ABSTRACT

CASTILLO, CLARIS. On the Design of Efficient Resource Allocation Mechanisms for Grids. (Under the direction of Dr. George Rouskas and Dr. Khaled Harfoush).

In this thesis we consider the problem of providing QoS guarantees to Grid users through advance reservation. Advance reservation mechanisms provide the ability to allocate resources to users based on agreed-upon QoS requirements and increase the predictability of a Grid system, yet incorporating such mechanisms into current Grid environments has proven to be a challenging task due to the resulting resource fragmentation. In view of these observations we have devised efficient scheduling algorithms that support advance reservations. We can organize this thesis in two parts.

We first use concepts from computational geometry and efficient data structures to present a framework for tackling the resource fragmentation, and for formulating a suite of scheduling strategies. We also develop efficient implementations of the scheduling algorithms that scale to large Grids. We conduct a comprehensive performance evaluation study using simulation, and we present numerical results to demonstrate that our algorithms perform well across several metrics that reflect both user- and system-specific goals.

Advance reservations has also been proposed as one mechanisms to provide Grid resource managers with the ability to co-allocate resources. Co-allocation of resources is one problem that has gained increasing attention due to the emergence of complex applications that require orchestration of resources never envisioned before. In practice, a co-allocation request can be handled manually as a set of individual advance reservations requests. However, such a solution can be computationally expensive and inappropriate for time-sensitive applications. Furthermore, the trend towards more dynamic solutions has emphasized the need for more automatic solutions. As a second contribution, in this thesis we design and develop a co-allocation algorithm that is efficient in co-allocating resources while respecting the atomicity of the co-allocation request and improving user and system performance. This is achieved by quantizing the temporal space and using efficient 2-dimensional balanced search trees. We perform a comparative analysis of our algorithm by means of simulations driven by workloads from real systems and show that our algorithm scales to Grid systems with large number of resources and heavy workloads.

On the Design of Efficient Resource Allocation Mechanisms for Grids

by
Claris Castillo

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Helen Gu	Dr. Gregory Byrd
Dr. George Rouskas Chair of Advisory Committee	Dr. Khaled Harfoush Co-Chair of Advisory Committee

DEDICATION

To Lupe, Luchi and Cristel:
You have been my strength through all these years.

To my dear friend Flor Barrios:
You have been like a sister to me.

To my dearest brother-in-law, Joseph Payne:
Welcome to the Family.

To my unborn niece/nephew:
Voy a darte mucho amor!

BIOGRAPHY

Claris Castillo was born in Panama, Panama. She received an undergraduate degree in Electrical Engineering from the University of Panama where she also worked as a lecturer. She later received a MS degree in Computer Networking from the department of Electrical Engineering in North Carolina State University. Where she is also pursuing her PhD. degree in the department of Computer Science.

She has worked as a consultant for Centauri Technologies Corporation in Panama and has interned at Intel Research, IBM Research (T.J. Watson) and Cisco Systems Inc. in the United States. She is actively involved in improving participation of minority groups such as women and Latinos in computing related disciplines. She is an active member of WiCS (Women in Computer Science) and co-founder of the Latinas in Computing community established in 2006 with the support of the Anita Borg Institute. She is also member of IEEE (Institute of Electrical and Electronics Engineers), ACM (Association of Computing Machine), SHPE (Society of Hispanic Professional Engineers), SWE (Society of Women in Engineering) and Systers. She is member of IEEE and ACM.

After graduation she will join IBM T.J. Watson Research Center as a Research Staff Member.

ACKNOWLEDGMENTS

I would like to thank Dr. George Rouskas for believing in me and teaching me how to do research. You certainly changed the course of my life and left your footprint in my professional career. This thesis would have not been possible without your help and support. My thanks also go to Dr. Khaled Harfoush who gave me the opportunity to do research during my masters. Our discussions on different topics of Computer Science strengthened my interest in pursuing a doctoral degree. I thank you both for your invaluable input into this doctoral thesis.

Much gratitude to my parents, Lupe and Luchi, who gave me all kind of support during my graduate school. Special mention to my mom who has been standing by me in my every step. To my dad whose advice through out all these years have illuminated the path in several occasions. To my sister, Cristel, whose experience, support and *tutorials* have influenced many of my decisions in the last years. To Flor who has shared with me all the up and downs. Flor, your friendship means so much to me.

Special thanks to Dr. Mladen Vouk and Dr. David Thuente who have not only supported my graduate work but also my extra-curricular efforts. Many doors have been opened thanks to this support. My gratitude also goes out to the departmental staff who often guided me in different aspects of the graduate program and made sure that I complete all the requirements to achieve my doctoral degree. To my friends in the department, who brought color, love and joy into my daily routine at the lab.

Finally, I would like to acknowledge Cisco Systems Inc., CACC (Center for Advanced Computing and Communication), NSF (under grants ANIR-0347226 and CNS-0434975), IEEE (travel grants to IPDPS) and Google scholarship (Anita Borg Institute) for funding my research through out my PhD.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 QoS in Grids

Much progress has been made in Grid technologies. Grids have become the de-facto computing infrastructure for research institutions and Industry. Moreover, the advances made in resource virtualization and service technologies have enabled a new broad range of complex applications and computing paradigms never envisioned before. For instance, Amazon Elastic Computing Cloud [1] allows users to obtain and configure computational capacity, providing them with complete control of the computing resources and a fully proven virtual environment.

Quality of Service (QoS) is one aspect that is fundamental to the full realization of these emerging applications. In practice, however, most Grid infrastructures offer limited QoS support. This state of affairs has its root mainly in historical and technical reasons. First, Grid development was initially driven by the need of aggregating resources in a cost-effective way, thus leaving QoS support as a secondary consideration in their design. Second, the complex nature of Grid environments imposes several technical challenges that hinder the development of QoS-support techniques.

One scheduling mechanism that has been proposed to provide for QoS in Grids is advance reservations. Advance reservations capabilities have gained increasing interest in the Grid community due to their ability to increase the predictability of the system, enable the co-allocation of resources and provide availability guarantee of resources to applications. Nevertheless, several arguments have made their development and adoption difficult. (1) Advance reservations have shown to cause severe system performance degradation; (2) typical advance reservation mechanisms lack flexibility, as they do not permit graceful degradation in application performance when resource management policies mandate changes in allocations; and (3)

existing approaches suffer from poor scalability as they are not effective in managing large sets of advance reservations or handling resource fragmentation. To overcome these challenges, algorithms for advance reservations need to be *efficient* so they can adapt to dynamic changes in resource availability and users' demand without affecting system and user performance.

Co-allocation of resources enables applications with high QoS requirements to orchestrate multiple resources for their execution. In fact, it is one functionality resulting from the use of advance reservations since by means of advance reservations a resource manager can guarantee the availability of multiple resources at a specific time in the future. Nevertheless, such approach may incur long scheduling times and, therefore, hinder the QoS perceived by applications. This observation leads to the conclusion that, although the co-allocation problem is a specific case of the problem of allocating resources in advance, there is a need for developing new mechanisms capable of accommodating for their differences.

We believe that the ability to offer and guarantee QoS to users is of utmost importance to Grid providers. Without QoS guarantees, users may be reluctant to pay for Grid services or contribute resources to Grids, impeding further development of the Grid model and limiting its economic significance. Mechanisms for support of QoS also enable service providers to differentiate themselves by offering an optimized menu of services. Therefore, in this thesis we present a framework for designing effective and efficient scheduling algorithms that employ advance reservations and support co-allocation of resources to guarantee QoS to users.

1.2 Contributions

The contribution of this thesis is two-fold. First, we use concepts from computational geometry to present a framework for tackling resource fragmentation, and for formulating a suite of scheduling strategies that support advance reservations and user deadlines. We also design and develop efficient implementations of the scheduling algorithms that scale to large homogenous and heterogeneous Grid environments. We conduct a comprehensive performance evaluation study using simulation, and we present numerical results to demonstrate that our strategies perform well across several user and system performance metrics and overcome the lack of scalability and adaptability of existing mechanisms.

