assigned. *Negotiation* mechanism allows

the system to provide more efficient service for more users under QoS constraints. This *negotiation* process adds considerable overhead to the system due to multiple messages being transmitted between the system and the user.

Another important QoS concept used in this thesis is a benefit function. Benefit functions are an abstractions developed to model an application's QoS requirements and preferences in the communication network area. "The benefit function is a multidimensional graph specifying the benefit that the user receives if the system provides a certain level of QoS. The dimensions of the benefit function correspond to QoS metrics of interest to the application. The benefit function is especially useful for facilitating graceful degradation between the application and the system." [ChS98]. If the system is not able to provide desired levels of QoS for an application due to resource scarcity, the benefit function can then be used to make intelligent decisions without asking the user regarding which QoS metrics to degrade, and by how much. Thus, reducing the re-negotiation overhead.

Figure 1.1 shows some examples of benefit functions for CPU reservation in a Grid system. Although, in these examples, benefit functions are used only for quantifying the CPU requirements and only have two dimensions, it may be used for other resources as well as the time constraints and can be extended to multiple dimensions to support multiple QoS parameters.

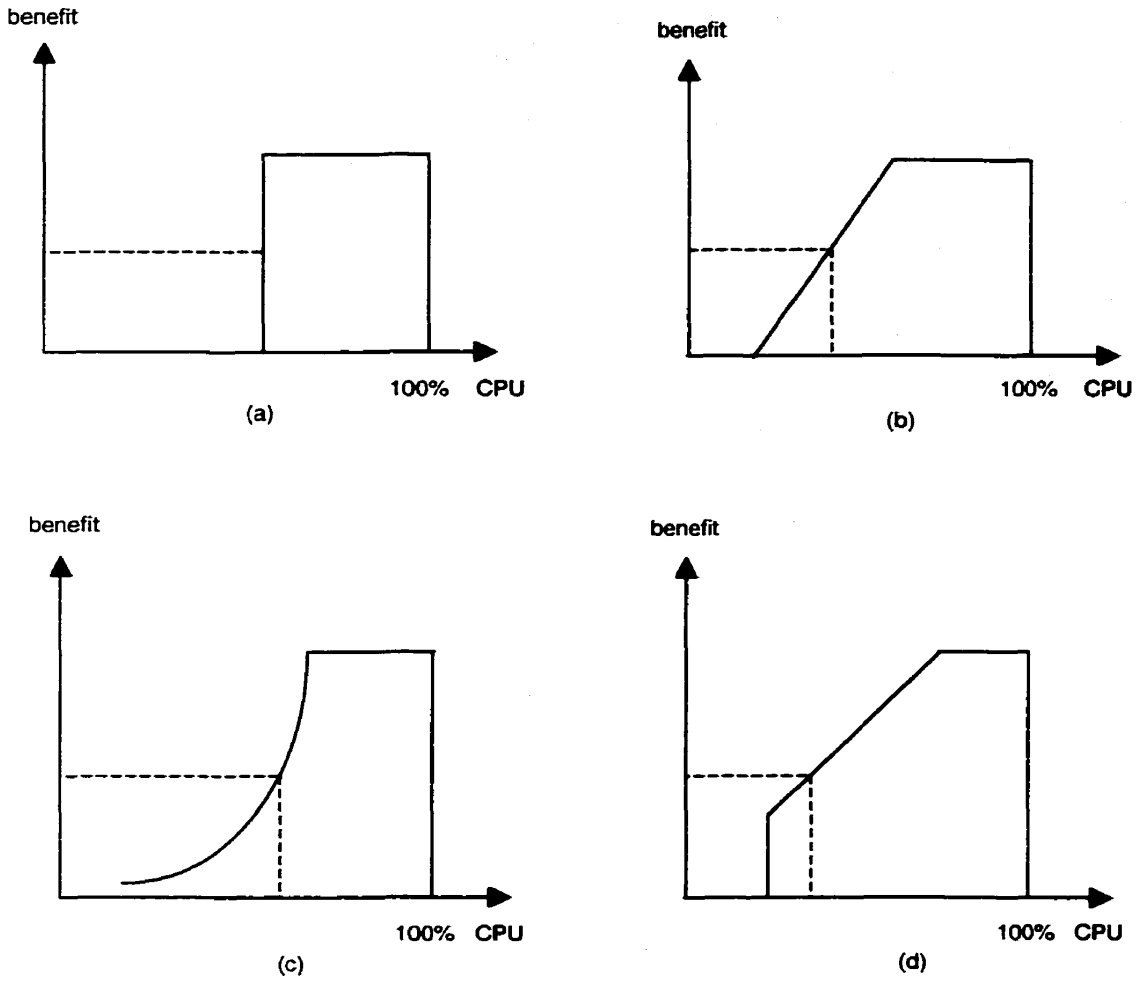


Figure 1.1: Some examples of benefit function shape.

Figure 1.1(a) shows a benefit function where the application does not gain *any* benefit if the system does not reserve at least a certain fraction of the CPU. Figures 1.1(b), 1.1(c), and 1.1(d) show cases where the application gains a reduced amount of benefit even if the system reserves less amount of CPU than what is ideally required. From the different benefit functions in Figure 1.1, it can be noted that some requests have "hard" QoS requirements and others have "soft" QoS requirements. The requests with soft QoS requirements get some benefit even if the system does not reserve the desired CPU percentage for them, although, the amount of benefit gained will be

lesser than a amount of benefit gained if the system would have reserved the desired CPU percentage for them.

In case of resource scarcity, the RMS requires a way to evaluate the relative importance of multiple, different applications which compete for the resource. The priority can be used to represents the application's importance. Depending on the different design objective, the priority can be determined by different aspects. For example, in a commercial system, the priority can be determined by the cost that the user is wiiling to pay for a service. In a military system, the priority can be determined by the importance of the user or of the application [ChS98].

1.5 Motivation and Scope of the Thesis

As discussed above, Grid computing is an emerging paradigm for next generation distributed computing. The Grid is a highly dynamic environment with on-line and off-line servers, and with continuously varying demand from the clients. In such an environment, it is necessary to consider QoS requirements of different clients to ensure that the resources are used in the most beneficial manner. Due to the uncertainties of resource availability in a dynamic system such as the Grid, it is necessary to support advance reservations to provide QoS guarantees.

In practice, different design objectives of a Grid system leads to different requirements of a reservation algorithm. For example, in a high-throughput computational Grid, an application may request to schedule reservations on a single high-performance machine. Whereas, in a collaborative computing Grid, an application may request to schedule reservations on multiple machines simultaneously (sometimes some machines may be directly specified) in order to

guarantee all required resources available while demanding. Therefore, it is necessary to design different algorithms supporting advance reservations for different purposes.

This thesis presents an overall resource management architecture for a Grid environment and proposes two algorithms for scheduling advance reservations on resources. *Reservation Scheduler with Priorities and Benefit Functions* (RSPB) schedules reservations on a single machine. *Co-Reservation Scheduler with Priorities and Benefit Functions* (Co-RSPB) schedules reservations on multiple resources simultaneously. The two algorithms schedule reservations while considering the relative priorities of the various reservation requests. Although, only CPU resources are considered here, this approach may be generalized to other resources such as network and storage. Also, in this thesis, immediate reservations are modeled as advance reservations with current time as the start time and a predefined length for the duration. This allows us to unify advance and immediate reservations.

In RSPB and Co-RSPB, each reservation request has an associated benefit function that quantifies the "profit" accrued by the client, by securing the resource at the requested level. When the client is willing to negotiate for lower service levels, it could indicate this by providing a benefit function that shows a reduced but positive benefit for lower resource levels. This facility provided by the benefit functions removes the need for negotiations when there is a resource scarcity.

Furthermore, in Co-RSPB, requests for fixed machine and floating machine are also considered in order to satisfy some applications with special resource requirements. Requests for fixed

machines require that, only specific machines can be mapped to each sub-request of an application. Requests for floating machines have more flexibility, so that all sub-requests of an application can be mapped to any machines in the system if the machine can satisfy the sub-request's QoS requirement.

Both RSPB and Co-RSPB can be implemented on top of a CPU scheduler such as *the Dynamic Soft Real Time* (DSRT) system [ChN97, DSRT] or a QoS enhanced operating system kernel such as QLinux [GoG96, QLinux].

The proposed algorithm RSPB is compared with an existing approach. The simulation results indicate that the RSPB can improve the overall the performance by satisfying a larger number of reservation requests.

Because there is no open literature available to compare with Co-RSPB, we also developed two comparison algorithms *Co-reservation Scheduler with Best Fit scheme* (Co-RSBF) and *Co-reservation scheduler with Best Fit and Refine scheme* (Co-RSBFR) as a base line to see the performance of Co-RSPB. In order to evaluate the performance of proposed algorithms in terms of QoS, a system benefit calculation model is developed. The simulation results indicate that Co-RSPB has a very good performance by satisfying larger number of reservation request.

1.6 Structure of the Thesis

In the following chapter, related work that appeared in the open literature is discussed. An architecture for a Grid *resource management system* (RMS) and how the proposed scheduling

algorithms fit into the architecture is examined in chapter 3. The reservation scheduler algorithm RSPB is presented in chapter 4. Using simulation studies, RSPB is compared with an existing resource reservation algorithm and simulation results are also discussed in this chapter. In chapter 5, co-reservation scheduler algorithm Co-RSPB and two comparison algorithms Co-RSBF, Co-RSBFR is presented. The complexity of running time of these three algorithms is analyzed. The simulation results and performance for three algorithms are also discussed. Finally, chapter 6 summaries the thesis and points out directions for future work.

2 RELATED WORK

2.1 Preliminary Remarks

A considerable amount of literature has been emerged on supporting advance reservations in the context of network QoS that involves *bandwidth* guarantees. Network QoS can be given by using a well-defined QoS model and a setup protocol such as RSVP [ZhD93]. However, the concept of advance reservations is relatively new in the realm of Grid computing. There are very few publications on this topic. This chapter will give a brief literature review on these two topics and other topics directly relevant to this thesis.

2.2 Advance Reservations for Communication Networks

Foster et al. [FoK99b] proposes *A Globus Architecture for Reservation and Allocation* (GARA) that enables co-reservation and co-allocation of heterogeneous resources (such as process, flow, disk object, memory object, etc.) for end-to-end QoS guarantees in emerging, network-based applications. It also addresses issues such as dynamic discovery and independently controlled and administered resources. GARA treats both reservations and computational elements as first class entities, allowing them to be created, monitored, and managed independently and uniformly. A prototype of GARA implementation is described and performance results are provided to quantify the costs of the techniques.

Schelen el al. [ScP98] describes an architecture supporting end-to-end resource reservations through agents. An agent in each domain in the network performs admission control for immediate and advance reservations. The architecture allows immediate and advance reservations to share network resource without pre-partitioning. Information about advance reservations is used to perform admission control for immediate reservations. In other words, information delivered from advance reservations help prevent immediate reservations from being rejected or even preempted. Simulation results are provided to show the effects of providing advance reservations with this model and the cost in terms of resource utilization, the probability of rejecting and preempting an immediate reservation. Schelen et al. [ScN99] provides a prototype implementation of this model and focuses on obtaining performance measures for admission control within a single link-state routing domain.

Berson et al. [BeL98] introduces a server-based architecture supporting advance reservations. It is domain-based, and it allows simple functioning with inter-domain routing. In this architecture, there is no reservation or multicast routing state needed in the routers until the reservation becomes active. It allows applications to request advance reservations without the application running during the length of the advance reservations.

Ferrari et al. [FeG95] discusses the requirements of the clients of an advance reservation service, and distributed design of a multi-party, real-time communication scheme for such a service. Simulation results are provided to show the performance and some of the properties of these mechanisms.

2.3 Advance Reservations for Grid Computing

Depending on different design objectives of a computational Grid, a reservation algorithm can be developed to support scheduling reservations either on single machine or on multiple machines.

2.3.1 Scheduling Advance Reservations on Single Machine

Garimella [Gar99] implements an *Advance Reservations Server* (ARS) that works in conjunction with the DSRT [ChN97] to reserve CPU resources in advance. In ARS, the client needs to specify some QoS parameters such as the percentage of CPU required as well as start time and duration. Once the reservation request is admitted, the reserved resources will be available for the client after the start time for the duration at the predefined percentage. However, in practice, most applications have QoS requirements that are negotiable. Because ARS does not support re-negotiations, it leads to higher number of rejected reservation requests.

The *Resource Broker* (RB) proposed in [KiN00] integrates with the ARS presented in [Gar99]. The RB improves ARS to give a fast and constant response by using a CPU resource broker model with a new admission control and also improves ARS by providing multiple negotiation options for the clients. However, the occurrence of re-negotiation adds considerable overhead to the system. Further, in order to allocate a resource to multiple competing applications, the admission control algorithm requires a way to evaluate the relative importance of the different applications. In this way, the admission controller can make decisions to reject less important applications first to ensure a group of clients get the most benefit.

2.3.2 *Scheduling Advance Reservations on Multiple Machines (Co-reservation)*

Smith et al. [SmF00] proposes and evaluates several algorithms for supporting advance reservations in supercomputer scheduling systems. These algorithms improve traditional scheduling algorithms by unifying scheduling traditional tasks from job queues with the reservation requests. These advance reservations allow users to request multiple resources simultaneously from scheduling systems at specific times. However, [SmF00] allocates the "time slots" exclusively, i.e., the resources are not reserved in a shared fashion by multiple clients for the same duration. The applications are assumed to operate on a "best effort" basis and the reservation requests are assumed to have different priority than the applications. These differences in priorities are considered while the reservations and applications are scheduled by the system. Fixing machine scheduling is not supported.

2.4 QoS Supported CPU Scheduler

The two advance reservation algorithms developed in this project are based on the assumption that the local management service support immediate reservations. With traditional general purpose operating systems such as Windows NT and Unix which schedule processes based on *the Time Sharing* (TS) principle, the contracted advance reservations cannot be guaranteed when they are allocated to the local resource. Therefore, a CPU scheduler supporting immediate reservations is needed.

The Dynamic Soft Real Time (DSRT) System based on research in [ChN97] is a user-level scheduler, which can provide processor CPU guarantees to soft real time periodic and aperiodic

tasks. The DSRT system is built on various platform including SunOS 5.7, SGI IRIX 6.5, Linux (RedHat 6.2), and Windows NT. It provides functions including protection among real-time (RT) processes, fairness among RT and non-RT processes, rate monotonic scheduling, and a fix to the UNIX security problem.

A QoS enhanced Linux Kernel for Multimedia Computing (QLinux) is a replacement of Linux 2.2.x kernel with the ability to provide quality of service guarantees. It includes the following features:

- *Hierarchical Start Time Fair Queuing* (H-SFQ) CPU scheduler;
- *Hierarchical Start Time Fair Queuing* (H-SFQ) network packet scheduler;
- *Lazy Receiver Processing* (LRP) network subsystem;
- Cello disk scheduling algorithm [not stable yet].

When a QLinux is enabled, any selected combination of these features will replace the standard features/schedulers available in Linux.

The H-SFQ CPU scheduler is based on research in [GoG96]. Goyal et al. [GoG96] presents a Start-time Fair Queuing (SFQ) algorithm for operating system supporting variety of hard and soft real-time as well as best effort applications in a multimedia-computing environment. SFQ enables "hierarchical partitioning of CPU bandwidth, in which an operating system partitions the CPU bandwidth among various application classes, and each application class, in turn, partitions its allocation (potentially using a different scheduling algorithm) among its sub-classes or applications."

2.5 Benefit Functions in Related Projects

ERDoS [ChS98] describes "the End-to-End Resource Management of Distributed Systems QoS Architecture, which enables adaptive, end-to-end, scalable resource management of distributed systems. The purpose of the architecture is to delineate a common application programmer's interface (API) between and within the resource management layers so work by multiple research groups can be integrated into a common product." It provides domain-specific application (such as command and control, multimedia, and medical domains etc.) QoS support. ERDoS employs a Hierarchical System Model to deal with heterogeneous resource and system scalable problem. QoS level for each application is effected depending on scheduling policy when the state of system changes. Benefit functions are used to communicate user QoS preferences of an application to the resource manager. In case of failure or scarcity of resource, the resource manager uses this information to gracefully degrade application QoS, therefore, minimize the adverse effect on each application user.