Second, we consider the problem of co-allocation of resources in Grid-like environments. We develop an online co-allocation algorithm that is *efficient* in co-allocating resources while providing support for advance reservations (QoS guarantees). We achieve this by partitioning the temporal space into a set of quanta and by using efficient 2-dimensional balanced search

trees to organize the co-allocations. We have also performed an in-depth comparative analysis of our algorithm against conventional batch schedulers under real workloads. Our results provide some insightful conclusions indicating that online scheduling algorithms may achieve–under most conditions–high overall rate of utilization, while providing smaller delays and better QoS guarantees without adding much complexity. We also show that our co-allocation algorithm scales to systems with large number of resources and heavy workloads.

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2 we briefly introduce Grids and motivate their need for QoS support. In Chapter 3 we propose a framework that uses concepts from computational geometry to schedule resources in homogeneous environments. We consider the same problem of allocating resources in advance but in heterogeneous environments in Chapter 4. In Chapter 5 we present a scheduling algorithm that uses quantization to co-allocate resources. Finally, we conclude the thesis and discuss future work in Chapter 6.

Chapter 2

Grids, QoS and Advance Reservations

2.1 The Grid

The term "Grid" emerged as a term denoting a proposed distributed computing infrastructure for advanced science and engineering [2, 3]. It has its root in the power grid: just as the power grid brought about a radical change in making power universally accessible in the beginning of the 20th century, computation Grids provide users with reliable, pervasive and low-cost access to computational power.

Grids have became an essential infrastructure for resource-intensive scientific and commercial applications [2, 4] as they enable the sharing and dynamic allocation of distributed, high-performance computational resources. They minimize the associated ownership and operating costs and hence promote flexibility and collaboration among diverse organizations.

Initially, the development of Grid technologies was driven by the need of the scientific community to collaborating over the network. Therefore, enabling resource sharing was the major goal in the design and development of the first Grids in the mid 1990s; resources from a broad variety such as large data sets, computational resources, software, scientific instrumental devices (e.g., telescopes) were shared and coordinated among multiple virtual organizations and academic institutions. Since then, a wide variety of scientific applications have been developed to leverage the aggregative capability of resources, and have rapidly increased in complexity and size [5, 6, 7, 8].

2.2 Quality of Service (QoS) in Grids

Owing to the advances in technologies such as resource virtualization, web services, service oriented architectures (SOA) and network management among others, Grids have experienced enormous growth not only in respect to their adoption–they have became the de-facto infrastructure for provisioning computing service in academia and corporate *R&D* environments– but also in their functionality, complexity and size.

This phenomenon has led to the emergence of a wide range of applications capable of performing tasks of a complexity not envisioned before. For instance, several scientific workflow applications [9, 6, 10] involve the orchestration of multiple computation and data transfer stages. These stages normally have strong dependency on completion times; thus the ability to co-schedule and synchronize resource usage is crucial. Furthermore, emerging classes of deadline-driven scientific and commercial applications such as financial market and severe weather modeling [5] require simultaneous access to multiple resources and predictable completion times.

One major paradigm that has its roots in the advancement of Grid technologies is *on-demand computing*. On-demand computing [11, 12] has been proposed as a viable model in which a wide range of finer grained commercial, business, and scientific applications would tap into the Grid resources on an as-needed basis, extending the reach and utility of Grid computing far beyond its current user base to society as a whole. For instance, more recently Amazon launched its own Cloud computing service (EC2) [1]; earlier, Sun Microsystems started offering the Sun Grid computation utility [13], and more such services are expected in the near future. This vision of computing as an utility is expected to change not only the way scientists and businesses work, but also the way they think about computing resources.

The realization and full adoption of the aforementioned applications and paradigms depends on the development of sophisticated resource management systems capable of allocating resources to users based on agreed upon quality of service (QoS) requirements [14], while satisfying certain system level objectives (e.g., high utilization, economic constraints, etc.) [15, 16]. Furthermore, without QoS guarantees, users may be reluctant to pay for Grid services or contribute resources to Grids, hindering further development of the Grid model and limiting its economic significance. Mechanisms for supporting QoS also enable service providers to differentiate themselves by offering an optimized menu of services.

In spite of these observations, support for QoS is fairly limited in existing Grid infrastructures. This state of affairs has its roots in many historical and technical factors. Initially,

the major driving force behind the development of Grid technologies was the need to aggregate and share computational resources in a *cost-effective* way among the scientific community. Hence QoS-support was left as a secondary consideration. Furthermore, Grid environments are intrinsically complex in that they consist of heterogeneous resources geographically and logically distributed among multiple virtual organizations, with different policies dictating access and usage of resources. Moreover, due to the lower costs of commodity hardware and the development of technologies such as OS virtualization–that break the physical barrier of isolated systems–Grids are increasingly growing in size.

Much work can be found in the literature addressing the aforementioned technical challenges. Majority of this work lays within the context of service level agreements (SLA) [17, 18], which in essence aim at providing QoS by defining a set of rules that determine the interaction between the user and the Grid service provider. We argue that although embedding QoS-support at higher layers of the architecture is required, having QoS-oblivious mechanisms (e.g., schedulers) at lower layers is counter-productive as they are likely to hinder the effectiveness of the high-layer mechanisms in achieving their goals (i.e., providing QoS guarantees).

One important component of a Grid architecture is the scheduler. The scheduler's main task is to allocate resources to incoming requests for service, and therefore, it is primarily responsible for the QoS permissible in a Grid. Following this observation, in this thesis we focus on embedding QoS-support into the Grid scheduler.

2.3 Scheduler and QoS

Scheduling and management of Grid resources is an area of ongoing research and development. Several open source or proprietary schedulers have been developed for clusters of servers, including Maui [19, 20], CONDOR [21], Catalina [22], LoadLeveler [23], portable batch system (PBS) [24], and load sharing facility (LSF) [25]. They typically work under best-effort policies, run in batch mode, can be customized to specific policies, and balance the load among various servers. However, the primary objective of most existing approaches is to improve overall system performance (e.g., utilization), while the QoS experienced by Grid users is, at best, of secondary consideration [4]. For instance, batch systems typically allocate resources to jobs as they become available, without considering applications that need to obtain results within a strict deadline [14]. In general, the schedulers process jobs in order of priority, which is determined based on job attributes such as job class and time in queue [19, 20]; they also employ *backfilling* operations, i.e., run jobs out of order, to make better use of the available resources.

Unfortunately, backfilling often interferes with the ability of the system to provide QoS guarantees, as in its attempt to improve utilization it may bypass the job priorities set by the system administrator [20].

2.4 Advance Reservations

Advance reservations, i.e., the ability of the scheduler to guarantee the availability of resources at a particular time in the future, is one mechanism that has been proposed for provisioning performance predictability, meeting resource requirements and providing guaranteed QoS to applications in Grid environments.

Overall, advance reservation of resources [26, 4, 27, 2, 28, 29, 26, 4, 2, 28, 29, 30, 31, 32, 33, 31, 34, 35, 36] has generated great interest in the Grid community; and some existing schedulers, including Maui [19] and Condor [21], provide some sort of advance reservation mechanisms. However, existing approaches for making reservations in the future lack sophistication, are expensive, and do not scale well.

As a matter of fact, despite the attractive features of advance reservations, there is great scepticism in the Grid community about their ability to meet their promise; this fact is mainly due to three reasons. First, advance reservations have shown to cause severe performance degradation [29, 27]. Second, typical advance reservation mechanisms lack flexibility as they do not permit graceful degradation in application performance when resource management policies mandate changes in allocations [37]. Third, existing approaches suffer from poor scalability as they are not effective in managing large sets of advance reservations or handling resource fragmentation. Also, most solutions lack sophistication, and are not able to address the user needs (e.g. time guarantees) and system requirements (e.g., high performance/throughput) in an integrated manner. To overcome these challenges, algorithms for advance reservations need to be *efficient* so they can adapt to dynamic changes in resource availability and user demand without hurting system and user performance.

2.4.1 Co-allocation of Resources

Advance reservation has also been proposed as one mechanism to effectively support co-allocation of resources [38], i.e., the simultaneous allocation of multiple resources to a single user or application. For instance, by incorporating advance reservation capabilities the resource manager can guarantee that a set of resources will be available and offer a specific level of service when required. The problem of co-allocating resources has gained increasing recognition as a

result of a broad range of emerging end-to-end sophisticated applications that harness network and computational resources over the Internet. One such common application is experimental reconstruction. For example, in [39] a scientific instrument along with multiple computers and display devices were used for collaborative real-time reconstruction of X-ray source data. The importance of these applications has been recognized by research, industry and government communities through the support of several initiatives [7] [40] [8].

In principle, such simultaneous allocations can be achieved manually by reserving each allocation in advance. However, the growing trend towards more virtualized environments has emphasized the need to provide automated solutions to support the management and coordination of multiple resources.

One of the key challenges in the design of such management mechanisms is *scalability*. Large scale distributed systems tend to aggregate large number of resources that are typically geographically distributed in a network to support sophisticated end-to-end applications. Designing algorithms that are efficient in scheduling jobs in such large-scale environments is difficult. There is two-fold benefit of achieving this goal. First, it offers QoS support to users, as reflected by short response times–a feature that is crucial to the success of emerging sophisticated end-to-end applications. Second, it provides the resource managers with an ability to respond quickly to changes in resource availability and users' demand that occur over short periods of time–a common condition in emerging distributed environments [41].

Meta-scheduling [42, 43, 44] is one such mechanism that has been proposed to orchestrate the access to resources within several domains in a Grid. However, lack of efficient scheduling algorithms capable of scaling to large number of resources hinder their implementation and adoption. To exacerbate the problem, wide adoption of OS virtualization [45] promises to enable computing systems of large magnitudes, further escalating the aforementioned issues.

2.4.2 Related Work

In this section, we discuss some of the most relevant work to this thesis in the context of advance reservations in general, and resource co-allocation more specifically.

GARA (Globus Architecture for Resource Allocation) [38] is one of the seminal works on advance reservation and defines a basic architecture and simple API for the manipulation of advance reservations of different resources. Since then, numerous papers have studied the impact of advance reservation on system and user performance. One such work theoretically proves that reservations can be used to improve the performance predictability of applications [32]. To order

to prove this, the author enhanced the ASKALON scheduler [46] to allow for negotiation between a user and the resource manager. The performance of advance reservations in the context of workflow applications has also been investigated [30]: this work concludes that significant reduction in the completion time of is possible by using FIFO or fair share policies. It also shows that dynamic provisioning, with no support for advance reservation at the remote sites, can also be used with equal or more effectiveness than advance reservations. It also has been observed that waiting times of applications submitted to the scheduling queue increases when reservations are supported [27]. Further, the authors also found that the best performance is achieved when we assume that applications can be terminated and restarted, backfilling is performed, and relatively accurate run-time predictions are used. A simulation based study to analyze the impact on system and user performance of advance reservation is presented in [29]. The study concludes that reservation based systems benefit parallel and reserved jobs significantly. A more general study on the usefulness of advance reservation is presented in [33]. Reaching similar conclusions regarding the usefulness of advance reservations by means of theoretical analysis, the authors in [47] propose a reservation architecture based on a "the predictability trinity": Grid Scheduling, Reservation Fabric and Performance Models.

It is worth mentioning that most of the aforementioned research work overlooks the issues of scalability and efficiency encountered when supporting advance reservations. This is in contrast to our work where efficiency is an important factor in the design and development of our scheduling algorithms. In fact, the mentioned works show that the Grid community has not reached a consensus regarding the benefits and drawbacks of advance reservations. Nevertheless, we believe that the drawbacks far outweigh the positives of current advance reservations mechanisms and this justifies the importance of the problem being solved in this thesis.

One of the major criticism of advance reservations is that they do not allow for graceful degradation in performance. This is due to the assumption that users know a priori the duration of their jobs. The introduction of laxity and fuzziness has been explored to overcome this challenge [48, 49].

More recent works in advance reservations in Grids [50] [35] incorporates for negotiation and optimization techniques. For instance, a multi-objective genetic algorithm for selecting resources so as to optimize the application performance while minimizing the resource costs is proposed in [50]. It has also been shown that a cost-aware resource model in which reservation for each application task is performed separately by negotiating with the resource provider is feasible to support advance reservations [35]. Leveraging economy concepts, a broker service for Grid resources is presented in [51] which takes into account the fact that deadline and budget

are specified, and then optimizes the usage of resources only by considering the current state of the resources but without any planning horizon. Note that these works rely on multiple mechanisms such as negotiation to schedule reservations and ignore the issues of efficiency that become important as Grid systems grow in size. In our work we focus on providing algorithms that are efficient and fully capable of scheduling reservations, thus, avoiding dependency on additional administrative mechanisms.

Several commercial and open-source resource managers have been developed for the Grid [23, 25, 52, 24, 53]. Most of this work relies on batch scheduling techniques and therefore suffer from the same limitations mentioned earlier. Furthermore, most of these schedulers employ backfilling operations, i.e., run jobs out of order, to make better use of the available resources. Unfortunately, backfilling often interferes with the ability of the system to provide QoS guarantees, as in its attempt to improve utilization it may bypass the job priorities set by the system administrator [20, 54]. A complete survey and discussion on parallel scheduling techniques such as backfilling and gang scheduling can be found at [54, 55]. In such existing managers the support advance reservations is very limited and lack of sophistication.

The studies in [56, 57] represent some of the earliest work on co-allocation mechanisms for distributed computing environments. The main contribution of these works is a multi-layered approach in which a set of basic co-allocation mechanisms can be used to construct a wide range of application-specific co-allocation strategies. In [58], the authors address several issues regarding the management of resources in the context of meta-computing. More specifically, they focus on location and allocation of resources, process creation and authentication. The concept of virtual resources is introduced in [59] to support co-reservations. In [60, 61] the authors propose a framework for mapping resources to tasks in which the overall objective is to minimize schedule length. In order to do this the framework relies on an adaptation process that changes the scheduling order and/or resource assignments at run time. Similar to our previous observation on schedulers that support advance reservations, most of this work rely on additional mechanisms on top of the scheduler to guarantee resource allocation.

Scheduling in multi-site Grid environments is in essence equivalent to the co-allocation problem with network resources being the major differentiating factor. There has been work showing that such co-allocation is beneficial as long as the communication overhead is kept below 25% [62]. In a similar vein, in [63] authors present a comparative study of local and global queues and propose four scheduling policies for processor co-allocation in multi-clusters. Their main conclusion is that, in general, using multiple local queues yields better performance than using a single global queue. Two algorithms based on batch scheduling techniques are proposed in [64].

Authors conclude from their experiments that scalability is one of the key differences between the co-allocation problem in Grid and parallel computing environments.

In order for workflow applications to meet time dependencies in between sub-tasks they require simultaneous access to multiple resources. Therefore, algorithms such as list scheduling– widely used in this context–have been adapted to accommodate for the specifics of Grid environments. For instance, in [65] the authors propose a scheduling algorithm based on list-scheduling algorithm for finding the minimum execution time of a set of parallel tasks that require co-allocation in Grids. More recently, authors in [66] extended the well known HEFT (Heterogeneous Earliest Finish Time) algorithm and proposed two heuristics that support co-allocation and advance reservations in Grids. Both works [32, 30] mentioned earlier also address the co-allocation problem in the context of workflow applications.

Chapter 3

Efficient Scheduling Algorithm for Grids - homogeneous

In this chapter we consider the problem of providing QoS guarantees to Grid users through advance reservation of resources in homogeneous environments. We use concepts from computational geometry to present a framework for tackling the resource fragmentation, and for formulating a suite of scheduling strategies. We also develop efficient implementations of the scheduling algorithms that scale to large Grids. We conduct a comprehensive performance evaluation study using simulation, and we present numerical results to demonstrate that our strategies perform well across several metrics that reflect both user- and system-specific goals.