Maheswaran [Mah99] presents a dynamic and centralized scheduling algorithm for computational resources in a network computing system. The algorithms takes into account applications' QoS requirements when scheduling. The level of service received by each application is quantified by a benefit function defined for that application. The objective of the algorithm is maximizing the total benefit provided to the applications. Simulation results are presented to evaluate the performance of the algorithm.

3 SYSTEM MODEL

3.1 Preliminary Remarks

The Grid *Resource Management System* (RMS) in this thesis is designed to support advance reservations and immediately allocations of resources, which is dedicated to the system. In order to provide QoS guarantees, all applications coming to Grid for service with QoS requirements are required to reserve resources before allocation. Applications requesting best-effort service are not required to reserve resource in advance. The reservations can be either immediate or advance reservations. For each application requesting QoS service, the user needs to specify following parameters:

- Machine type on which the user desires to reserve resource;
- The starting time. It can be either current time for immediate reservation or future time for advance reservations;
- The duration for the application execution;
- Preferred priority according to the importance of the application or the cost level which the user is willing to pay; and
- Benefit function shape which indicates the user's preference about QoS degradation in case of resource scarcity.

3.2 A Grid Resource Management Architecture

The resource management architecture for the Grid is shown in Figure 3.1. The architecture shows the components involved in advanced reserving and immediately allocating resources for a user request.

In this architecture, when a user logs onto the Grid, the Grid launches a Client. The Client authenticates the user. If the application request best-effort service, the *application information service* (AIS) uses historical information and learning algorithms to predict the resource requirements of an application's running. For applications with hard or soft QoS requirements, the resource requirements are specified by the user during application submitting stage. The Client then interacts with the QoS broker to implement the applications that are submitted to it. The QoS broker provides a virtual resource to the Client with the desired QoS attributes. This virtual resource will also provide feedback to the Client if the capability of the virtual resource drops sufficiently to affect the QoS attributes.

Depending on the extent of the Grid, there will be thousands of QoS brokers. When a Client needs service it will connect to a QoS broker that is in its neighborhood. Once the Client connects to the QoS broker, for each application, it submits the resource requirements along with the desired QoS constraints to the QoS broker. By default, the application is provided the best-effort service. Based on the level of subscription, the QoS broker will determine whether the level of service requested by the Client is valid. This preliminary admission control will preclude any Client from monopolizing the resources.

Once the QoS broker receives a valid request for resource allocation or reservation, it contacts the *admission controllers* (AC) to implement it. The set of ACs contacted by a QoS broker is determined by the resource discovery agent. The resource discovery agent could be implemented in several ways. Several alternative approaches for scalable, high-performance resource

discovery agents for a Grid system are evaluated in [MaK00a, MaK00b, and Mah01]. The trade-off of using general-purpose resource discovery/naming systems versus Grid specialized systems are discussed in [MaK00a, MaK00b, and Mah01]. When there is multiple ACs that are willing to schedule allocations and reservations towards a resource request, the QoS broker can use different strategies in handling such a situation. One strategy would be to rank the resource offerings and pass them onto the Client so that the Client could choose one to implement the application. Another strategy is to select one resource offering using some heuristics and Client supplied information. Yet another strategy would be to poll the eligible ACs in sequence and select an offering based on some criteria.

Once the QoS broker finds an agreeable resource reservation from an AC, it forms a QoS contract with the AC. Because the resources may not exclusively be under the Grid control (e.g., the resources may be used by owners without the Grids intervention), therefore the QoS contract formed between QoS broker and AC may be violated. Therefore, the QoS broker should monitor for any possible violations of the contracts and initiate renegotiations with the Client and the admission controller to remedy them.

The QoS contract violation could be caused by two reasons: (a) fluctuations in resource availability and (b) variation in the resource requirement of the application. The resource management architecture presented here decouples the contract into two stages. This enables more robust scheduling environments because when an application overruns the expected resource requirement, the Client needs to renegotiate the contract with the QoS broker. A rogue application would not affect the resource reservations of the other applications.

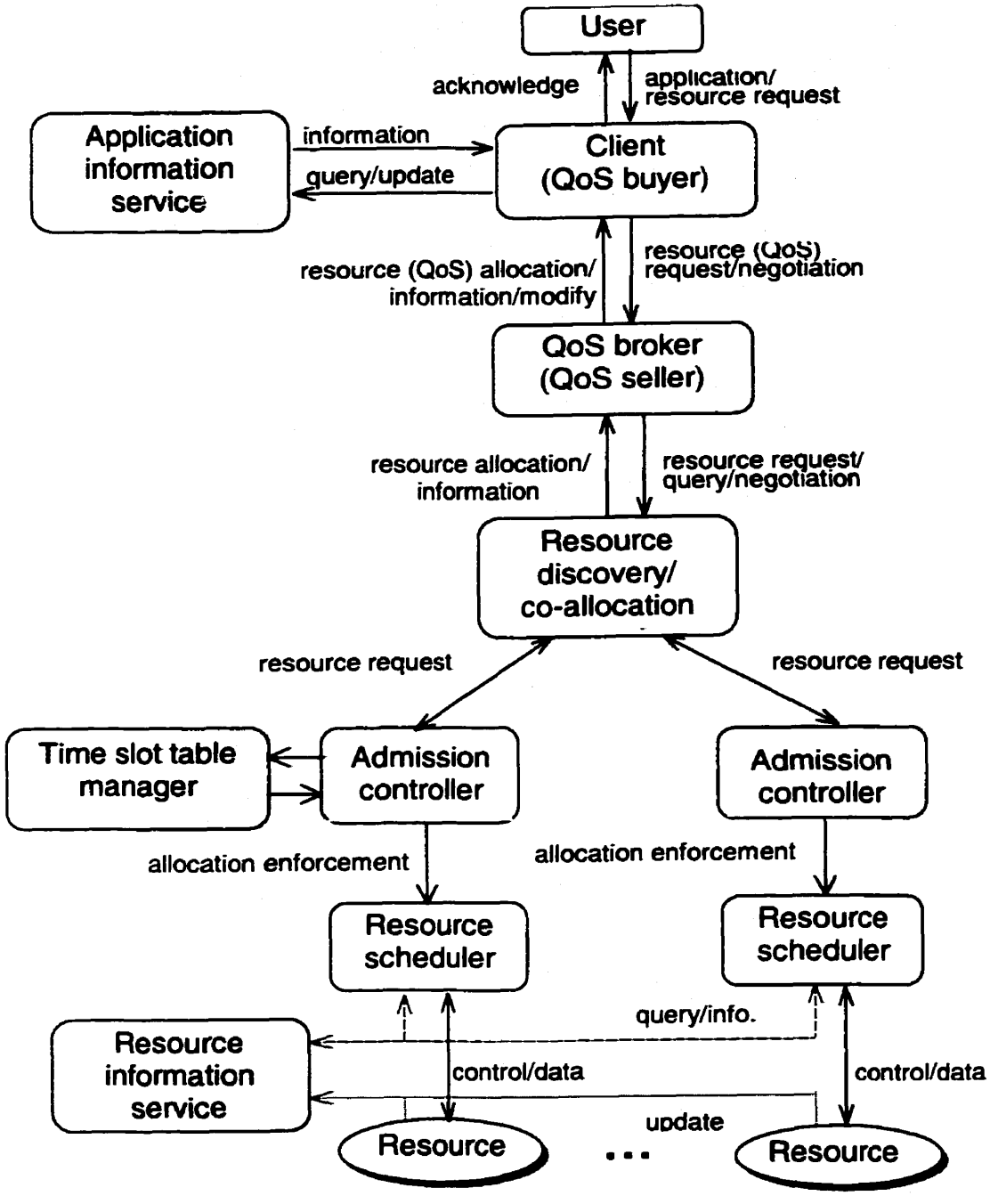


Figure 3.1: A resource management architecture for the Grid.

Resource Information Service (RIS) provides information of resources in terms of their load, operating system version, installed software, availability, etc. the contents of information service are updated by automated discovery and publication mechanisms. The information of a RIS is queried by resource discovery agent to locate resources with desired QoS characteristics.

3.3 Scheduling Reservations

The *admission Controller* (AC) has two responsibilities:

- Making decisions about accepting an application's request using proposed algorithm. Applications requesting the best-effort service are always accepted by the AC;
- Enforcing allocation of reserved resources when an allocation request is received.

When an AC receives a reservation request from a QoS broker, the AC scans the time slot tables in the required time duration for all machines which are controlled by the connected resource scheduler. If the QoS requirements of the application (either hard QoS requirements or soft QoS requirements) can be satisfied, the AC updates the time slot tables for selected machines with the amount of promised resources and responds the user using the unique user ID. The value of the parameter that has soft QoS requirements can be obtained from a user's preferred benefit function shape and the parameter value that can give the highest benefit to the user. Once the application's request is accepted by the AC, a contract about the application's QoS guarantees between the user and the system is assigned. This procedure only marks the time slot tables for reservations. The reserved resource is not effective until an allocation request is received. If the request is rejected, the QoS broker will try to contact with other Acs provided by the resource discovery agent or re-negotiate with the user until the requested resource is found or the user gives up the reservation.

The time slot table is updated every time interval, which is equal to a time slot, in order to keep the first slot always being the current time slot. When the start time of an application is in the first slot in the time slot table, the system is ready for the application's execution using reserved resources. Upon usage request's arrival, the AC sends allocation requests to the resource scheduler. The resource scheduler arranges the resources for an application's execution. For the application that is still running at the end of the requested duration, the system will provide a best-effort service to it. If the application terminates before the end of the requested duration, the user can claim these reserved resources for other application's execution without additional reservation using the same user ID. Or the resources can be released automatically by the system.

The user also can modify or cancel the reservation before resources allocation occurs. In the case of QoS degradation or reservation cancellation, the AC modifies the corresponding time slot tables and cancels the QoS contract if reservation cancellation is required. On the other hand, if the user requests more resources to be reserved, the AC has to perform the admission control procedure again to make a decision as described above.

3.4 About the Time Slot Table

A *time slot table* is responsible for keeping track of current allocations and future reservations for resources. Each individual resource has a mapped *time slot table*. Figure 3.2 shows an example of a *time slot table*. It is a two-dimension Cartesian coordinate. The "y" axis represents the percentage of the resource. The "x" axis represents the time, which is divided into a number of slots. The first *time slot* represents the slot in which the *current time* is included.

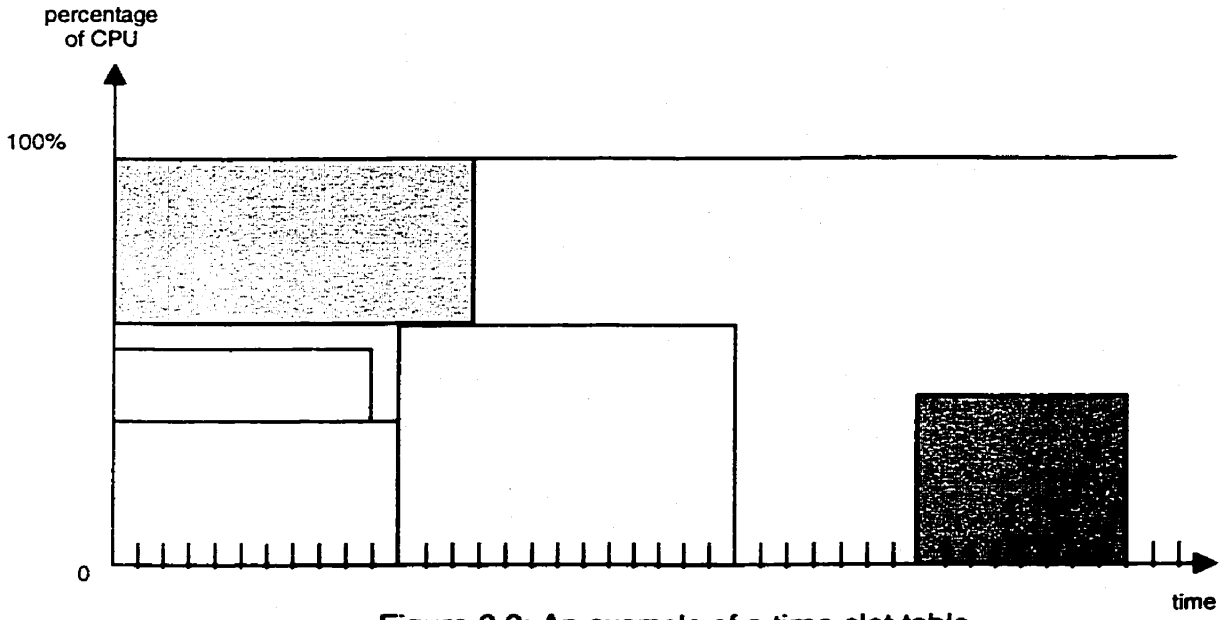


Figure 3.2: An example of a time slot table

A *time slot table manager* is used to provide functions to maintain a set of *time slot tables*. It has following responsibilities:

- Whenever a new resource is available for the system, the resource reports its percentage value that can be managed by the system to the *time slot table manager*. The *time slot table manager* then creates an empty *time slot table* for this resource;
- If a resource is assigned with an advance reservation, the *time slot table manager* is responsible for marking the *time slot table* associate with the resource in corresponding slots with the reservation value;
- If a reservation on a resource is canceled, the *time slot table manager* is responsible for erasing the reservation from the *time slot table* in corresponding slots with reservation value;
- If a resource leaves the system, the *time slot table manager* is responsible for removing the *time slot table* for this resource from the *time slot table manager*. Applications assigned on this resource should be rescheduled on other resources. This procedure is handled by the QoS broker;
- When the AC queries, the *time slot table manager* is responsible for providing reservation information on all resources;

- In order to keep the *first slot* as the *current time slot* the *time slot table manager* is also responsible for updating *time slot tables* every time interval, which is equal to a *time slot.*

The pseudo code of a *time slot table manager* is showed in figure 3.3. Class *slotManager* is a *time slot table manager* responsible for managing *time slot tables* for all machines controlled by the AC. Function *initialize* creates an empty *slotManager* at the system set up time. Whenever a machine comes in, function *create_slot_table* creates an empty *time slot table* for this machine. Then sends this to *slotManager* to manage. Function *add_reservation* marks *the time slot table* in specific slots for a specific machine when the reservation is assigned on this machine. If the slot is assigned with other jobs already, the value of reserved CPU for this slot is the sum of all reserved CPU value. Otherwise, the slot is marked with the new reserved CPU. Function *query_slot_table* is responsible for querying the reservation information on a specific machine in a specific slot. Function *update_slot_table* decreases slot order by one every slot time interval. This function ensures that the *current time* is always within the first slot. When a machine leaves the system or encounters some failure, function *handle_failure* removes *the time slot table* for the machine from slotManager. The reservations assigned on this machine needs to be rescheduled using the unique user ID. Function *cancel_reservation* is responsible for canceling reservations within given slots. Because the start time and the desired reservation duration from an application can be an arbitrary value rather than slotted time used in the system, therefore, function *calculate_start_slot* and *calculate_end_slot* interpret user's arbitrary time value to slotted value.

```
class SlotManager
   function Initializer()
      create an empty slotManager at system set up time;

   function create_slot_table(machineName mj)
      create an empty slot table table_mj;
      sent table_mj to slotManager;

   function add_reservation(machineName mj, startSlot t_start, endSlot t_end, CPUValue
CPU_rsv)
      get table_mj from slotManager;
      for (slot t = t_start, t<= t_end, t++)
         if (t contains some CPU value)
            get CPU percentage CPU_old already reserved in this slot;
            CPU_new = CPU_old + CPU_rsv;
            mark the slot with CPU_new;
          else
            mark the slot with CPU_rsv;
      endfor

   function query_slot_table(machineName mj, slot t)
      get table_mj from slotManager;
      return reserved CPU value in slot t;

   function update_slot_table()
      for each slot table in slotManager
         for each slot in a slot table
            new_slot_order = old_slot_order -1;
            if (new_slot_order < 0)
               cancel the slot with reservation in this slot;
          endfor
      endfor

   function handle_failure(machineName mj)
      remove table_mj for slotManager;

   function cancel_reservation(machineName mj, startSlot t_start, endSlot t_end, CPUValue
CPU_cancel)
      get table_mj from slotManager;
      for (slot t=t_star, t<=t_end, t++)
         get CPU percentage CPU_old already reserved in slot t;
         CPU_new = CPU_old – CPU_cancel;
         mark the slot with CPU_new;
      endfor

   function calculate_start_slot(double start_time, int slot_interval)
      get current_time;
      startSlot t = floorFunction((start_time - current_time)/slot_interval);
      return t;

   function calculate_end_slot(double start_time, double duration, int slot_interval)
      get current_time;
      endSlot  t = ceilingFunction((start_time + duration - current_time)/slot_interval);
      return t;
```

Figure 3.3: Functions of a time slot table manager.