The chapter is organized as follows. In Section 3.1 we describe the online scheduling problem we study in this work, and in Section 3.2 we present a framework for reasoning about advance reservations that borrows ideas from computational geometry; we also describe a suite of scheduling strategies that arise naturally within the framework. In Section 3.3 we provide additional details on the implementation of the scheduling algorithms and of the data structures related to managing the fragmentation of resources. In Section 3.4 we present simulation results to evaluate the various strategies in terms of several performance metrics, and we conclude the chapter in Section 3.5.

3.1 Problem Description

Consider a scheduler $\mathcal{S}$ for a Grid with n servers which may be geographically distributed in a network.We make the assumption that all servers are identical in terms of their processing capacity C. In Chapter 4 we extend the algorithms we present here to non-identical

resources. A user with job j requiring service submits a request to the scheduler. The request is characterized by a three-parameter tuple (r_j, l_j, d_j), where:

1. r_j is the *ready time* of the job, i.e., the earliest the job can be made available to the Grid for processing;

2. l_j is the *length* of the job, i.e, the amount of work the job requires; and

3. $d_j (\geq r_j + l_j)$ is the *deadline* of the job, i.e., the latest time by which the job can be completed.

The deadline is a measure of the quality of service required by the user. We assume that deadlines are *hard*, in that a user receives utility only if the job completes service by its deadline. Therefore if $\mathcal{S}$ determines that the deadline cannot be met, it drops the job and notifies its user accordingly.

We consider the online scheduling problem whereby users submit service requests to $\mathcal{S}$ at random instants. We assume that $\mathcal{S}$ maintains a schedule which records, for each server i, the time periods in the future during which the server is reserved for jobs that have already been accepted to the system. In essence, this schedule represents the set of *advance reservations* that have been made, and it guarantees that server resources will be available to the accepted jobs at specific future times. Figure 3.1(a) shows an example schedule for a 2-server system. The schedule shows that at the current time (i.e., time $t = 0$ in the figure), there are three jobs scheduled for server 1: the job currently in service which will end at time t_1, job A which has reserved the server from time t_3 to t_5, and job D which has reserved the server from time t_9 to t_{11}; similarly, three jobs have been scheduled for server 2. The figure also shows a service request for scheduling a new job j with ready time $r_j = t_6$ and length $l_j = t_8 - t_6$.

When a service request (r_j, l_j, d_j) for a new job j arrives, $\mathcal{S}$ immediately runs an algorithm to determine whether it is feasible to schedule the job so as to meet its deadline. If so, then $\mathcal{S}$ uses a set of criteria to select one of the (possibly multiple) servers who can handle this job, updates its schedule, and returns a reference to this server to the user; otherwise, the job is dropped. The scheduling decision impacts the performance perceived by users as reflected by the fraction of jobs meeting (or missing) their deadlines and the turnaround times of the jobs. It also impacts the overall system performance as reflected by the system utilization, which is a measure of how well the overall service capacity of the system is used. The challenge, therefore, is to develop efficient online scheduling algorithms that minimize the fraction of dropped jobs while maximizing utilization.

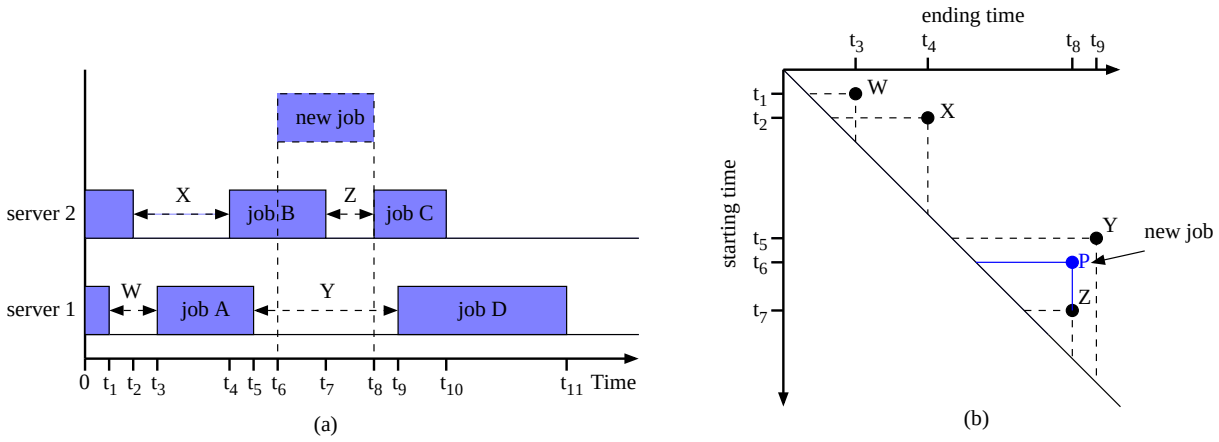


Figure 3.1: (a) Advance reservations in a 2-server system: jobs scheduled and idle periods, (b) equivalent geometric representation of the schedule: idle periods as points in the plane

Several variants of this scheduling problem with advance reservations and/or deadlines have been studied in multiprocessor and Grid systems [27, 28, 67, 68, 69]. However, the heuristic solution approaches that have been proposed may not scale well and may not utilize the available system capacity efficiently [14, 2]. In the next section, we present a new framework for developing efficient algorithms for this problem taking into account a range of optimization criteria.

Before we proceed to address the general scheduling problem, let us consider a restricted version in which jobs must be scheduled as soon as they are ready. In this case, deadlines are immediate (i.e., $d_j = r_j + l_j$), and we refer to this problem as *resource scheduling with immediate deadlines*. One straightforward approach for tackling this problem is for the scheduler $\mathcal{S}$ to keep track of the *completion time* of each server, defined as the latest time at which the server becomes free based on the existing advance reservations. The scheduler then assigns an arriving job to the server with the latest completion time that is earlier than the ready time of the new job. This latest available completion time (LACT) algorithm takes time $O(\log n)$ to schedule a job. However, it can be inefficient in terms of both capacity utilization and job drop rate, as it does not consider the idle periods created at each server between the times reserved for jobs whose requests were submitted earlier. For instance, in the scenario shown in Figure 3.1(a) for a 2-server system, the completion time for server 1 is t_{11} (the service completion time of job D), while the completion time for server 2 is t_{10}. Therefore, the LACT algorithm will reject the service request for the new job with arrival time $t_6 < t_{10} < t_{11}$, although the job can be accommodated on server 1 within the idle period Y created between jobs A and D.

An algorithm that considers the idle periods when making decisions was developed in [70] in the context of scheduling bursts in optical burst switched networks. The algorithm

uses concepts from computational geometry [71] to represent the time intervals corresponding to idle periods as points in a plane, as illustrated in Figure 3.1(b). Since the ending time of an idle period must be greater than its starting time, all points will always be above the diagonal in Figure 3.1(b). Then, the problem of finding a feasible idle period for scheduling a new job (also represented as a point P in the plane) is equivalent to finding a point that *completely contains*[1] point P. In Figure 3.1(b), it is seen that point Y completely contains the point corresponding to the new job, thus the latter can be scheduled within idle period Y on server 1. By maintaining a balanced priority search tree data structure [72] containing all the idle periods on all servers, finding an idle period for a new job, or determining that one does not exist, takes time $O(\log K)$, where K is the number of idle periods. Updating the data structure to add new idle periods (created when a new job is scheduled) or remove ones in the data structure (as time advances), also takes time $O(\log K)$. The value of K, however, can be significantly larger than the number n of servers, and we have found that its value increases rapidly with the offered load of jobs; in other words, in moderately to highly loaded systems, in which it is important to make scheduling decisions quickly, the running time of the algorithm is longer.

3.2 Scheduling with General Job Deadlines

We now present a general framework that provides new insight into the problem of online scheduling with advance reservations in Grid environments. Our approach extends previous work in three directions: (1) it allows for general job deadlines (i.e., the deadline of a job j may take any value $d_j \geq r_j + l_j, \forall j$); (2) it provides the foundation for formulating a range of scheduling strategies based on a variety of optimization criteria; and (3) it leads to highly efficient algorithms for these strategies.