4 RSPB: RESERVATION SCHEDULER WITH PRIORITIES AND BENEFIT FUNCTIONS

4.1 Preliminary Remarks

Reservation Scheduler with Priorities and Benefit Functions (RSPB), which schedules reservations on a single machine, can be used in an admission control for a system such as high-throughput computational Grid. RSPB schedules reservations while considering relative priorities of various application requests. Benefit functions are used to model user's QoS requirements. Machine load balancing is also considered. A detailed description of the algorithm is presented in this chapter. Simulation results and discussion are also given.

4.2 Assumptions

The algorithm is based on the following underlying assumptions. A centralized resource reservation scheduler is assumed, i.e., all the resource reservations are performed by a centralized unit. Requests arrive randomly based on a Poisson arrival process. Because the requests are arriving in a random fashion in real time, the reservation scheduler cannot wait until all the requests have arrived to commence the scheduling. It should make the decisions on the requests as each one arrives or makes a decision after the arrival of a batch of requests. This algorithm follows the later method. Once a request is granted the reservation, a contract for the reservation is signed between the application and the system. The reservation scheduler won't examine the same request more than once except the case in which a QoS violation occurred. This situation should be handled by a higher level QoS broker that engages in re-negotiation to

establish another reservation or a continuation of the current reservation. Based on the operating policies, the reservation scheduler may find another reservation or the application may operate under best-effort conditions.

4.3 Notations and Mathematical Model

Let m be the number of machines in the system. The machines are assumed to be homogeneous. Let CPU_sys_j represent the percentage of CPU of the j-th machine M_j that is dedicated to the Grid system. For each request R_k arriving at the reservation scheduler, following parameters are defined.

- $t_{k\text{-}start}$: start time of reservation for R_k;
- $t_{k\text{-}end}$: end time of reservation for R_k;
- $CPU_{max\text{-}k}$: minimum CPU requirement of R_k for delivering the maximum benefit to the application;
- p_k: priority of R_k, due to limited amount of resources, the scheduler cannot meet the demands of all the requests. When the overall demand exceeds the available resources, the objective of the reservation scheduler is to minimize the sum of priority of the requests that are rejected. This study uses heuristic approaches to achieve this objective.
- $B_k(p_{cpu})$: the benefit function associated with R_k. It gives the benefit which the client will receive if it is reserved CPU at the requested level. Figure 1.1 shows some of shapes that the benefit function could take for a reservation request. Although, in this project, benefit functions are used only for quantifying the CPU requirements, it may be used for other resources as well as the time constraints.

4.4 Reservation Scheduling with Priorities and Benefit Functions

This section examines RSPB reservation scheduling algorithm. In this study, each reservation request involves a single resource, i.e., no co-reservation of resources is considered here. Figure 4.1 shows the outline of the dynamic reservation scheduler. In this scheduler, dynamically arriving requests are collected for a predefined time interval to form a meta-request.

```
t=t0: scheduler start time
Δt: inter-schedule time
while (true)
        t = t + Δt;
        while (current time < t)
                get current request R;
                add R to Rmeta
                if (requested start time of R < t)
                        t = current time;
        endwhile
        scheduleRmeta(Rmeta)
endwhile
```

Figure 4.1: Outline of the dynamic reservation scheduler.

The dynamic reservation scheduler makes a decision upon receiving a *meta-request* using the *scheduleR*$_{meta}$ function that is shown in Figure 4.2. The *scheduleR*$_{meta}$ function is called in following two situations:

- When the *current time* is equal to the *current scheduling event time* that is equal to *t*;
- The requested *start time* of request *R* is less than *current scheduling event time t*. This ensures that requests with *start time* less than *current scheduling event time* is scheduled before *current time* is equal to *current scheduling event time*. One example of this kind of requests is immediate reservations.

```
function scheduleR_meta (meta-request R_meta)
(2)   R_k : the k^th request in R_meta;
(3)   M_j : the j^th machine in the system;
(4)   CPU_min-k : minimum CPU requirement when R_k
                  get lowest acceptable benefit from reservation;
(5)   load_heaviest_j(t_m, t_n) : the heaviest load of M_j within a reservation duration t_m to t_n;
(6)   CPU_smallest_j(t_m, t_n) : the smallest availability of CPU to reserve
                  for M_j within a duration t_m to t_n;
(7)   machQueue_hard : a machine queue used to store machines
                  which can satisfy hard QoS request of R_k;
(8)   machQueue_soft : a machine queue used to store machine
                  which can satisfy soft QoS request of R_k;
(9)   B_jk : benefit value which R_k can get from M_j;
(10)  R_nonsatisfy : meta-request used to store rejected requests;

(11)  for all requests R_k in R_meta
(12)      sort the requests in descending order by p_k;
(14)  for each sorted request R_k in R_meta
(15)      reset machQueue_hard and machQueue_soft
(16)      get t_k-start and t_k-end;
(17)      calculate CPU_min-k according to selected benefit function shape;
(18)      for each machine M_j in the system
(19)          get load_heaviest_j(t_k-start, t_k-end);
(20)          CPU_smallest_j(t_k-start, t_k-end) = CPU_sys_j - load_heaviestj(t_k-start, t_k-end);
(21)          if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_max-k)
(22)              machQueue_hard ← M_j;
(23)          else
(24)              if (machQueue_hard is empty)
(25)                  if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_min-k)
(26)                      if (machQueue_soft contain machine M_i)
(27)                          B_jk = B_k(CPU_ smallest_j(t_k-start, t_k-end));
(28)                          B_ik = Bk(CPU_ smallest_i(t_k-start, t_k-end));
(29)                          if (B_jk > B_ik)
(30)                              machQueue_soft ← M_j;
(31)                          else
(32)                              machQueue_soft ← M_j;
(33)      endfor
(34)      if (both the machQueue_hard and machQueue_soft are empty)
(35)          put the request R_k into meta-request R_nonsatisfy;
(36)      else
(37)          if (machQueue_hard is not empty)
(38)              select the machine with the lowest average CPU load within duration (t_m, t_n);
(39)          else
(40)              the machine in the machQueue_soft is the choice
(41)          mark the time slot table for this machine with requested reservation;
(42)  endfor
```

Figure 4.2: A priority and benefit function based scheduling algorithm for indivisible reservations.

Figure 4.2 shows the pseudo code for the *scheduleR*$_{meta}$ function. Line (12) sorts the requests in R_{meta} in descending order by the priority of the requests. This ensures that if multiple reservation requests require reservation in the same duration, the requests with higher priority will be scheduled first. This reduces the sum of rejected priorities thus ensuring the resources are used in the most beneficial manner. Line (17) determines the minimum CPU requirement for request R_k (i.e., CPU capacity at which R_k can provide the lowest acceptable benefit to the application according to the selected benefit function shape). This value will be used to determine if the request should be rejected or admitted (with graceful degradation) in case of resource scarcity and its hard QoS requirement cannot be guaranteed.

In this reservation model, a CPU resource may be temporally shared by multiple reservations, that is, multiple reservations may overlap in time. Therefore, we need to determine the current CPU usage for a given time interval before admitting a reservation. In Lines (19) and (38), the current CPU (machine) usage is determined using *time slot* tables that keep track of reservations that have already been accepted. The time slot tables only keep track of the partition of the CPU that is dedicated to the Grid and thus, managed by the reservation scheduler.

Line (21) determines whether the examined machine can satisfy the hard QoS requirement of request R_k. If yes, this machine is put into a machine queue named *machQueue*$_{hard}$ for later selection in line (38). Although it is not shown in the pseudo-code in Figure 4.2, the search for a machine that satisfies the reservation request can be stopped when a machine that satisfies the hard QoS is found. If none of the machines in the system can satisfy the hard QoS requirement of R_k, the reservation scheduler will attempt to schedule the reservation with a degradation of the

CPU requirement for R_k if the benefit function is provided to allow for the degradation. Lines (24) to (32) attempt to find a machine that satisfies this situation and this machine is considered to satisfy the soft QoS requirement of R_k. Lines (26) to (32) attempt to maximize the benefit delivered to the application by the reservation.

Line (34) and (35) deal with the case where the system cannot provide the requested level of service to R_k. The reservation scheduler checks the meta-request $R_{nonsatisfy}$ and sends rejections messages to the clients that submitted the reservation requests in $R_{nonsatisfy}$. The clients may resubmit their reservation requests with modifications and these submissions will be considered for reservation at the next scheduling event. Line (38) ensures that the load is distributed across the machines.

4.5 Simulation Results and Discussion

This section presents some results from a simulation study designed to evaluate the performance of the algorithm provided in the previous section. In this simulation study, the RSPB is compared with the *Resource Broker* (RB) [KiN00]. For comparing the two reservation schemes, a discrete event simulator was written using the PARSEC language (represents for PARallel Simulation Environment for Complex System), which is a C-based discrete-event simulation language [BaM98, PARSEC]. In the simulations, the reservation requests arrived randomly according to a Poisson arrival process. With each request, several attributes were associated to define parameters like reservation start time, end time, percentage of resource, shape of the benefit function, priority, etc. The benefit functions were restricted to the four shapes in Figure 1.1 such

that each function is used by 25% of total number of requests. Time slot table for each machine is maintained by a modified version of the data structure called Interval Skip List [HaJ96, ISList]. The following parameters are true for the simulation results presented unless stated otherwise.

- 10 machines participated in the simulation;
- Each machine dedicated 70% of CPU to the Grid system;
- Each reservation requested for CPU usage was uniformly distributed in [20%, 70%];
- Requested duration was uniformly distributed in 20-300 time units (PARSEC clocktype);
- Requested starting times were uniformly distributed over 4,320 time units;
- Time was slotted with a granularity of one time unit. The simulation time is 100,000 time units. It created an average of about 10,505 requests.

Figure 4.3 shows the variation of the number of rejections with the number of requests. The simulation time ranged from 10,000 to 120,000 time units and the averages of about 1063 to 12,604 requests were created. Figure 4.4 shows the variation of the number of rejections with the number of machines. The number of machines participated in this experiment ranged from 2 to 20. Figure 4.5 shows the variation of the number of rejections with the average of duration. The average requested duration varied in the [30, 350] ranges.

From Figures 4.3, 4.4, and 4.5, it can be noted that the number of rejections for RSPB is considerably lower than that for RB in all three cases. This is because the benefit function is used in RSPB. The requests that need to be re-negotiated in RB may be admitted in RSPB with graceful degradation of QoS, provided they have specified soft QoS requirements using their benefit functions. This reduces the number of rejections.

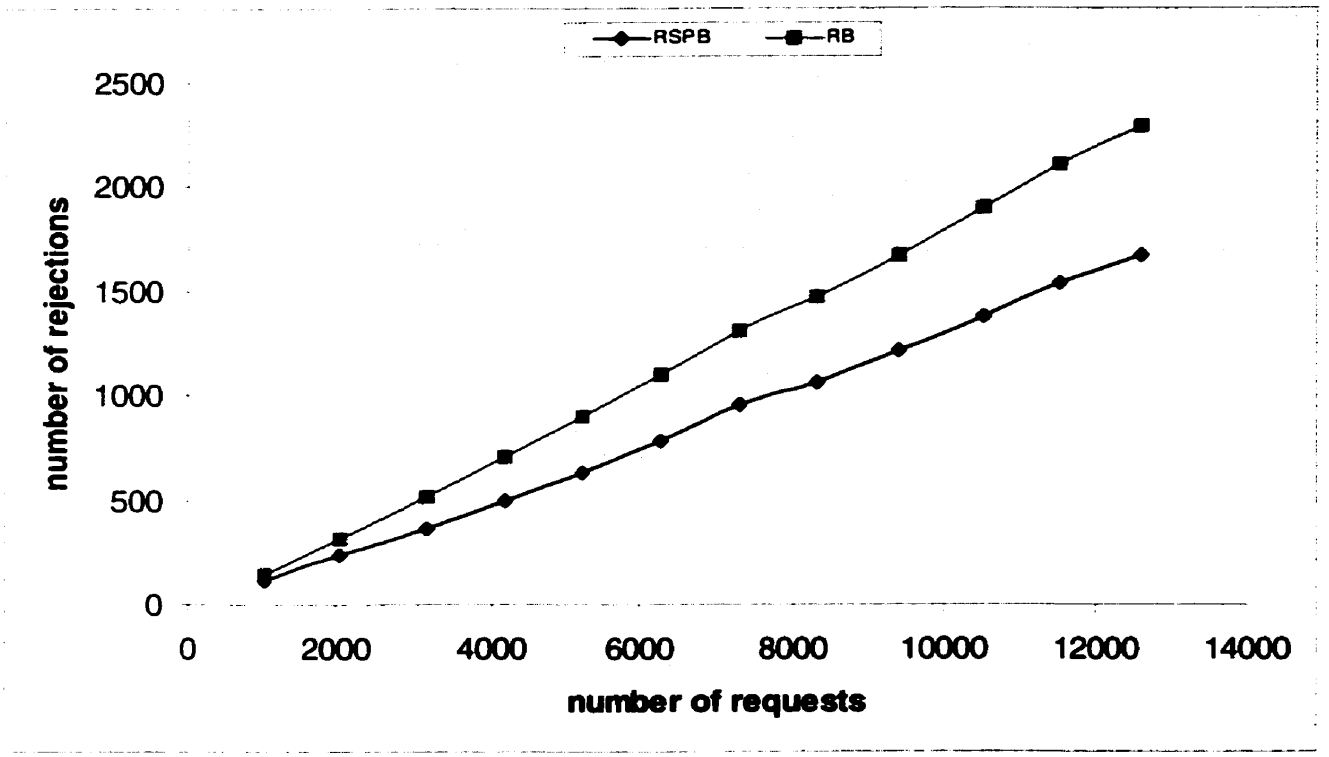


Figure 4.3: Number of rejections versus number of requests.

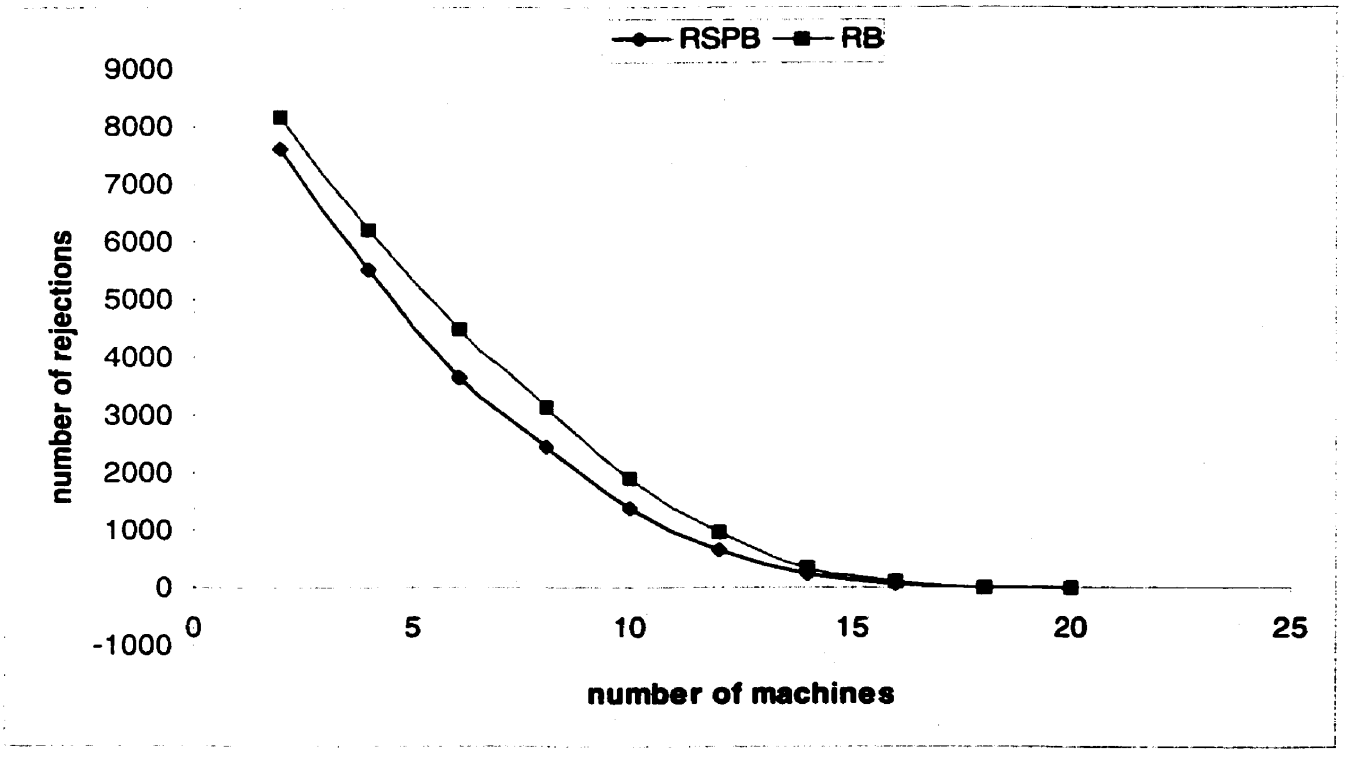


Figure 4.4: Number of rejections versus number of machines.

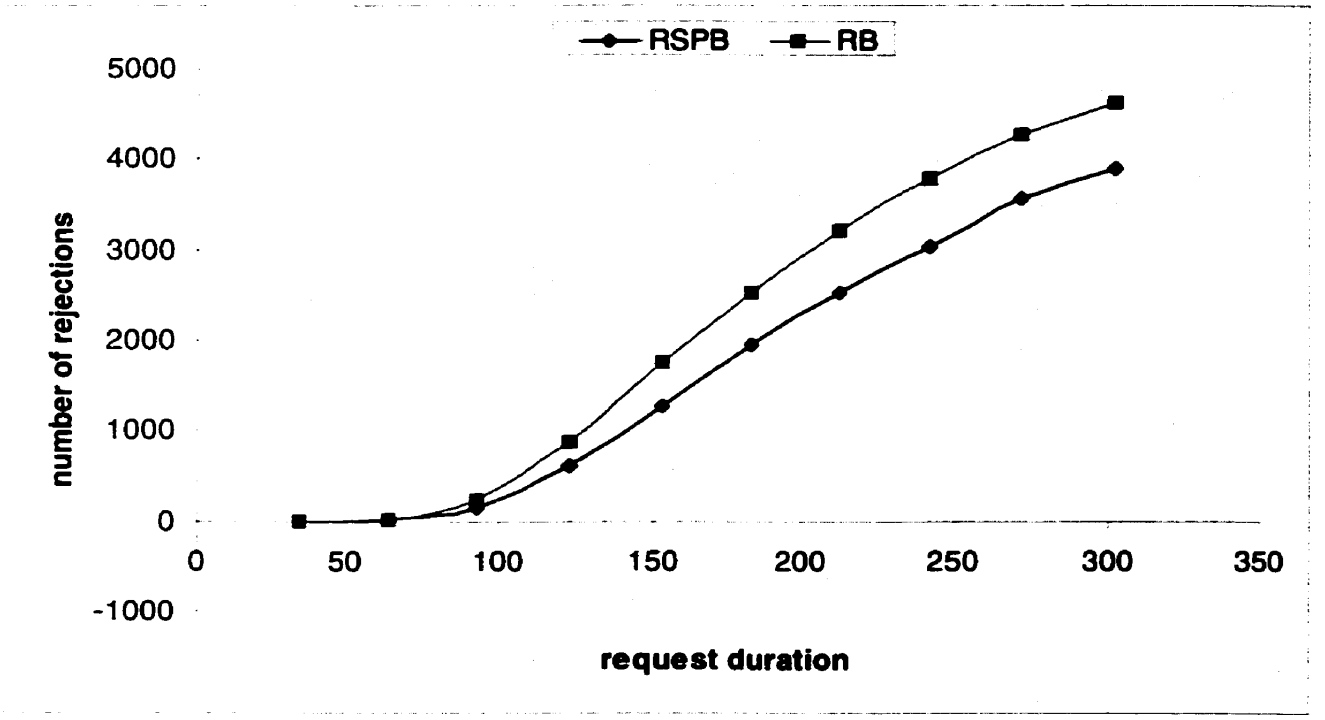


Figure 4.5: Number of rejections versus request duration.

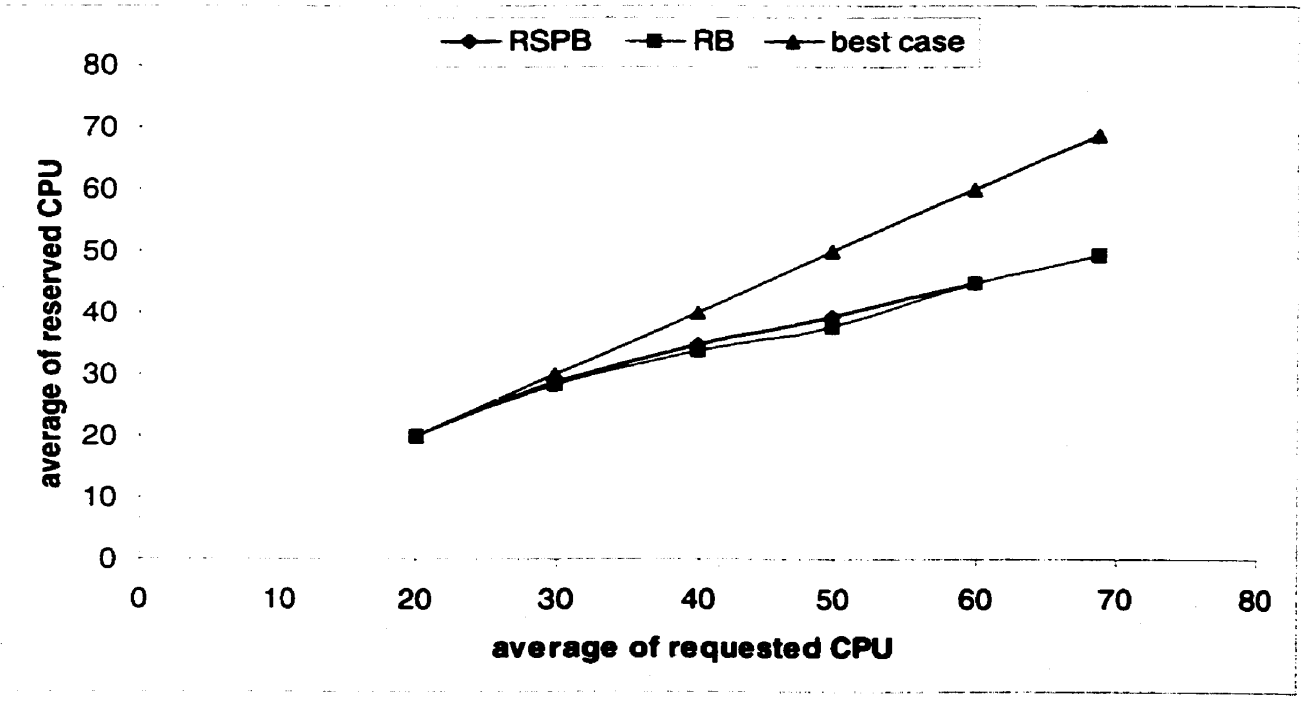


Figure 4.6: The average of reserved CPU versus the average of requested CPU.

Figure 4.6 shows the variation of the average of reserved CPU with the average of requested CPU. The average of requested CPU ranges from 20% to 70%. For contrast, we show the ideal case of satisfying each request to 100% as well.

In Figure 4.6, when the average of requested CPU is less than 25%, reserved CPU for all three curves are the same. This is because when the requested CPU is lower, the system can satisfy all requests. Therefore, reservations for all three cases are the same. When the average of requested CPU is greater than 25%, three approaches deviate. In particular, RB and RSPB deviate from the ideal approach.

The curve for RB is lowest before the average of requested CPU is less than 63%. The difference between RSPB and RB becomes bigger and bigger for the average of requested CPU less than 50%. However, this difference becomes smaller and smaller after 50% and almost the zero after 63%. The reason for this is when the average of requested CPU increases, the number of satisfied requests decrease. Therefore, the reserved CPU is decreased because of occurrence of rejections. However, considering benefit function in RSPB helps it to reduce the number of rejections, thus increasing the average percentage of reserved CPU. However, when the average requested CPU is greater than 50%, the advantage of this mechanism is not pronounced. After 63%, this advantage is almost none existent. This is because when the average of requested CPU is much higher, for example 60%, most machines often only have two possible states, either be reserved about 60% or idle. In our study, from four different benefit function shapes, the lowest CPU reservation that the user can be allocated and still gets acceptable benefit is *requested CPU *25%*. If in a certain time slot, let's assume all requests ask for 60% CPU, then, all machines in

the system are reserved by 60%, when a new request comes even with lowest acceptable CPU which is 60% * 25% (from selected benefit function shape), it will be rejected.

Figure 4.7 shows the result of sum of rejected priorities versus number of rejections. The simulation time ranges from 10,000 to 100,000 time units. It created about an average of 1,063 to 10,505 requests. Requested starting times were uniformly distributed over 4,320 time units. But, the difference from previous statement is that the starting time added an extra 300 time units in order to avoid too many times such that only one request in metaRequest is scheduled. Therefore, the advantage of ordering request by priority is more obvious.

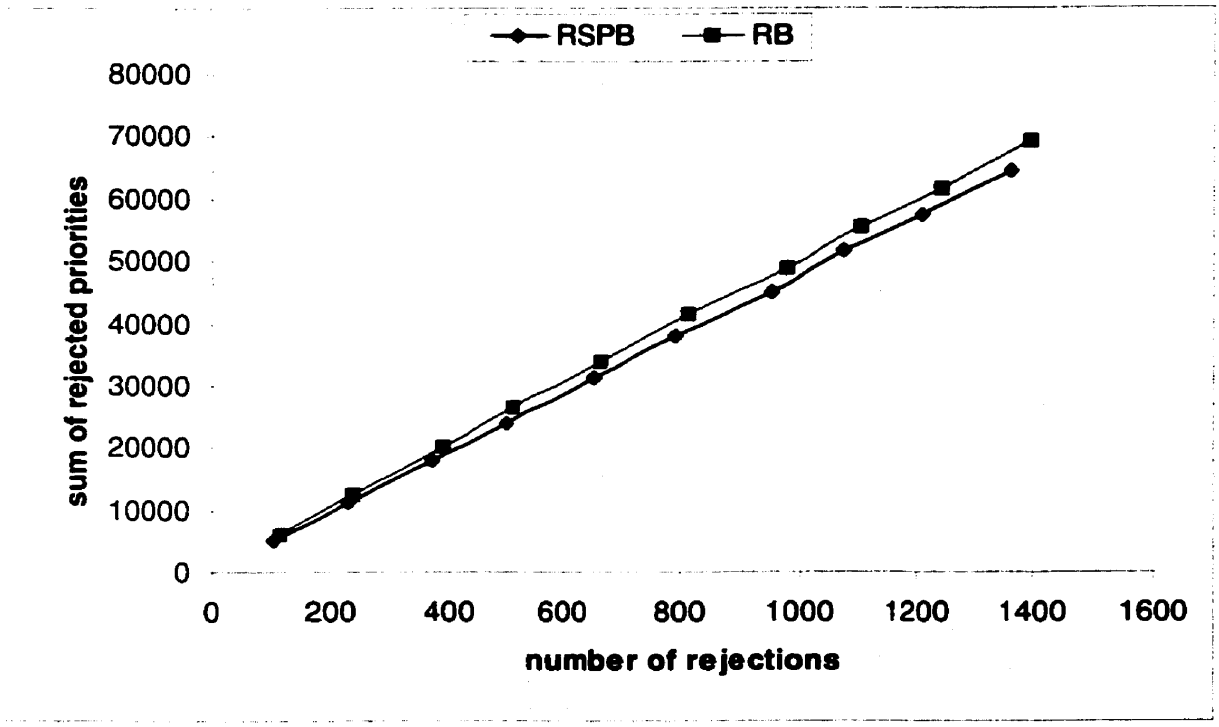


Figure 4.7: Sum of rejected priorities versus number of rejections.

From Figure 4.7, we can see the sum of rejected priorities of RSPB is less than that of RB. This is due to priority ordering in RSPB. In RSPB, requests with higher priorities are always scheduled prior to those with lower priority. This ensures when resources are scarce, requests with higher priority have more chance of being admitted.

5 CO-RSPB: CO-RESERVATION SCHEDULER WITH PRIORITIES AND BENEFIT FUNCTIONS

5.1 Preliminary Remarks

The difference between Co-Reservation Scheduler with Priorities and Benefit Functions (Co-RSPB) and RSPB is that Co-RSPB schedules reservations on multiple resources simultaneously, whereas RSPB schedules reservations on single resource. In some Grid systems such as collaborative computing Grid, a common characteristic of applications is a need to allocate multiple resources simultaneously. A challenge of co-allocation is, in a competition system, some required resources might not be available when demanded. Thus, the application cannot be executed with all required resources in desired time duration. To reduce the probability of resource unavailability while co-allocating resources, a co-reservation algorithm Co-RSPB is developed for a Grid system such as collaborative computing Grid. As with the RSPB described in the last chapter, Co-RSPB schedules co-reservation while considering relative priorities of various application requests. Benefit functions are used to associate a client's QoS requirements. Two comparison algorithms are also developed as a base line to compare the performance of Co-RSPB. Detailed description of these algorithms as well as simulation results and discussion are presented in this chapter. Because Co-RSPB is developed under the same assumptions as RSPB, the section for assumptions is skipped in this chapter.

5.2 Notations and Mathematical Model

Let m be the number of machines in the system. The machines are assumed to be homogeneous. Let CPU_sys_j represent the percentage of CPU of the j-th machine M_j that is dedicated to the Grid system. For each request R_k arriving at the reservation scheduler, following parameters are defined.

- $t_{k\text{-}start}$: start time of reservation for R_k;
- $t_{k\text{-}end}$: end time of reservation for R_k;
- R_k^i: the i-th sub-request of R_k;
- $CPU_{max\text{-}k}^{\ i}$: minimum CPU requirement of R_k^i for delivering the maximum benefit to the application;
- p_k: priority of R_k. When the overall demand exceeds the available resources, the objective of the reservation scheduler is to minimize the sum of priority of the requests that are rejected, therefore to maximize system benefit. This study uses heuristic approaches to achieve this objective.
- $B_k^i(p_{cpu})$: the benefit function associated with R_k^i. It gives the benefit R_k will receive if it is reserved CPU at the requested level. Figure 1.1 shows some of the shapes the benefit function could take for a reservation request.

In the following algorithm description, notations below are also used:

- $CPU_{min\text{-}k}^{\ i}$: minimum CPU requirement for R_k^i when R_k^i gets lowest acceptable benefit from reservation;
- M_j: the j-th machine in the system;
- $load\_heaviest_j(t_m, t_n)$: the heaviest load of M_j within a reservation duration t_m to t_n;
- $CPU\_smallest_j(t_m, t_n)$: the smallest availability of CPU to reserve for M_j within a duration t_m to t_n. $CPU\_smallest_j(t_m, t_n) = CPU\_sys_j - load\_heaviest_j(t_m, t_n)$;
- $B_{jk}^{\ i}$: benefit value which R_k^i can get from M_j;

- $R_{k\text{-sub-meta}}$: sub-request queue of R_k;
- isFloating: Boolean variable, true if the request R_k asks for floating machines; false if the request R_k asks for fixing machines;
- $machQ_{hard}$: a machine queue used to store machines which can satisfy hard QoS request of R_k^i;
- $machQ_{soft}$: a machine queue used to store machines which can satisfy soft QoS request of R_k^i.

In order to quantify the system performance in terms of service, a system benefit calculation model is also developed. Let $b_1, b_2, \ldots b_n$ be the benefit received by n sub-request $R_k^1, R_k^2, \ldots R_k^n$ of application R_k at the QoS level at which it reserved. The maximum value for $b_1, b_2, \ldots b_n$ is set to be 1. Let B_k be the benefit that the application R_k receives. B_k is define as:

$$B_k = 1/n \sum_i b_i$$

Thus, the maximum value for B_k is 1.