Let us return to the representation of idle periods as points in the plane that we illustrated in Figure 3.1. Assuming that the current time $t = 0$, Figure 3.2(a) shows the current schedule of advance reservations for a 3-server system, along with a request to schedule a new job j with the tuple $(r_j = t_6, l_j = t_8 - t_6, d_j = t_{12})$. Figure 3.2(b) is the geometric representation of this schedule. The fact that job j has a general deadline is represented in Figure 3.2(b) by the line segment between points P and P', where point $P = (r_j, r_j + l_j)$ (respectively, $P' = (d_j - l_j, d_j)$) corresponds to the earliest (respectively, latest) possible pair of starting and ending times for this job. Consequently, the scheduler may select *any* point on this line segment as the starting/ending times of the job, as long as there is an idle period completely containing

[1] We say that point $x = (x_1, x_2)$ completely contains point $y = (y_1, y_2)$ iff $x_1 \leq y_1$ *and* $x_2 \geq y_2$.

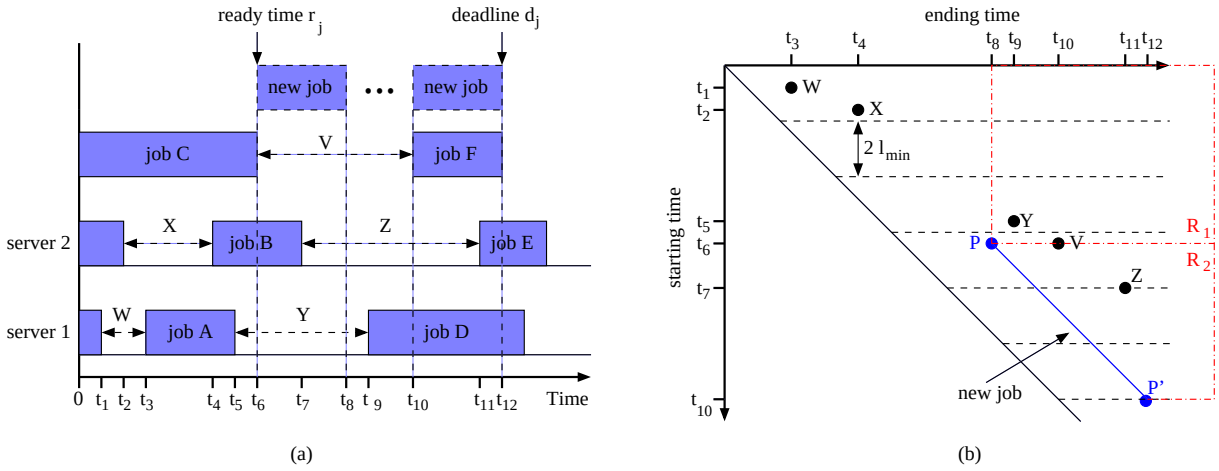


Figure 3.2: (a) Jobs scheduled and idle periods in a 3-server system, (b) idle periods as points in the plane, plane partitioned into strips of width $2 \times l_{min}$, and feasible regions R_1, R_2 for the new job

this point.

Consider the new job j and its geometric representation in the plane, as shown in Figure 3.2(b). The *feasible region* of job j refers to the part of the plane where all idle periods that can accommodate this job may lie. The feasible region is the part of the plane above and to the right of the line segment between P and P', since only any idle periods in that region will fully contain *some* point of the line segment. The feasible region can be partitioned into two subregions, R_1 and R_2, as in Figure 3.2(b). Any idle period lying in R_1 (e.g., idle periods Y an V in the figure) starts at or before the new job's ready time r_j ($= t_6$ in the figure), and ends after the earliest time the job can be completed ($= t_8$ in the figure). Therefore, any idle period in this region can accommodate the new job without delaying its execution, i.e., the job can start execution at its ready time r_j. Any idle period lying in R_2, on the other hand (e.g., idle period Z in Figure 3.2(b)), starts later than the job's ready time but is large enough for it. Hence, the job may be assigned to any idle period in R_2 at the cost of delaying its execution beyond its ready time.

3.2.1 Partitioning of the Idle Periods

Our objective is to obtain efficient algorithms for the online scheduling problem with general deadlines. We note that the work in [70] was developed for the special case of immediate deadlines. Recall also that the algorithm developed in [70] maintains a single priority search tree that contains all K points in the plane, i.e., all K idle periods on all servers. A single

tree structure is appropriate for immediate deadlines, in which case each job is represented by a *single* point in the plane. However, it cannot be directly applied to the more general problem we are considering, in which jobs are represented by a line segment, such as the one between points P and P' in Figure 3.2(b). With a single tree structure, the only way to handle a job with a general deadline is to perform multiple searches for multiple points along the line segment representing this job. Such an approach is inefficient if the points on the line segment are selected close to each other, since each search takes $O(\log K)$ time; whereas it may fail to find feasible idle periods if the points are selected far from each other to lower the worst-case running time.

In order to obtain efficient scheduling algorithms for the problem at hand, we partition the area of the plane above the diagonal into strips of width equal to twice the minimum job size l_{min}.Figure 3.2(b) shows the partitioning of the plane into *horizontal* strips. Alternatively, one might partition the plane into *vertical* strips of width $2 \times l_{min}$; the choice of direction depends on the optimization strategy selected, as we discuss shortly. Doing so in effect partitions the set of K idle periods into a number H of subsets, where subset $h, h = 1, \cdots, H$, contains the idle periods falling within the h-th strip.

Rather than maintaining a single tree data structure as in [70], we maintain H priority search trees, one for each strip. We also ignore (i.e., do not keep any information about) any idle period of length less than l_{min}, as it cannot be used for scheduling any job. Maintaining one tree structure for each strip is based on the observation that a given strip may contain *at most one idle period from each server*. To see that this is true, note that two consecutive idle periods on the same server must be separated by a job of length at least l_{min}, and that the length of each idle period is at least l_{min} (otherwise the idle period is discarded); therefore, the starting (and ending) times of two idle periods on any given server are at least $2 \times l_{min}$ time units apart from each other. In other words, the number of idle periods in a strip is bounded above by the number n of servers. Consequently, updating the schedule (i.e., adding or removing idle periods) takes time $O(\log n)$, rather than $O(\log K)$, where typically $n \ll K$.

Since each priority search tree structure contains only a subset of the set of idle periods, it may be necessary to search several trees to find a feasible idle period for a new job request[2]. Consider point P in Figure 3.2(b), representing the earliest time the new job may start execution.

[2]To improve the scalability of the algorithm, in terms of both running time and memory usage, we may partition the plane in strips of length $M \times 2 \times l_{min}$, where M is an integer greater than one. In this case, there will be no more than M idle periods from each server within each strip, or no more than nM idle periods in all. Consequently, the complexity of searching each tree becomes $O(\log(nM))$, or $O(\log M + \log n)$, but the number of strips (and corresponding trees) to be maintained decreases to H/M, where H is the number of strips for $M = 1$. Letting $M = n^k$, where k is a small integer, reduces the number of trees by a factor of n^k compared to the case $M = 1$, while the time to search each tree increases only by a factor of $k + 1$, i.e., becomes $O((k + 1) \log n)$.

In this example, the new job can be scheduled either in the idle period represented by point V or the one represented by Y. Point V can be found by searching the tree structure corresponding to the strip in which point P lies; however, if point V (i.e., the corresponding idle period) did not exist, one would have to continue searching strips above the one in which P lies (i.e., those with starting times earlier than the new job) in order to find an idle period (in this case, point Y) that would not delay the start of the job. On the other hand, if neither V or Y existed, the search would have to continue in strips below the one in which P lies, to identify idle periods (e.g., Z) that could accommodate this job at some starting time along the line segment from P to P'.