Let B be the benefit that the system provides. B is defined as:

$$B = \sum_k w_k B_k$$

Where w_k is the weight assigned to application R_k. This weight captures the importance of the application. In this study, it related to the application's priority. Let $p_1, p_2, \ldots p_m$ be the priority of applications $R_1, R_2, \ldots R_m$. then the weight w_k for the application R_k is defined as:

$$w_k = p_k / \sum_j p_j$$

Thus, the maximum value for B is 1. Therefore, the objective of the system is to achieve benefit value as close to 1 as possible.

5.3 Co-Reservation Scheduling with Priorities and Benefit Functions

This section examines the co-reservation scheduling algorithm. In this study, each co-reservation request involves in multiple resources. If any one required resource is not available to the application, the whole application will be rejected by the system. The outline of the dynamic co-reservation scheduler is the same as Figure 4.1. Therefore, in this section, the description of co-reservation scheduler is skipped and *scheduleR*$_{meta}$ function is presented directly.

Figure 5.1 shows the outline of *scheduleR*$_{meta}$ function. Line (2) to (4) sort the requests in R_{meta} in descending order by the priority of the requests. This ensures that if multiple reservation requests require reservation in the same duration, the requests with higher priority will be scheduled first. This reduces the sum of rejected priorities, thus ensuring the resources are used in the most beneficial manner. Line (5) to (12) assign sub-requests to suitable machines for each request. If the request requires floating machines, function *floatScheduling* is called. If the request requires fixing machines, function *fixScheduling* is called.

```
function scheduleRmeta (meta-task Rmeta)
(2)     for all requests Rk in meta-request Rmeta
(3)        sort the requests in descending order by pk;
(4)     endfor
(5)     for each sorted request Rk in Rmeta
(6)        get tk-start and tk-end ;
(7)        get all sub_request Rk^i of Rk, put Rk^i into Rk-sub-meta;
(8)        if(isFloating == true)
(9)           floatScheduling(Rk-sub-meta, tk-start, tk-end);
(10)       else
(11)          fixScheduling(Rk-sub-meta, tk-start, tk-end);
(12)    endfor
```

Figure 5.1: Outline of Co-RSPB scheduling.

Figure 5.2 shows the pseudo code for the *floatScheduling* function for Co-RSPB. Line (3) determines the minimum CPU requirement for each sub-request R_k^i of R_k (i.e., CPU reservation at which R_k^i can provide the lowest acceptable benefit to the application according to the selected benefit function shape). These values will be used to determine if the request should be admitted with graceful degradation of some or all sub-requests, or rejected when resources are scarce and hard QoS requirement of some or all sub-requests cannot be guaranteed. Line (4) sorts the sub-requests of R_k in $R_{k\text{-}sub\text{-}meta}$ in descending order by minimum CPU requirement of R_k^i. There are two purposes for this sorting. First, for sub-requests with a higher CPU requirement, there is less possibility to find a desired machine. Therefore, scheduling this kind of sub-requests first can increase the possibilities, thus, reduce the chance of rejection for overall requests. The second, because of a lower possibility to find a desired machine for sub-requests with a higher CPU requirement, scheduling it first can make the scheduling procedure faster. This is because if a sub-request with a higher CPU requirement is rejected by the system, the overall request is rejected. Thus, it is no longer necessary to schedule other sub-requests with lower CPU requirements.

In this reservation model, a CPU resource may be temporally shared by multiple reservations, that is, multiple reservations may overlap in time. Therefore, we need to determine the current CPU usage for a given time interval before admitting a reservation. In Line (9), the current CPU (machine) usage is determined using *time slot* tables that keep track of reservations that have already been accepted. The time slot tables only keep track of the partition of the CPU that is dedicated to the Grid and thus, managed by the reservation scheduler.

Line (10) determines whether the examined machine can satisfy the hard QoS requirement of sub-request R_k^i. If yes, this machine is put into a machine queue named $machQ_{hard}$. The size of $machQ_{hard}$ is set to be 1. Therefore, only one machine can be put into $machQ_{hard}$ for each sub-request. Line (11) to line (15) is used to select a machine that has smallest CPU availability to satisfy the hard QoS requirement of sub-request R_k^i. Although it is not shown in the pseudo-code in Figure 5.2, the search for a machine that satisfies the reservation sub-request can be stopped when a machine that satisfies the hard QoS is found. If none of the machines in the system can satisfy the hard QoS requirement of R_k^i, the co-reservation scheduler will attempt to schedule the reservation with a degradation of the CPU requirement for R_k^i if the benefit function is provided to allow for the degradation. Line (17) to (25) attempt to find a machine that satisfies this situation and this machine is considered to satisfy the soft QoS requirement of R_k^i. Line (19) to (25) attempt to maximize the benefit delivered to the application by the reservation.

Lines (27) and (30) deal with the case where the system cannot provide the requested level of service to R_k^i. In this case, the overall request is rejected. All reservations for sub-requests, which have been scheduled before R_k^i, should be canceled. Sub-requests that don't have a chance to be scheduled are no longer necessary to be considered. The co-reservation scheduler checks the meta-request $R_{nonsatisfy}$ and sends rejection messages to the clients whose submitted reservation requests are in $R_{nonsatisfy}$. The clients may resubmit their reservation requests with modifications and these submissions will be considered for reservation at the next scheduling event.

```
function floatScheduling (meta-request R_k-sub-meta, start-time t_k-start, end-time t_k-end)
(2)  for all sub_request R_k^i in R_k-sub-meta
(3)     calculate CPU_min-k^i according to selected benefit function shape;
(4)     sort sub-requests in descending order by CPU_min-k^i;
(5)  endfor
(6)  for each sub_requsest R_k^i in R_k-sub-meta
(7)     reset machQ_hard and machQ_soft
(8)     for each machine M_j in the system
(9)        get CPU_smallest_j(t_k-start, t_k-end);
(10)       if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_max-k^i)
(11)          if (machQ_hard contains another machine M_r)
(12)             if (CPU_smallest_j(t_k-start, t_k-end) < CPU_smallest_r(t_k-start, t_k-end)
(13)                machQ_hard <- M_j;
(14)          else
(15)                machQ_hard <- M_j;
(16)       else
(17)          if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_min-k^i)
(18)             if (machQ_hard is empty)
(19)                if (machQ_soft contains another machine M_r);
(20)                   B_jk^i = B_k^i(CPU_smallest_j(t_k-start, t_k-end));
(21)                   B_rk^i = B_k^i(CPU_smallest_r(t_k-start, t_k-end));
(22)                   if (B_jk^i > B_rk^i)
(23)                      machQ_soft <- M_j;
(24)                else
(25)                      machQ_soft <- M_j;
(26)     endfor
(27)     if (both the machQ_hard and machQ_soft are empty)
(28)        put the request R_k into meta-request R_nonsatisfy;
(29)        cancel reservation for R_k^r of which the order is bigger than R_k^i;
(30)        break;
(31)     else
(32)        if (machQ_hard is not empty)
(33)           the machine in machQ_hard is the choice;
(34)           reserved CPU = CPU_max-k^i;
(35)        else
(36)           the machine in machQ_soft is the choice;
(37)           reserved CPU = CPU_smallest_j(t_k-start, t_k-end);
(38)        mark the time slot table for this machine with reserved CPU;
(39) endfor
```

Figure 5.2: Function floatScheduling for Co-RSPB.

Figure 5.3 shows pseudo code for the *fixScheduling* function. Since all sub-requests have fixed machine requirements, the algorithm here is much simpler than the previous one. The sorting

procedure in Figure 5.2 is not necessary here. Further more, for each sub-request, only the desired machine need to be examined. Therefore, line (4) is used instead of a loop of number of machines in Figure 5.2. Once the desired machine is examined, the decision that if this sub-request can be satisfied is made immediately. The time slot table is also marked right after the examination.

```
function fixScheduling (meta-request R_k-sub-meta, start-time t_k-start, end-time t_k-end)
(2)   for all sub_request R_k^i in R_k-sub-meta
(3)     calculate CPU_min-k^i according to selected benefit function shape;
(4)     get request machine M_j;
(5)     get CPU_smallest_j(t_k-start, t_k-end);
(6)     if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_max-k^i);
(7)       mark the time slot table for this machine with requested reservation;
(8)     else
(9)       if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_min-k^i)
(10)        reserved CPU = CPU_smallest_j(t_k-start, t_k-end);
(11)        mark the time slot table for this machine with reserved CPU;
(12)      else
(13)        put the request R_k into meta-request R_nonsatisfy;
(14)        cancel reservation for R_k^r of which the order is bigger than R_k^i;
(15)        break;
(16)  endfor
```

Figure 5.3: Function fixScheduling for Co-RSPB.

5.4 Comparison Algorithms

To our knowledge, there is no similar algorithm available to compare with the performance of Co-RSPB. Actually, the problem that Co-RSPB solved is much like a *bin packing* problem [Wei95] if:

- We consider only the number of rejections from the system rather than the benefit that the system can provide;
- All the requests have a hard QoS requirement rather than some requests have a soft QoS requirement;

- For each scheduling interval, all machines have 100% CPU available to applications in R_{meta}.

Based on above observation, we designed a *Co-reservation Scheduler with Best Fit scheme* (Co-RSBF) as a comparison algorithm to Co-RSPB. Co-RSBF is similar as *Best Fit Decreasing* (BFD) algorithm in *bin packing* problem except the bin size (i.e. machine CPU availability in this study) is not always 1. Co-RSBF focuses on the number of rejections from the system rather than the benefit that the system can provide. Therefore, to reduce the number of rejections, all applications are given the minimum benefit by giving all sub-requests the minimum CPU reservations. As BFD in *bin packing*, Co-RSBF should give a result that is extremely close to lower bound of rejections.

Figure 5.4 shows the outline of Co-RSBF. Instead of sorting applications by priority, Co-RSBF sorts applications by the sum of sub-request's minimum CPU requirements as in BFD.

```
function scheduleRmeta (meta-task Rmeta)
(2)  for all requests Rk in meta-request Rmeta
(3)     sort the requests in descending order by Σi CPUmin-k^i;
(4)  endfor
(5)  for each sorted request Rk in Rmeta
(6)     get tk-start and tk-end ;
(7)     get all sub_request Rk^i of Rk and put Rk^i into Rk-sub-meta;
(8)     if(isFloating == true)
(9)        floatScheduling(Rk-sub-meta, tk-start, tk-end);
(10)    else
(11)       fixScheduling(Rk-sub-meta, tk-start, tk-end);
(12)endfor
```

Figure 5.4: Outline of Co-RSBF scheduling.

Figure 5.5 shows the *floatScheduling* function in Co-RSBF and Figure 5.6 shows the *fixScheduling* function for Co-RSBF.

```
function floatScheduling (meta-request R_k-sub-meta, start-time t_k-start, end-time t_k-end)
(2)  for all sub_request R_k^i in R_k-sub-meta;
(3)    calculate CPU_min-k^i according to selected benefit function shape;
(4)    sort sub-requests in descending order by CPU_min-k^i;
(5)  endfor
(6)  for each subresquest R_k^i
(7)    reset machQ_soft;
(8)   for each machine M_i in the system
(9)     get CPU_smallest_i(t_k-start, t_k-end);
(10)    if (CPU_smallest_i(t_k-start, t_k-end ≥ CPU_min-k^i);
(11)      if (machQ contains machine M_r)
(12)        if (CPU_smallest_i(t_k-start, t_k-end) < CPU_smallest_r(t_k-start, t_k-end))
(13)          machQ_soft <- M_i;
(14)      else
(15)        machQ_soft <- M_i;
(16) endfor
(17) if (machQ_soft is empty)
(18)   put the request R_k into meta-request R_nonsatisfy;
(19)   cancel reservation for R_k^r of which the order is bigger than R_k^i;
(20)   break;
(21) else
(22)   the machine in machQ_soft is the choice;
(23) mark the time slot table for selected machine with reserved CPU = CPU_min-k^i;
(24) endfor
```

Figure 5.5: Function floatScheduling for Co-RSBF.

```
function fixScheduling (meta-request R_k-sub-meta, start-time t_k-start, end-time t_k-end)
(2)  for all sub_request R_k^i in R_k-sub-meta
(3)    calculate CPU_min-k^i according to selected benefit function shape;
(4)    get request machine M_j;
(5)    get CPU_smallest_j(t_k-start, t_k-end);
(6)    if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_min-k^i)
(7)      reserved CPU = CPU_min-k^i;
(8)      mark the time slot table for this machine with reserved CPU;
(9)    else
(10)     put the request R_k into meta-request R_nonsatisfy;
(11)     cancel reservation for R_k^r of which the order is bigger than R_k^i;
(12)     break;
(13) endfor
```

Figure 5.6: Function fixScheduling for Co-RSBF.

Although Co-RSBF can give results that are extremely close to the lower bound for rejections, the benefit that the system can provide in Co-RSBF is minimum one because each sub-request is given a minimum benefit. According to the system benefit calculation model, the benefit $B_k = 1/n \sum_i b_i$ received by the user of the application is minimum. Therefore, the benefit $B=\sum_k w_k B_k$ that the system can provide is minimum. When the resources in a system are not extremely scarce, this result should not be expected. Therefore, we designed another comparison algorithm called *Co-Reservation Scheduler with Best Fit and Refining scheme* (Co-RSBFR). Co-RSBFR uses the same *floatScheduling* and *fixScheduling* functions as Co-RSBF to schedule requests in each scheduling interval in order to get lower rejections in each scheduling interval. Then, without increasing the number of rejections, Co-RSBFR tries to extend CPU reservation for each sub-request, thus extends system benefit within each scheduling interval. Figure 5.7 shows the outline of Co-RSBFR algorithm.

```
function scheduleR_meta (meta-task R_meta)
(2)  for all requests R_k in meta-request R_meta
(3)     sort the requests in descending order by Σ_i CPU_min-k^i;
(4)  endfor
(5)  for each sorted request R_k in R_meta
(6)     get t_k-start and t_k-end ;
(7)     get all sub_request R_k^i of R_k, put R_k^i into R_k-sub-meta;
(8)     if(isFloating == true)
(9)        floatScheduling(R_k-sub-meta, t_k-start, t_k-end);
(10)    else
(11)       fixScheduling(R_k-sub-meta, t_k-start, t_k-end);
(12)endfor
(13)for each sorted request R_k in meta-request R_meta
(14)   if R_k is accepted
(15)      get t_k-start and t_k-end ;
(16)      get all sub_request R_k^i of R_k, put R_k^i into R_k-sub-meta;
(17)      refineScheduling(R_k-sub-meta, t_k-start, t_k-end);
(18)endfor
```

Figure 5.7: Outline of Co-RSBFR scheduling.

```
Function refineScheduling(meta-task R_k-sub-meta, start-time t_k-start, end-time t_k-end)
(2)  for each sub-request R_k^i in R_k-sub-meta
(3)    if (it has soft QoS requirement)
(4)      get machine M_s on which R_k^i is assigned;
(5)      get CPU_ smallest_s(t_k-start, t_k-end);
(6)      CPU_smallest_s(t_k-start, t_k-end) = CPU_min-k^i + CPU_smallest_s(t_k-start, t_k-end);
(7)      if (CPU_smallest_s(t_k-start, t_k-end) ≥ CPU_max-k^i)
(8)        machQ_hard <- M_s;
(9)      else   machQ_soft <- M_s;
(10)     for each machine M_j
(11)       if (j != s)
(12)         get CPU_smallest_j(t_k-start, t_k-end);
(13)         if (CPU_smallest_j(t_k-start, t_k-end) ≥ CPU_max-k^i)
(14)           if (machQ_hard contains machine M_r)
(15)             if (CPU_smallest_j(t_k-start, t_k-end) < CPU_smallest_r(t_k-start, t_k-end))
(16)               machQ_hard <- M_j;
(17)           else   machQ_hard <- M_j;
(18)         else
(19)           if (machQ_hard is empty)
(20)             if (machQ_soft contains machine M_r)
(21)               B_jk^i = B_k^i(CPU_smallest_j(t_k-start, t_k-end));
(22)               B_rk^i = B_k^i(CPU_smallest_r(t_k-start, t_k-end));
(23)               if (B_jk^i > B_rk^i)   machQ_soft <- M_j;
(24)             else   machQ_soft <- M_j;
(25)     endfor
(26)     if (machQ_hard is not empty)
(27)       machine M_j in machQ_hard is the choice;
(28)       reserved CPU = CPU_max-k^i;
(29)     else
(30)       machine M_j in machQ_soft is the choice;
(31)       reserved CPU  = CPU_smallest_j(t_k-start, t_k-end);
(32)   mark time slot table for M_j with reserved CPU; cancel reservation on M_s;
(33) endfor
(34) for each sub-request R_k^i
(35)   if (reserved CPU < CPU_max-k^i)
(36)   get the machine M_s on which R_k^i is assigned;
(37)   get CPU_smallest_s(t_k-start, t_k-end);
(38)   CPU_smallest_s(t_k-start, t_k-end) = CPU_rsv_k^i + CPU_smallest_s(t_k-start, t_k-end);
(39)   if (CPU_smallest_s(t_k-start, t_k-end) ≥ CPU_max-k^i)
(40)     on M_s, cancel original reservation; mark time slot table with CPU_max-k^i;
(41)   else
(42)     on M_s, cancel original reservation;
(43)     mark time slot table with CPU_smallest_j(t_k-start, t_k-end);
(44) endfor
```

Figure 5.8: Function refineScheduling for Co-RSBFR.