In addition to allowing the scheduler to handle jobs with general deadlines efficiently, the partition of idle periods into subsets also enables the natural implementation of a variety of strategies for selecting one among multiple feasible idle periods. This unique feature of our approach, due to its inherent flexibility in terms of partitioning the plane either horizontally or vertically, and in terms of the order in which the strips are searched, is discussed in detail in the next subsection.

3.2.2 Scheduling Strategies

We now describe a suite of scheduling strategies which make use of the approach we outlined in the previous subsection. These strategies are based on the observation that a job scheduled in an idle period will create at most two new idle periods: one between the start of the original idle period and the start of the job (the *leading idle period*), and one between the end of the job and the end of the original idle period (the *trailing idle period*). The creation of these new, smaller idle periods results in further fragmentation of the available capacity, and may prevent future job requests from being accommodated. Therefore, it may be desirable to schedule a new job within the idle period such that the size of either the leading or trailing idle periods created is optimized, since doing so is likely to increase the chances that future jobs will fit in these new idle periods.

To illustrate how the partitioning of the plane into strips can facilitate the implementation of such scheduling strategies, consider again the new job in Figure 3.2. This job can be accommodated by three idle periods, corresponding to points Y, V, and Z. Selecting either point V or point Z will result in a leading idle period of zero length (in fact, any point in the feasible region R_2 will have the same effect). On the other hand, selecting point Y in region R_1 will result in a leading idle period of length $(t_6 - t_5)$; furthermore, the higher up in region R_1 a

point lies, the larger the leading period that will be created if the job is assigned to it. Based on these observations, if the objective is to *minimize* the leading idle period, the search must start in strips within region R_2 first; if that fails, the search should continue with the bottom strip within region R_1, and proceed upwards until a feasible idle period is found. If, however, the objective is to *maximize* the leading idle period, then the search must start at the topmost strip of region R_1, and proceed downwards. Note also that while all points in region R_2 will result in a leading period of zero length, the later the starting time of a point the longer the execution of the new job will be delayed. This suggests that the strips of region R_2 should be searched from top to bottom to minimize the job turnaround time.

Similar observations can be made regarding the goal of optimizing the length of the trailing idle period created when scheduling a new job. This objective can be achieved by partitioning the plane in vertical strips (as opposed to the horizontal ones shown in Figure 3.2(b)), and following a similar search strategy.

The following strategies for the scheduling problem with general job deadlines arise naturally within this framework:

1. **Min-LIP**, which minimizes the leading idle period;
2. **Min-TIP**, which minimizes the trailing idle period;
3. **Best-fit**, which minimizes the sum of the leading and trailing idle periods;
4. **First-fit**, which returns the first (i.e., earliest) feasible idle period, regardless of the sizes of the leading and trailing idle periods.

We discuss the implementation of these strategies in the next section. We have also considered the maximization versions of the first two strategies (i.e., max-LIP and max-TIP), but due to space constraints we do not discuss them here.

3.3 Algorithm Description and Implementation

We now describe in detail the algorithms and related balanced tree data structures for implementing the min-LIP and best-fit scheduling strategies, and we analyze their worst-case running time. At the end of the section, we discuss the modifications required to implement the min-TIP and first-fit strategies.

3.3.1 Balanced Tree Structure for the Min-LIP Strategy

Recall from Section 3.2.1 that we partition the set of idle periods on all servers into H subsets, each subset corresponding to one of the horizontal strips in the geometric representation of the schedule of advance reservations (refer to Figure 3.2(b)) and consisting of the idle periods in this strip. Each subset is of size at most n, where n is the number of servers. The number H of subsets (equivalently, of horizontal strips) depends on how far in the future users are allowed to make advance reservations. For a given system, the value of H is fixed.

By construction, each subset $h, h = 1, \cdots, H$, contains all idle periods with starting times in the interval $[2(h-1)l_{min}, 2hl_{min})$. The idle periods in subset h are stored in a priority balanced search tree T_h; in our implementation, we use augmented red-black trees [73]. Whenever the scheduling algorithm (described in the next subsection) needs to search subset h to find an idle period for a new job, tree T_h is searched; as we explain shortly, the manner in which the tree is searched depends on the part of the feasible region (R_1 or R_2 in Figure 3.2(b)) in which the corresponding strip lies. The search of tree T_h will be unsuccessful if and only if no feasible idle period for the new job exists in this strip. Otherwise, the search will return a feasible idle period that optimizes a given objective; for the min-LIP strategy we are considering, it will return the idle period that will result in the minimum leading idle period among all feasible idle periods in the strip.

In tree T_h, the actual idle periods are in the leaf nodes, arranged in ascending order of their starting time. For the min-LIP strategy, a leaf node corresponding to idle period X stores the following information:

- the starting time of X;
- the ending time of X; and
- other auxiliary data, such as the identity of the corresponding server.

Internal tree nodes store information regarding the idle periods in their subtree. This information is used to navigate the tree and locate idle periods appropriate for the new job to be scheduled. In the case of the min-LIP strategy, the information stored in internal node v consists of:

- the median of the starting times of the idle periods stored in the subtree of T_h rooted at v;
- a pointer to the idle period in v's subtree with the latest ending time; and

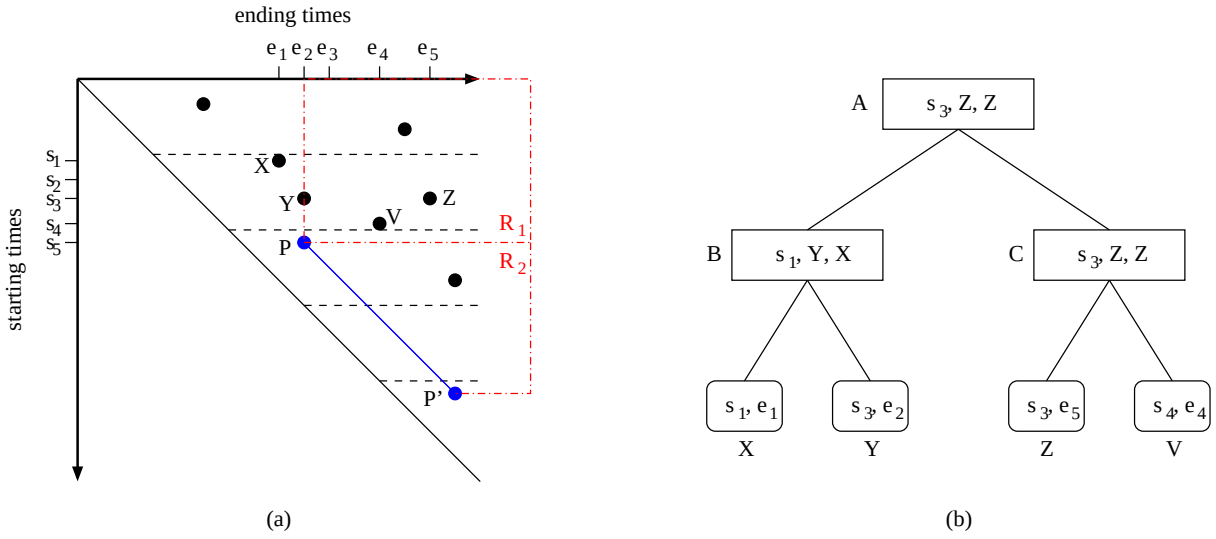


Figure 3.3: (a) Schedule of advance reservations, (b) balanced tree structure storing the idle periods in the second strip from the top

- a pointer to the idle period in v's subtree with the maximum length.

Figure 3.3(b) shows the balanced tree T_h associated with the second strip from the top of the schedule shown in Figure 3.3(a). This strip contains four idle periods with starting and ending times: $X = (s_1, e_1)$, $Y = (s_3, e_2)$, $Z = (s_3, e_5)$, and $V = (s_4, e_4)$. Since $s_1 < s_3 < s_4$, the idle periods are stored in this order as the leaves of the tree in Figure 3.3(b). Internal node B of the tree stores the median s_1 of the starting times of idle periods X and Y stored in its subtree, along with pointers to the idle period with the latest ending time (i.e., Y) and the largest one (i.e, X); similar information is stored in node C and the root A of the tree.

Note that as time advances, idle periods expire (i.e., their ending time passes) and must be discarded. Our approach of partitioning the plane into strips and maintaining a separate tree structure for the idle periods within each strip makes it easy to handle expired idle periods. Let us assume that the system starts operation at time $t = 0$, and that we maintain H strips, each of width $2l_{min}$. Since the scheduling horizon (i.e., the time in the future during which a job can be scheduled) is $H \times 2 \times l_{min}$ time units, then no idle period can end at time $t' > t + H \times 2 \times l_{min}$, where t is the current time. Consider the topmost strip with index $h = 1$. Initially, the latest time at which an idle period in this strip may end (expire) is at time $t' = (H+1) \times 2 \times l_{min} - \epsilon$, corresponding to the scheduling, at time $t = 2 \times l_{min} - \epsilon$, of a job with ready time $H \times 2 \times l_{min}$ time units in the future. Therefore, at time $t = (H+1) \times 2 \times l_{min}$, the tree corresponding to strip with index $h = 1$ is discarded, since all idle periods recorded in that tree have already expired. At the same time, all strips (and corresponding trees) with indices $h, h = 2, \cdots, H$,

are renumbered to $h' = h - 1$, and a new empty tree is created to record idle periods falling in the new strip with index $h' = H$. This discard operation is repeated every $2l_{min}$ time units thereafter. All the operations involved in discarding a tree can be performed in $O(1)$ time with no extra memory cost by using (1) a circular queue to record the tree indices, and (2) modulo-H arithmetic. If a single tree structure were used instead to store all idle periods, deleting expired idle times would require additional information to be kept at internal nodes, as well as costly periodic operations to locate all idle times with past ending times.

3.3.2 Min-LIP Algorithm

Consider a request to schedule a new job j with parameters (r_j, l_j, d_j). Let P and P' be the points in the geometric representation of the schedule that correspond to the earliest and latest times, respectively, at which the new job can be scheduled (refer also to Figure 3.3(a)). Let $p, 1 \leq p \leq H$, be the index of the horizontal strip in which point P lies; let $p' \geq p$ be the index of the strip where point P' lies. Similar to our earlier discussion, we also let R_1 (respectively, R_2) denote the part of the feasible region for the new job j containing idle periods with starting times earlier (respectively, later) than the job's ready time r_j.

The min-LIP algorithm to find a feasible idle period for the new job j that minimizes the length of the leading idle period created consists of two steps: a search in region R_2, followed by a search in region R_1, if necessary. Next, we describe these two steps in detail.

Step 1: Search in region R_2. The algorithm first searches for a feasible idle period in region R_2. Any such idle period has starting time $s \geq r_j$; hence, we schedule job j to start at time s, avoiding the creation of a leading idle period. Although any feasible idle period in this region is optimal in terms of the objective we consider, assigning the new job to an idle period with starting time s will delay the execution of the job by an amount of time equal to $s - r_j$ units beyond its ready time. In order to minimize this delay, the min-LIP algorithm explores the horizontal strips in this region in top-to-bottom fashion, i.e., by examining the corresponding trees in the order $T_p, T_{p+1}, \cdots, T_{p'}$.

The min-LIP algorithm exploits the observation that any feasible idle period in region R_1 is optimal in order to examine each tree $T_h, h = p, \cdots, p'$, in this region in $O(1)$ time. Recall that the root of T_h maintains a pointer to the largest idle period in the tree (refer to Figure 3.3(b)). If this idle period is smaller than the new job, then we know that no idle period in this tree can accommodate this job, and the algorithm proceeds to examine the next tree in the region; otherwise, the algorithm assigns the job to this largest idle period. Consequently,

each horizontal strip that contains no feasible idle period is eliminated in $O(1)$ time. At most one strip with a feasible idle period (the first such strip in the sequence) is examined, and the assignment of a job to the largest idle period in this strip takes time $O(1)$. In this case, the corresponding tree T_h must also be updated (to delete the largest idle period); this operation takes $O(\log n)$ time, where n is the number of servers in the system. If a trailing idle period that is larger than the minimum job size l_{min} is created, it has to be inserted in the appropriate tree (which may be different than T_h). Locating the appropriate tree from the trailing idle period's starting time takes constant time, and the insert operation takes $O(\log n)$ time. Since the number of strips that fall within region R_2 is at most $k = \lceil \frac{d_j}{2l_{min}} \rceil$, where d_j is the deadline of the new job, the worst-case running time of this step is $O(k + \log n)$ if the region contains a feasible idle period, and $O(k)$ if it does not.

Step 2: Search in region R_1. If Step 1 fails (i.e., no feasible idle period for the new job exists in region R_2), the algorithm proceeds to explore region R_1. If any feasible period in this region starting at time s is selected, the job will start execution at its ready time r_j, creating a leading idle period of length $r_j - s$. Since our goal is to minimize this length, the algorithm examines the horizontal strips in this region in bottom-to-top fashion, i.e., it searches the corresponding trees in the order $T_{p-1}, T_{p-2}, \cdots, T_1$. Note also that in this step of the algorithm we may safely ignore the line segment representing the job (e.g., the segment from point P to point P' in Figure 3.3(a)), and simply focus on the single point representing the job starting at its ready time (i.e., point P).

Each tree $T_h, h = p-1, \cdots, 1$, in region R_1 is searched using a standard algorithm for red-black trees [73] to find the idle period (if any) with the latest starting time that is large enough to accommodate the new job. This search takes time $O(\log n)$. If a feasible idle period is found in some tree T_h, three update operations must be performed: to delete the idle period from T_h, and to insert the newly created leading and trailing idle periods (as long as they are larger than l_{min}) into the appropriate trees; all these operations take $O(\log n)$ time [73, 71]. The number of strips within region R_1 is at most $m = \lceil \frac{r_j}{2l_{min}} \rceil$, where r_j is the ready time of job j. The worst-case running time of this step is $O(m \log n)$ and occurs when either no feasible idle period exists, or one exists in the topmost strip. Similarly, the worst-case running time of the overall algorithm is $O(k + m \log n)$.

Let us illustrate how the tree search algorithm operates by considering the second strip from the top in Figure 3.3(a), i.e., the one containing the idle periods X, Y, Z, and V. It is clear from the figure that only Y, Z, and V can accommodate the new job; of these, V is optimal in terms of minimizing the leading idle period for the job represented by point P, as it has the

latest starting time.

The algorithm starts at the root A of the tree in Figure 3.3(b) that stores the idle periods in this strip. It compares the ready time ($r_j = s_5$) of the new job j to the median ($= s_3$) of the starting times of the idle periods in this tree stored at the root. In this case, $s_3 < s_5$, which implies that some idle periods in the left subtree of A, as well as some idle periods in the right subtree, start before r_j, hence both subtrees may have to be examined further (if the reverse were true, the algorithm would have eliminated the right subtree of A immediately). The algorithm then compares the ending time of the job ($= e_2$) to the maximum ending time of the idle periods in the left subtree of A; this value ($= e_2$) can be obtained by following the pointer to the idle period Y with the maximum ending time that is stored in the root B of the left subtree. Since the two values are equal, a feasible idle period may exist for this job in the subtree rooted in B. Therefore, the algorithm *marks* node B for possible consideration in the future, and proceeds to examine the right subtree of A.

The search continues in a recursive manner until a leaf node is reached. In this example, the ready time ($r_j = s_5$) of the job is compared to the median starting time s_3 stored in node C. Since $s_3 < s_5$, the algorithm compares the ending time ($= e_5$) of the left child of C to the ending time e_2 of the job. Since $e_5 > e_2$, the idle period Z in the left child of C is feasible, and the algorithm marks the leaf node Z. It then similarly examines the right child of C, and determines that it also represents a feasible idle period; since this is the one with the latest starting time, it is optimal and is the one returned by the algorithm. In general, once the algorithm reaches a leaf node, all idle periods with starting time earlier than or equal to r_j are to its left. If the idle period represented by this leaf is feasible, then it is returned and the algorithm terminates. Otherwise, it is sufficient to continue the search recursively from the *last* marked node.