Figure 5.8 shows the *refineScheduling* function for Co-RSBFR. Line (3) gives the condition that only sub-requests with soft QoS requirements need to be refined. The benefit for sub-requests with hard QoS requirements are 1, already. When trying to extend the benefit for sub-request R_k^i, the reserved CPU for R_k^i should be counted while examining the CPU availability for the machine on which R_k^i is assigned in previous scheduling. Lines (4) to (9) work on this purpose. Lines (13) to (17) try to get the machine, which can satisfy the maximum QoS requirement of sub-request R_k^i and have the smallest CPU availability in order to leave machines with greater CPU availability to sub-requests with higher CPU reservation requirements. If there is no machine which can satisfy the maximum QoS requirement of sub-request R_k^i, then lines (20) to (24) try to get the machine which can give the greatest benefit to R_k^i. Note that there is at least one machine – that's the machine on which R_k^i is assigned previously, can satisfy at least the minimum CPU reservation request. After the first round refining, there may be the situation that after refining sub-request R_k^i, the reservation for other sub-request on the same machine as R_k^i is canceled, in this case, if the benefit value for R_k^i is not 1, it's reservation can be extended once more. Lines (34) to (45) work on this purpose.

5.5 Complexity Analysis

The running time of an algorithm is generally the most important aspect of concern. In this section, the complexity of running time for three algorithms is analyzed and compared.

It can be noted that there are four function calls inside three algorithms, which was not shown in previous sections. They are:

- Sorting function, which sorts requests or sub-requests by different criticality;
- Querying function, which queries machine utilization information within a specific duration from a *time slot table*;
- Marking function, which marks new reservations on a *time slot table* for a specific machine within a specific duration;
- Canceling function, which cancels reservation from a *time slot table* for a specific machine within a specific duration.

It is obvious that these four functions cannot be completed within one basic operation. The determination of running time upper bound for these four functions is dependent on what kind of algorithm or data structure these functions use. It can be noted that for the purpose of comparing the running time of Co-RSPB and Co-RSBF, the values of running time for these four functions will not affect the comparison result. Therefore, let S_i, T_q, T_m, and T_c is the running time for function sorting, querying, marking and canceling, respectively. Where i is the number of items, which will be sorted. Thus, the running time upper bound for both Co-RSPB and Co-RSBF is $O(S_N + NS_K + NMKT_q + NK^2T_c)$ if $(KT_c > T_m)$ or $O(S_N + NS_K + NMKT_q + NKT_m)$ if $(KT_c < T_m)$, where N is the number of requests in R_{meta}, M is number of machines participated in scheduling and K is the number of sub-requests in $R_{k\text{-}sub\text{-}meta}$.

From Figure 5.4, 5.7 and 5.8, it can be noted that the running time upper bound for Co-RSBFR is the running time upper bound for Co-RSBF plus the running time upper bound for the refining procedure, that is $O(S_N + NS_K + NMKT_q + NK^2T_c + NKT_m)$.

From the above analysis, we know that Co-RSPB and Co-RSBF have the same running time upper bound. However, the running time upper bound of Co-RSBFR is higher.

5.6 Simulation Results and Discussion

This section presents some results from a simulation study designed to evaluate the performance of the algorithm provided in the previous sections. In the simulation study we compared the Co-RSPB with Co-RSBF as well as Co-RSBFR. A discrete event simulator was written using the PARSEC language [BaM98] for comparing the three reservation schemes. In the simulations, the reservation requests arrived randomly according to a Poisson arrival process. With each request, several attributes were associated to define parameters like reservation start time, end time, sub-requests, percentage of resource, shape of the benefit function, priority, etc. The benefit functions were restricted to the four shapes in Figure 1.1 such that each function is used by 25% of total requests. Time slot table for each machine is maintained by a modified version of the data structure called Interval Skip List [HaJ96]. The following parameters are true for the simulation results presented unless stated otherwise.

- 10 machines participated in the simulation;
- Machine CPU dedication to the Grid system was uniformly distributed in [50%, 100%];
- It created 300 requests for each simulation. Each request consist of 1 to 6 sub-request;
- Each reservation (for each sub-request) requested for CPU usage was uniformly distributed in [10%, 90%];
- Requested duration was uniformly distributed in 20-180 time units (PARSEC clocktype);
- Requested starting times were uniformly distributed over 4,320 time units;
- The priority of request was uniformly distributed in [1, 100];
- 80% percent of requests required floating machines;
- Scheduling interval was 50 time units;

- Time was slotted with a granularity of one time unit.

Figure 5.9(a) shows the variation of system benefit with the number of requests. Figure 5.9(b) shows the variation of the number of rejections with the number of requests. The simulation created requests which ranged from 100 to 1,900. Figure 5.10(a) shows the variation of system benefit with the number of machines. Figure 5.10(b) shows the variation of the number of rejections with the number of machines. The number of machines participated in this experiment ranged from 4 to 40. Figure 5.11(a) shows the variation of system benefit with the average of duration. Figure 5.11(b) shows the variation of the number of rejections with the average of duration. The average requested duration varied in the [24, 159] ranges.

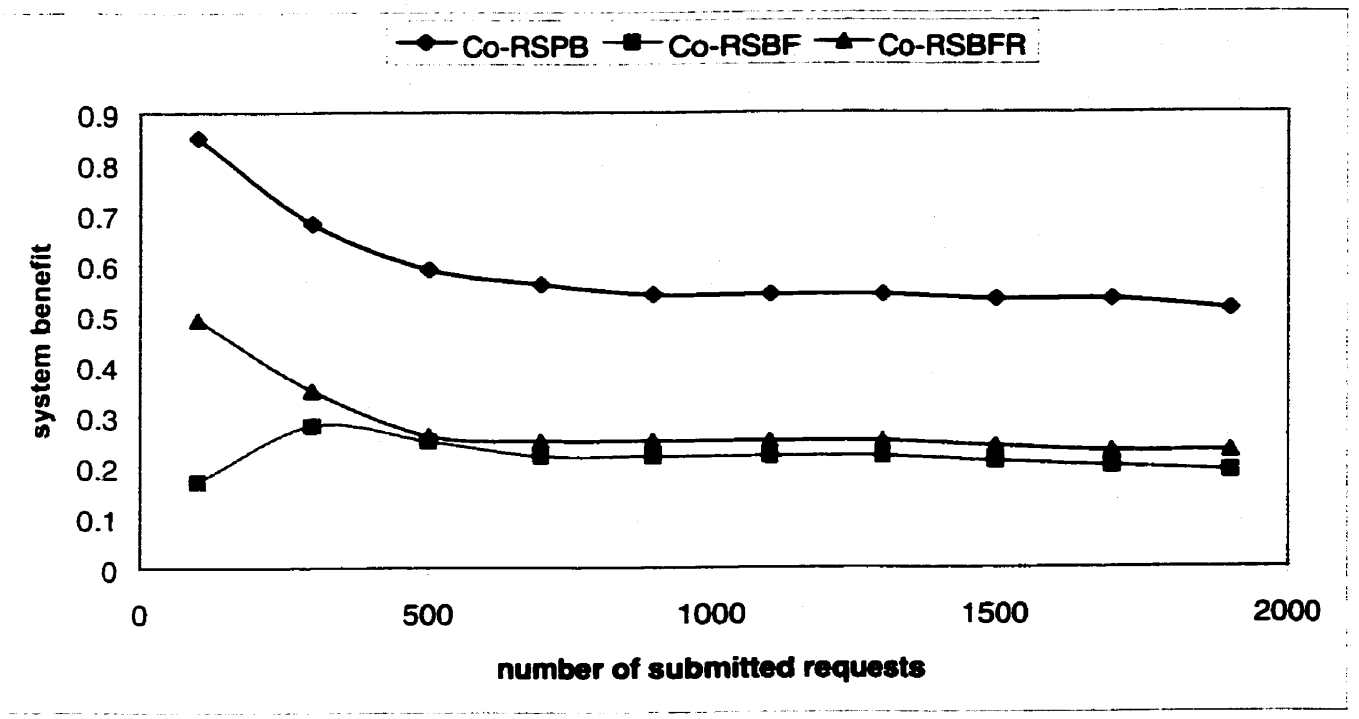
From Figures 5.9(a), 5.10(a), and 5.11(a), it can be noted that the system benefit for Co-RSPB is considerably higher than that for Co-RSBF and Co-RSBFR in all three cases. It's easy to understand that the system benefit for Co-RSBF is always lower because each application is given the minimum benefit in each scheduling interval even though the resource is enough to give more benefit to applications. Two reasons may result in why Co-RSBFR gives lower system benefit than Co-RSPB does. One is: even though Co-RSBFR can give a less a number of rejections, from system benefit calculation model described in previous section, the priority of application is an important factor for system benefit. Without considering priority in Co-RSBFR scheduling leads to lesser number of rejections but a lower system benefit. The second reason is Co-RSBFR may have the same number of rejections as Co-RSBF in one scheduling interval, but the refining procedure makes more resource unavailable for later scheduling than Co-RSBF does. This may lead to higher number of rejections occurring for the overall scheduling

procedure, therefore resulting in a lower system benefit. From Figure 5.9(b), 5.10(b) and 5.11(b), this assumption is approved.

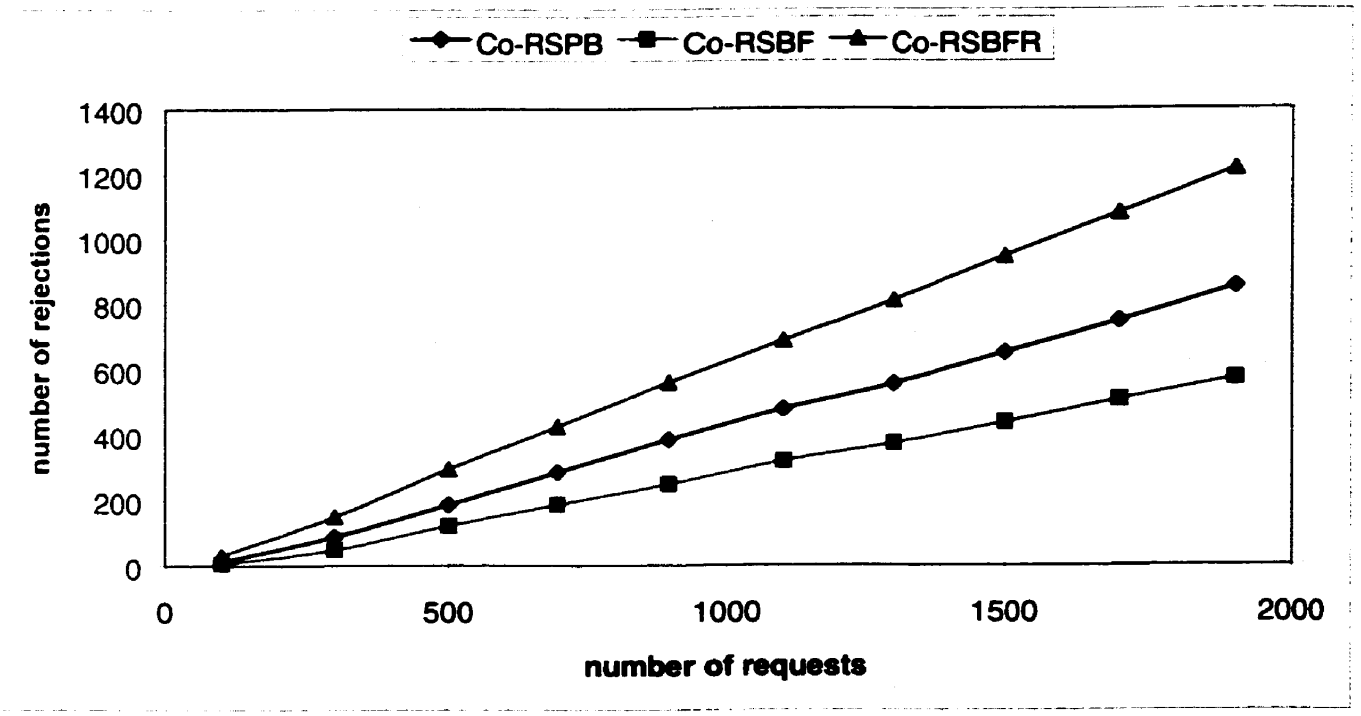
Another fact is that when resources are not scarce the system benefit for Co-RSBFR is considerably higher than that for Co-RSBF. However, when resources are extremely scarce, the system benefit for Co-RSBFR is very close to the benefit for Co-RSBF. This is because when resources are extremely scarce, after Co-RSBF scheduling, there will be very limited resources available for the refining procedure. Therefore, the refining procedure cannot make a significant difference to overall system benefit.

From Figure 5.9(b), 5.10(b) and 5.11(b), we can note that as we analyzed previously, the number of rejections for Co-RSBFR is much higher than that for Co-RSPB and Co-RSBF. The number of rejections for Co-RSBF is the lowest as we expected. The number of rejections for Co-RSPB is about 10% higher than that for Co-RSBF and this difference increases slightly when resource becomes scarcer.

Figure 5.12(a) shows the variation of system benefit with an increase in the percentage of floating requests. Figure 5.12(b) shows the variation of the number of rejections with percentage of floating requests. The percentage of requests asking for floating machines ranged from 0 to 100%.

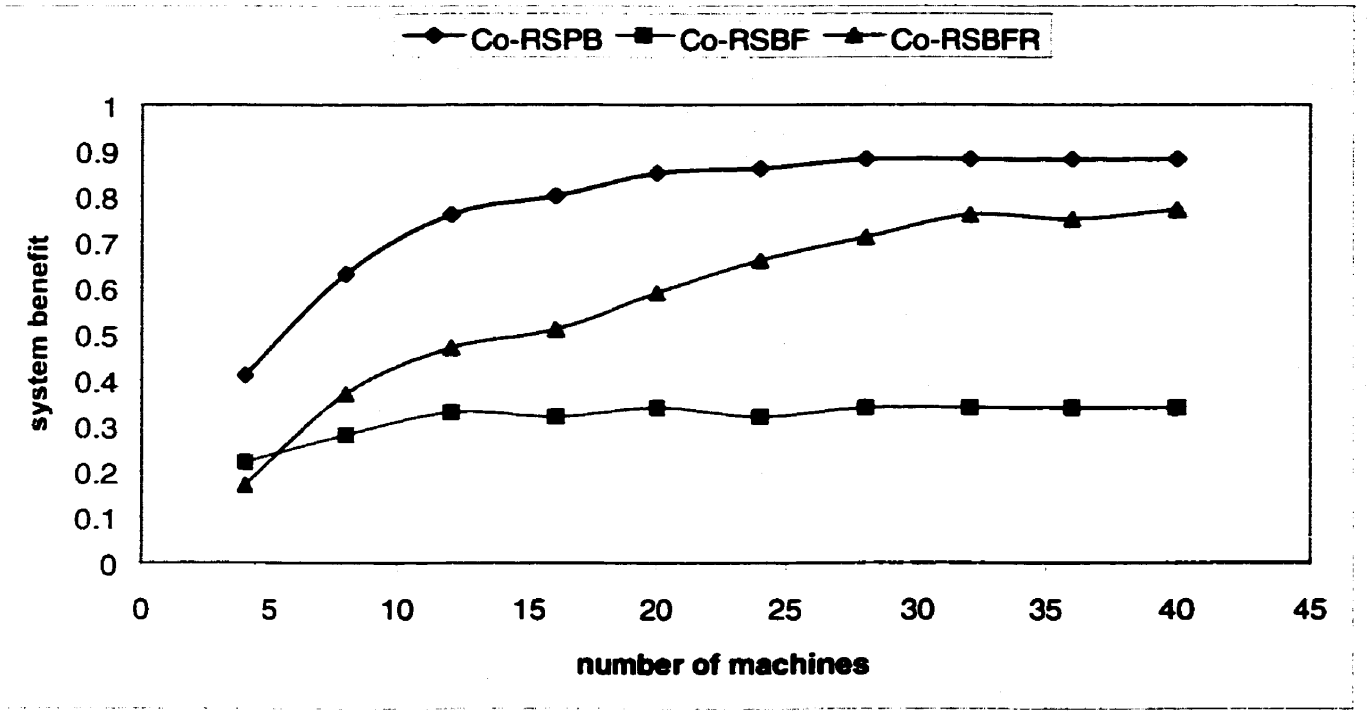


(a)

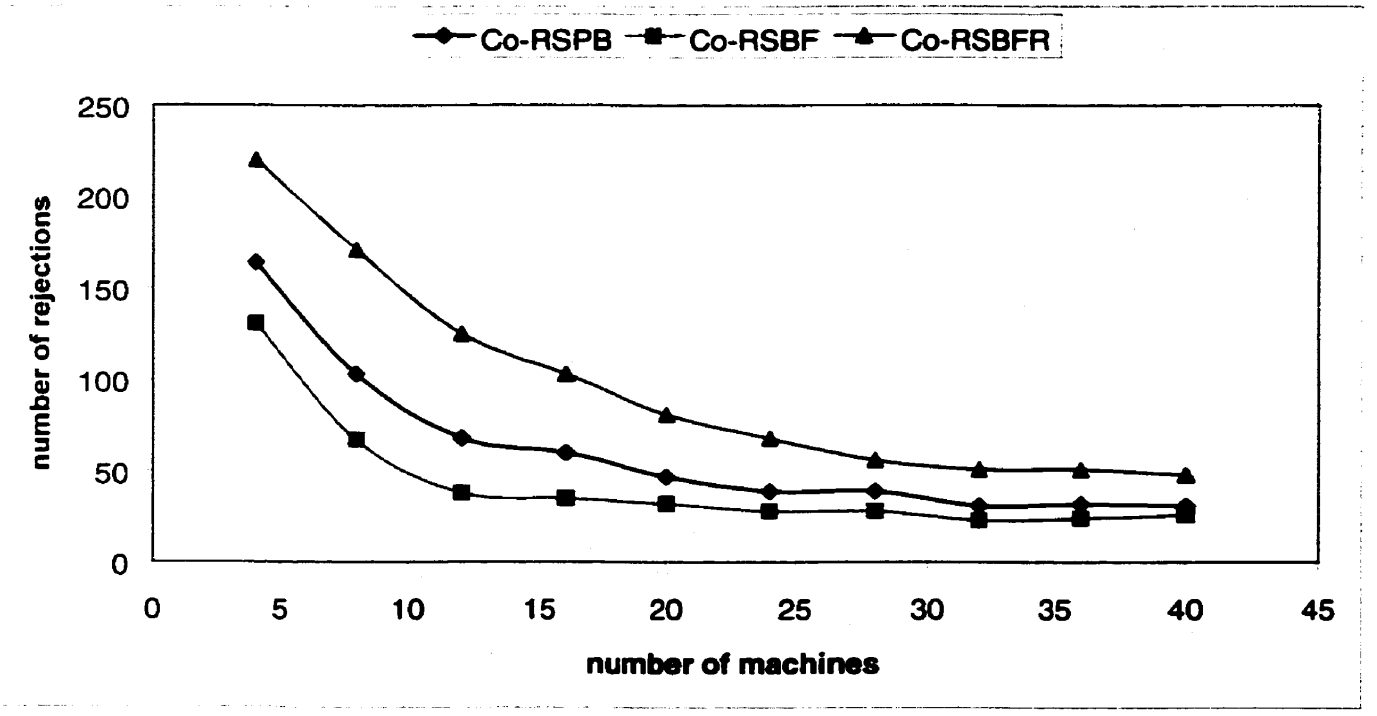


(b)

Figure 5.9: (a) System benefit and (b) number of rejections versus number of requests.

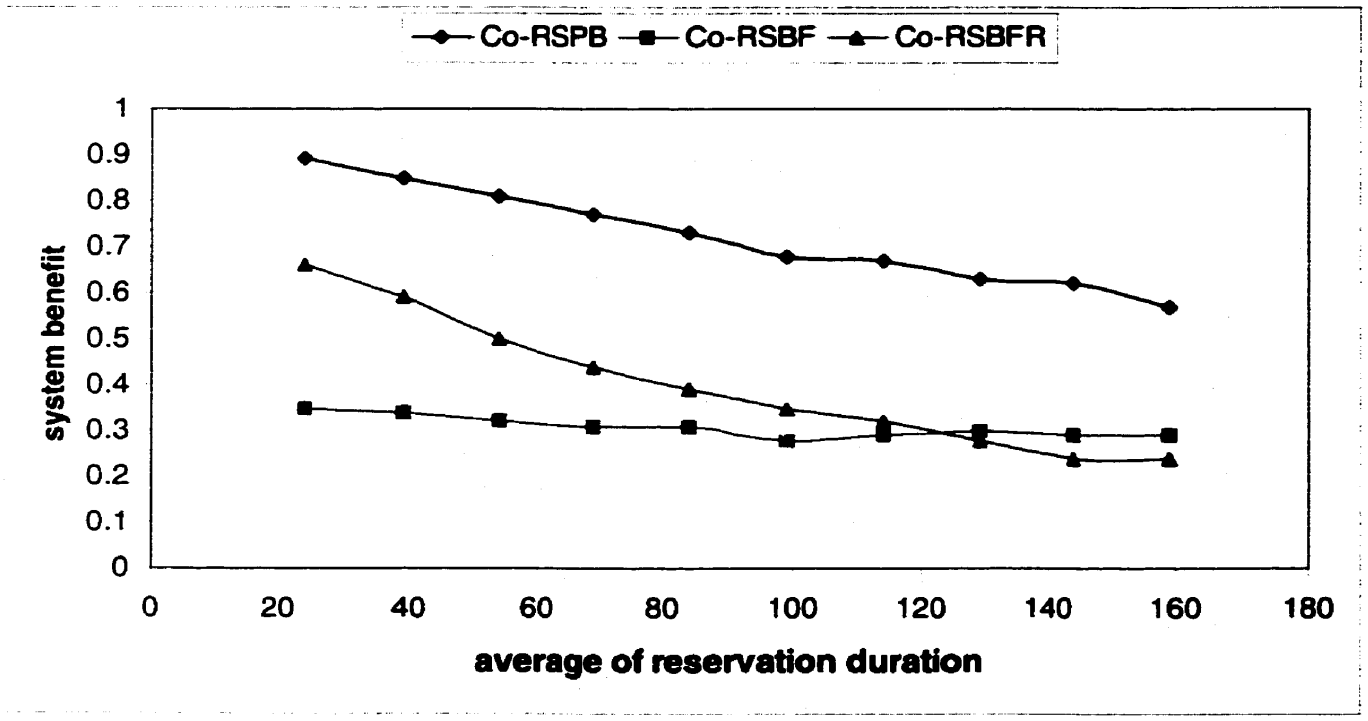


(a)

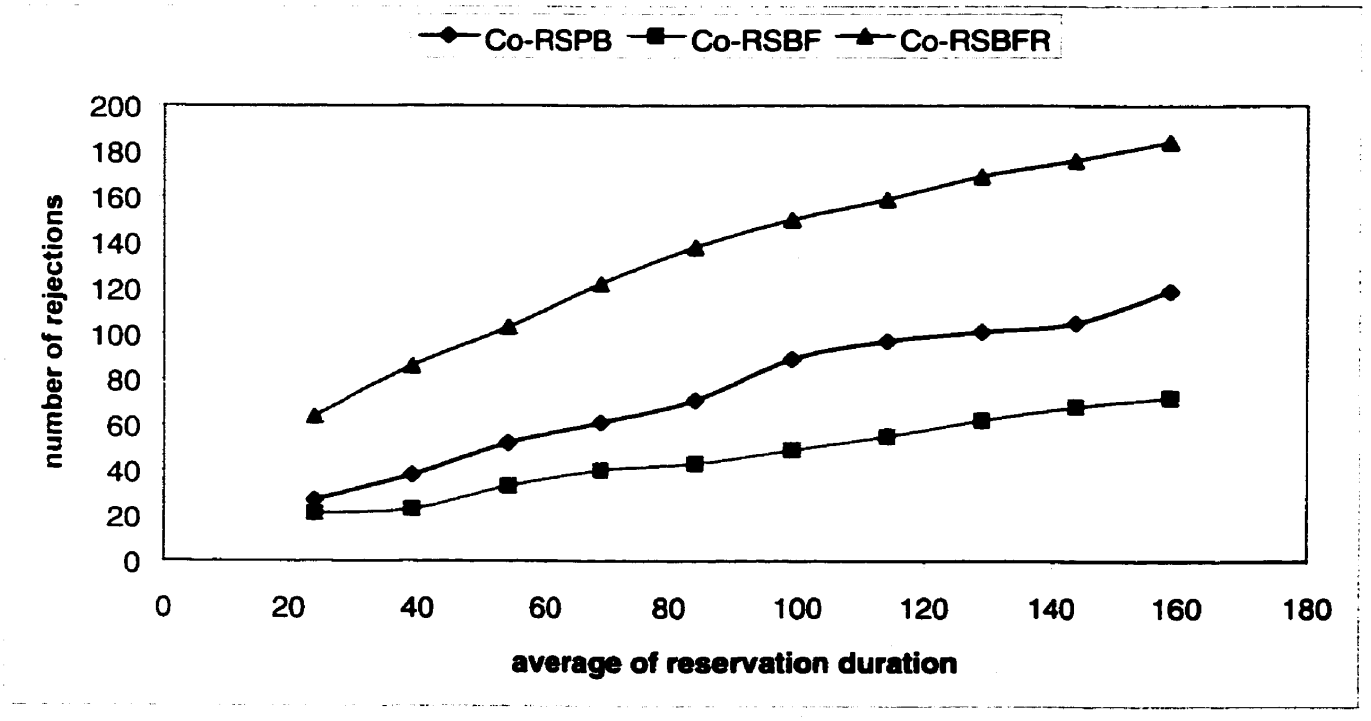


(b)

Figure 5.10: (a)System benefit (b)number of rejections versus number of machines.

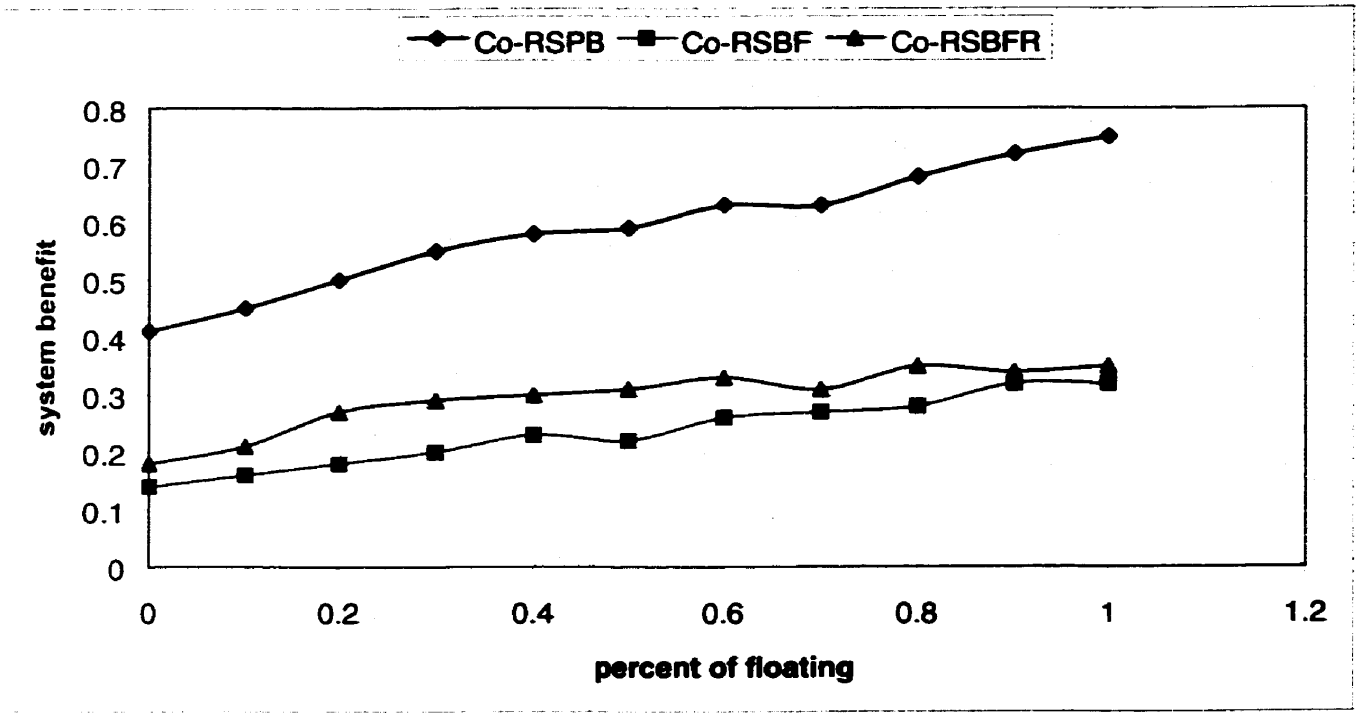


(a)

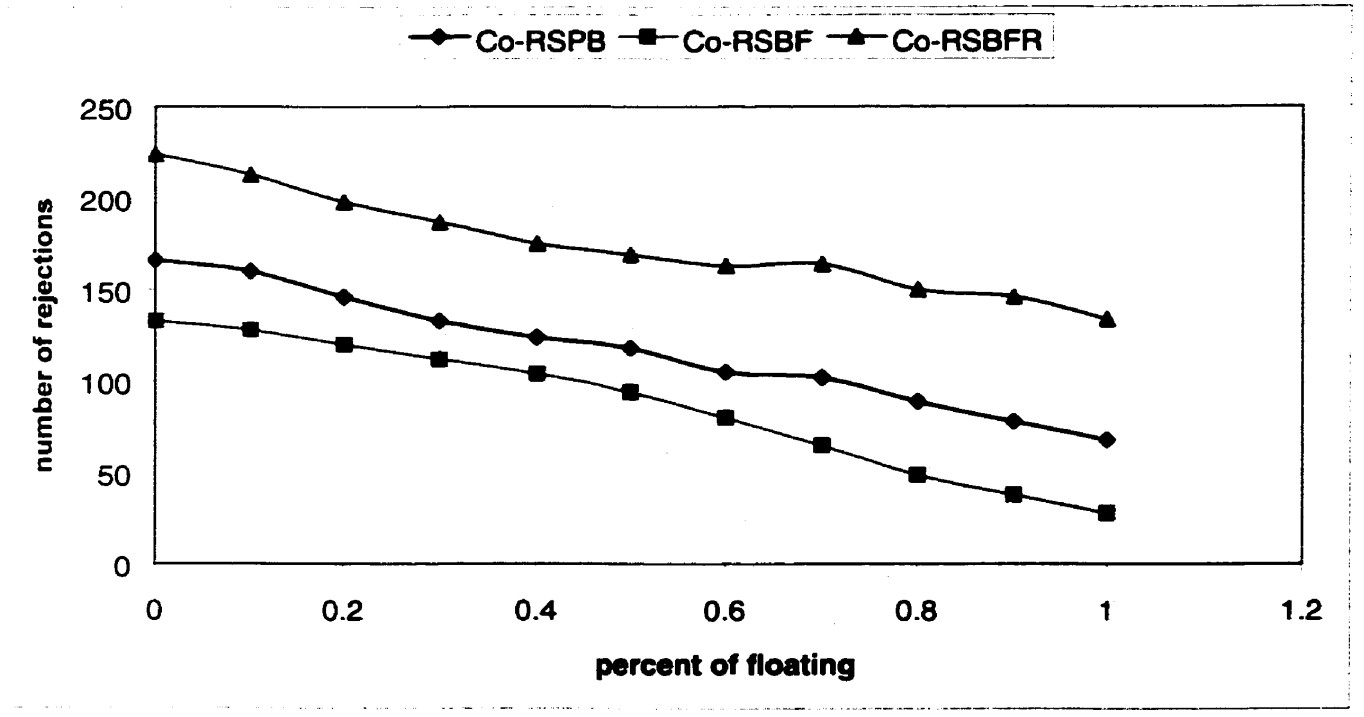


(b)

Figure 5.11: (a) System benefit (b) number of rejections versus requested duration.

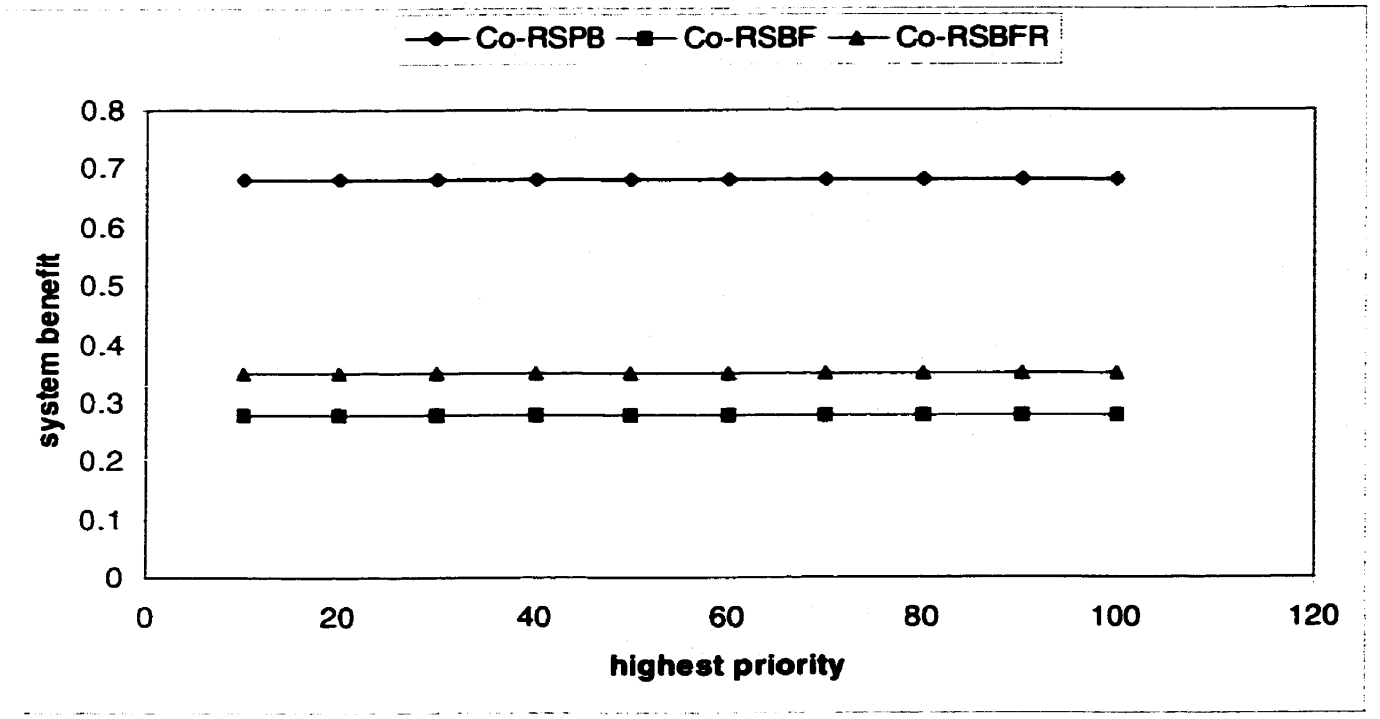


(a)

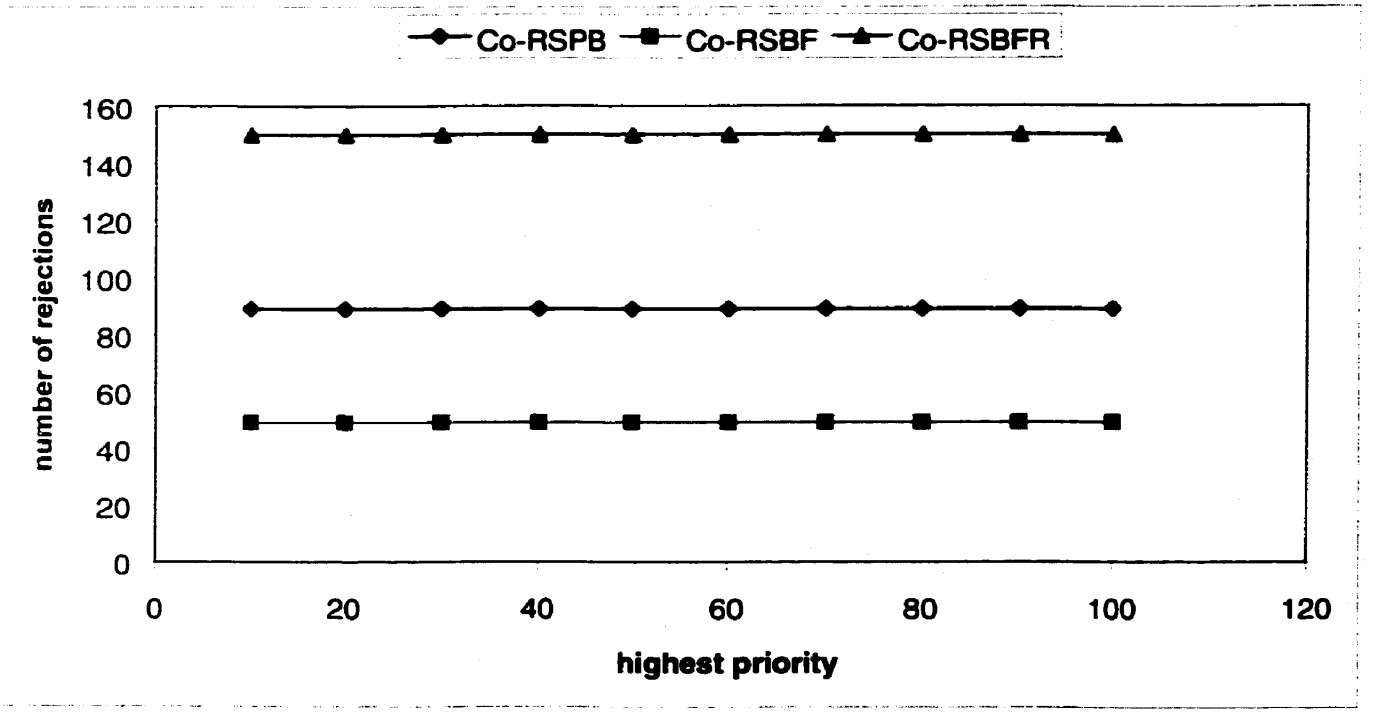


(b)

Figure 5.12: (a) System benefit (b) number of rejections versus percentage of floating requests.



(a)



(b)

Figure 5.13: (a) System benefit (b) number of rejections versus scope of priority.

From Figure 5.12(a) and 5.12(b), it can be noted that when all requests ask for fixed machines, the number of rejections are highest, therefore the system benefit are lowest for all three algorithms. As the percentage of requests requiring floating machines increases, the number of rejections for all three algorithms decreases and the system benefit increases. This is because the requests for the fixed machines have less flexibility, therefore increasing the chance of rejections.

Figure 5.13(a) shows the variation of system benefit with priority. Figure 5.13(b) shows the variation of the number of rejections with the highest priority. The lowest priority is set to be 1. The highest priority ranged from 10 to 100.

From Figure 5.13(a) and 5.13(b), we can see that the scope of priority does not affect the value of system benefit and the number of rejections. This proves that the system benefit calculation model described previously is correct.

6 CONCLUSIONS AND FUTURE WORK

This project presents a resource management architecture supporting advance reservations for a Grid computing system and introduces a novel way of incorporating QoS constraints as well as priority into an advance reservation system including a co-reservation scheduling algorithm. The project compares the performance of the proposed RSPB algorithm with an existing advanced reservation algorithm, namely the Resource Broker, and compares the performance of Co-RSPB with two comparison algorithms developed in this thesis, and analyzes the simulation results. The QoS constraints are specified using an abstraction called benefit functions. Although the proposed algorithm is designed to reserve CPU resources, it is easy to extend the algorithm to reserve other resources such as network bandwidth, disk, memory, etc. It is also possible to extend the algorithm to support multiple dimension benefit functions, such as time deadline benefit functions. The primary contributions of this thesis are:

- Designing a resource management architecture supporting advance reservations for a Grid computing system;
- Introducing a novel way of incorporating QoS constraints and priority into an advance reservations scheduling algorithm for a Grid computing system;
- Developing a *Reservation Scheduler with Priorities and Benefit Functions* (RSPB), which improves the performance of existing approach (RB) by considering priorities and benefit functions associating the application;
- Developing a *Co-Reservation Scheduler with Priorities and Benefit Functions* (Co-RSPB), which is the first co-reservation scheduling algorithm separating from traditional scheduling for admission control in Grid computing system;
- Developing two comparison algorithms Co-Reservation Scheduler with Best Fit scheme (Co-RSBF), and Co-Reservation Scheduler with Best Fit and Refine scheme (Co-RSBFR), as

base line to evaluate the performance of Co-RSPB. Simulation results show that the Co-RSPB has a very good performance by satisfying larger number of reservation request.

- Developing a novel system benefit calculation model to quantify the system performance in terms of QoS service.

Several future directions are identified for further investigation. Some of them include:

- Developing schemes for incorporating multiple QoS constraints into the admission control problem;
- Comparing different data structures to find fastest algorithm for searching in the time slot table;
- Extending reservation of CPU resources to reservation of other resources, such as network bandwidth, disk, memory, etc.;
- Designing protocols to address communication overheads problem;
- Implementing the prototype of a QoS driven RMS using the proposed algorithms.

ACRONYMS

AC	Admission Controller
API	Application Programmer's interface
ARS	Advance Reservations Server
BFD	Best Fit Decreasing algorithm
Co-RSBF	Co-Reservation Scheduler with Best Fit scheme
Co-RSBFR	Co-Reservation Scheduler with Best Fit and Refine scheme
Co-RSPB	Co-Reservation Scheduler with Priorities and Benefit functions
CPU	Central Processing Unit
DSRT	the Dynamic Soft Real Time system
ERDoS	the End-to –End Resource Management of Distributed System QoS architecture
GARA	A Globus Architecture for Reservation and Allocation
H-SFQ	Hierarchical Start-time Fair Queuing
LRP	Lazy Receiver Processing
PARSEC	PARallel Simulation Environment for Complex System
PVM	Parallel Virtual Machine
QLinux	A QoS enhanced Linux kernel for multimedia computing
QoS	Quality of Service
RB	Resource Broker
RIS	Resource Information Service
RMS	Resource Management System

RSPB	Reservation Scheduler with Priorities and Benefit functions
RSVP	Resource ReSerVation Protocol
RT	Real-Time
SFQ	Start-time Fair Queuing
WWW	World Wide Web

REFERENCES

[BaM98] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song, "PARSEC: A Parallel Simulation Environment for Complex System", *IEEE Computer*, Vol. 31, No.10, Oct, 1998, pp.77-85.

[BeL98] S. Berson, R. Lindell, and R. Braden, *An Architecture for Advance Reservations in the Internet*, technical Report, USC Information Sciences Institute, Los Angeles, July 1998.

[ChN97] H. Chu and K. Nahrstedt, "A Soft Real Time Scheduling Server in UNIX Operating System", *in Proceedings of IDMS'97, (European Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services),* September 1997, Darmstadt, Germany.

[ChS98] S. Chatterjee, B. Sabata, and J. J. Sydir, *ERDoS QoS Architecture*, Technical Report ITAD-1667-TR-075, SRI International, California, May 1998.

[DSRT] software for DSRT, http://cairo.cs.uiue.edu/software/DSRT-2/dsrt-2.html

[FeG95] D. Ferrari, A. Gupta, and G. Ventre, *Distributed Advance Reservation of Real-Time Connections*, technical Report, TR-95-008, Telnet Group, University of California and International Computer Science Institute, Berkeley, March 1995.

[FoK97] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, No. 2, summer 1997, pp. 115-128.

[FoK99a] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure, Second Edition*, Morgan Kaufmann, 1999.

[FoK99b] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservation and Co-allocation," *in Proceedings of Seventh IEEE International Workshop on Quality of Service (IWQoS 99)*, London, UK, May 31 - June 4, 1999.

[Fos99] I. Foster, "Building the Grid: Integrated Services and Toolkit Architecture for Next Generation Networked Applications", http://www.gridforum.org/building_the_grid.htm, 1999.

[Gar99] G. Garimella, "Advance CPU Reservations with the Dynamic Soft Real-Time Scheduler", Master's Thesis, University of Illinois at Urbana-Champaign, 1999.

[GoG96] P. Goyal, X. Guo, and H.M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems", *Proceedings of 2nd Symposium on Operating System Design and Implementation (OSDI'96), Seattle, WA*, October 1996, pp.107-122.

[HaJ96] E. Hanson, and T. Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," Information Systems, Vol. 21, No. 3, 1996, pp. 269-298.

[ISList] software for Interval Skip List, http://www.cise.ufl.edu/~hanson/IS-lsits/

[JoG99] W. E. Johnston, D. Gannon, and B. Nitzberg, "Information Power Grid Implementation Plan: Research, Development, and Testbeds for High Performance, Widely Distributed, Collaborative, Computing and Information Systems Supporting Science and Engineering," NASA Ames Research Center, http://www.nas.nasa.gov/IPG, 1999.

[KiN00] K. Kim and K. Nahrstedt, "A Resource Broker Model with Integrated Reservation Scheme", Proceedings of IEEE International Conference on Multimedia and Expo 2000 (ICME2000), New York, NY, July 31 - August 2, 2000.

[Legion] Legion home page, http://www.cs.virginia.edu/~legion/

[Mah99] M. Maheswaran, "Quality of Service Driven Resource Management Algorithms for Network Computing," *1999 International Conference on Parallel and Distributed Processing Technologies and Applications (PDPTA '99)*, June 1999, pp. 1090-1096.

[Mah01] M. Maheswaran, "Data dissemination approaches for performance discovering in Grid Computing System," 10th IEEE Heterogeneous Computing Workshop (HCW 2001), Apr. 2001, to appear.

[MaK00a] M. Maheswaran and K. Krauter, *A Parameter-based Approach to Resource Discovery in Grid Computing System*, Technical Report TR-CS00-13, Department of Computer Science, University of Manitoba, Winnipeg, May, 2000.

[MaK00b] M. Maheswaran and K. Krauter, "A Parameter-based Approach to Resource Discovery in Grid Computing System," 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000), Dec. 2000.

[Man99] F. Manola, *Characterizing Computer-Related Grid Concepts*, technical report, Object services and consulting, Inc., Texas, March, 1999. http://www.objs.com/agility/tech-reports/9903-grid-report-fm.html

[PARSEC] PARSEC home page, http://may.cs.ucla.edu/projects/parsec/

[PVM] PVM home page, http://www.epm.ornl.gov/pvm/pvm_home.html

[QLinux] software for Qlinux, http://www.cs.umass.edu/~lass/software/qlinux

[ScN99] O. Schelen, A. Nilsson, J. Norrgard, and S. Pink, "Performance of QoS Agents for Provisioning Network resources", In IFIP Seventh International Workshop on Quality of Service (IWQoS'99), London, UK, June 1999.

[ScP98] O. Schelen, and S. Pink, "Resource Sharing in Advance Reservation Agents", *Journal of High Speed Networks*, Special Issue on Multimedia Networking, 1998.

[SmF00] W. Smith, I. Foster, and V. Taylor, "Scheduling with Advanced Reservations", *International Conference on Parallel and Distributed Processing System*, May 2000.

[VoK95] A. Vogel, B. Kerhervé, G. Bochmann, and J. Gecsei, "Distributed Multimedia and QoS: A Survey," IEEE MultiMedia, summer 1995, pp. 10-19.

[ZhD93] L. Zhang, S. Deering, D. Estrin, S. Shenkar, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," IEEE Networks, September 1993, pp. 8-18.