3.3.3 Tree Structure and Algorithm for the Best-fit Strategy

For the best-fit strategy, we use a 2-dimensional tree T_h to store the idle periods within each strip h, $h = 1, \cdots, H$. The tree corresponding to T_h's first dimension, t_h^s, is an augmented version of the min-LIP tree introduced earlier and the information stored at each of its internal nodes u consists of:

- the median starting time of the idle periods stored in the subtree of t_h^s rooted at u;
- a pointer to a secondary priority search tree t_h^e; and
- a pointer to a secondary regular binary search tree t_h^l.

Trees t_h^e and t_h^l store the idle periods in u's subtree in descending order of their ending time and length, respectively. The information stored at each internal node v of tree t_h^e consists of:

- the median ending time of the idle periods stored in the subtree of t_h^e rooted at v; and;
- a pointer to the idle period with minimum length in v's subtree.

As we explain shortly, the manner in which the data structure is searched depends on the part of the feasible region (R_1 or R_2) in which the corresponding strip lies.

The best-fit algorithm consists of two steps: a search for b_{R_1}, the local best fit in region R_1, followed by a search for b_{R_2}, the best fit in region R_2. After exploring both regions, the algorithm returns the overall best fit for the given job, if one exists. Since in this strategy the algorithm searches for a local best fit in every strip in order to obtain a global optimal, the order in which this search proceeds is irrelevant. However, for the sake of simplicity in our implementation we search both regions in a top-bottom fashion.

Step 1: Search in region R_1. Since the best-fit among a set of feasible idle periods is the idle period with the smallest length, the algorithm first identifies the set of feasible idle periods in the strip, and then retrieves the one with the smallest length. Recall also that all idle periods in R_1 start before r_j (see figure 3.2) and hence, meet the feasibility requirement in terms of their starting time. However, they may or may not be feasible depending on their ending time. To identify the set of feasible idle periods for a given job j in a strip in R_1, the algorithm searches the secondary tree associated with the strip t_h^e using a simplified version of the min-TIP algorithm. Min-TIP is similar to the min-LIP algorithm we just described with the difference being that the search performed is a function of the ending time. More specifically, the algorithm visits every internal node v in t_h^e whose subtree contains *exclusively* idle periods with ending time larger than the earliest time the job can be completed; the algorithm stops as soon as it reaches a leaf. In terms of complexity, the same arguments presented earlier for min-LIP hold for min-TIP, therefore, the cost of visiting the $O(\log n)$ internal nodes is $O(\log n)$ per strip.

For each internal node v visited in tree t_h^e the algorithm computes the local best fit b_v corresponding to the idle periods in v's subtree. Such an idle period is the smallest idle period in v'subtree and can be retrieved by means of the pointer stored at v at a cost of $O(1)$. The algorithm then compares b_v to the most up to date b_{R_1} at that particular point in time; if b_v has a smaller length it updates b_{R_1} with b_v, otherwise, it discards b_v. Recall that retrieving b_v from a given v's subtree costs $O(1)$; therefore, the overall cost for searching b_{R_1} is $O(m \log n)$

where m is the number of strips in R_1 and is at most $m = \lceil \frac{r_j}{2l_{min}} \rceil$.

Step2: Search in region R_2. After the algorithm has searched for b_{R_1} it proceeds to search b_{R_2} in R_2. Notice that as an idle period in R_2 moves further up (down) from the line segment between P and P' its length increases (decreases), until it reaches the line segment itself where the length of the idle period is l_j. It follows that the best fit in a strip in R_2 is the closest idle period to the line segment between P and P'. To find such idle period the algorithm performs a simple binary search on tree t_h^l. More specifically, it searches for the idle period with the minimum length larger than the length of the job, l_j. Since in R_2 there are at most $k = \lceil \frac{d_j}{2l_{min}} \rceil$ strips, the overall complexity for searching b_{R_2} in Step 2 is $O(k \log n)$.

3.3.4 Implementation of Other Scheduling Strategies

The scheduling strategies we defined in Section 3.2.2 can be implemented by appropriately modifying either the tree data structure or the search algorithm we described above for the min-LIP strategy. In order to optimize the trailing idle period, the plane must be partitioned into vertical strips of length $M \times 2 \times l_{min}, M \geq 1$, and each tree must store the idle periods in the corresponding strip in increasing order of their ending, rather than starting, times; the search algorithm is similar to the corresponding algorithm for min-TIP. Finally, the first fit strategy can be implemented by exploring the horizontal strips in increasing order of index h, and selecting from each tree the first feasible idle period found.

3.4 Performance Evaluation

We use simulation to evaluate the performance of the various scheduling strategies. We use the method of batch means to estimate the performance parameters we consider (and which we discuss shortly), with each batch consisting of thirty simulation runs and each run lasting until 10^6 jobs have been submitted to the Grid scheduler. We have also obtained 95% confidence intervals for all the results, which are shown in the figures.

In our simulation, we assume that job requests arrive as a Poisson process with rate λ. Job sizes are distributed according to a bounded Pareto distribution. The minimum job size is set equal to 1, and is taken as the unit of time. The maximum job size is set to 50 time units, and we vary the mean job size $\bar{x}$ by changing the value of the parameters of the Pareto distribution. We let L denote the amount of time that the scheduler $\mathcal{S}$ can look "into the future"; in other words, a job may request to be scheduled at most L units of time in the future. We let the

deadline d_j of job j be uniformly distributed in the interval $(r_j + l_j, r_j + l_j + q(L - r_j - l_j))$, where $q, 0 \leq q \leq 1$ is a parameter that controls the "tightness" of the job deadlines. In our simulations, we let $L = 200$.

We use four performance metrics in our study. The *loss rate* is the fraction of jobs that are dropped due to the fact that their deadline cannot be met. The *system utilization* is the fraction of time the n servers are busy serving jobs. The *average delay* is the mean amount of time that a job has to wait beyond its ready time until it starts execution; note that dropped jobs do not contribute to the average delay. Finally, the *fairness ratio* is a measure of how fairly jobs of different sizes fare in terms of drop probability under a given scheduling algorithm. To compute the fairness index, we partition the domain $[1, 50]$ of the job size distribution into $B = 100$ bins of equal size. Let z_i be the number of arriving jobs that fall into the i-th bin during a certain simulation run, and let $z_i' \leq z_i$ be the number of these jobs that are scheduled successfully. Let w_i be the fraction of jobs in the Pareto distribution that fall in the i-th bin. The fairness index F of a scheduling strategy is calculated as:

$$0 \quad \leq \quad F \quad = \quad \sum_{i=1}^{B} w_i \, \frac{z_i'}{z_i} \quad \leq \quad 1. \tag{3.1}$$

Clearly, the closer the value of the fairness index is to one, the more fair the scheduling discipline is.

We compare five scheduling strategies: first-fit, min-LIP, min-TIP, best-fit and LACT. The LACT algorithm, which we described in Section 3.1, does not consider the idle periods created at each server, and hence suffers the effects of capacity fragmentation; we consider this algorithm as a baseline case. Although we do not show any results for the max-LIP and max-TIP scheduling algorithms, their overall behavior is similar to that of min-LIP and min-TIP in that they are efficient in utilizing the available system capacity.

Figures 3.4-3.5 plot the loss rate, utilization, average delay, and fairness ratio, respectively, for the five scheduling strategies against the system load ρ. The system load is calculated using the familiar from queueing theory expression $\rho = (\lambda \bar{x})/n$. For the results shown in these figures, we let the number of servers $n = 20$, the mean job size $\bar{x} = 3.28$, and the tightness of the job deadlines $q = 0.1$. Note that the load values in the figures range from low ($\rho = 0.1$) to very high ($\rho = 1.1$) at which the system is more than 100% loaded. Also, the 95% confidence intervals are quite narrow for all curves shown.

From Figure 3.4 we can see that the loss rate increases with the system load for all five scheduling algorithms, as expected. However, the LACT algorithm performs significantly worse

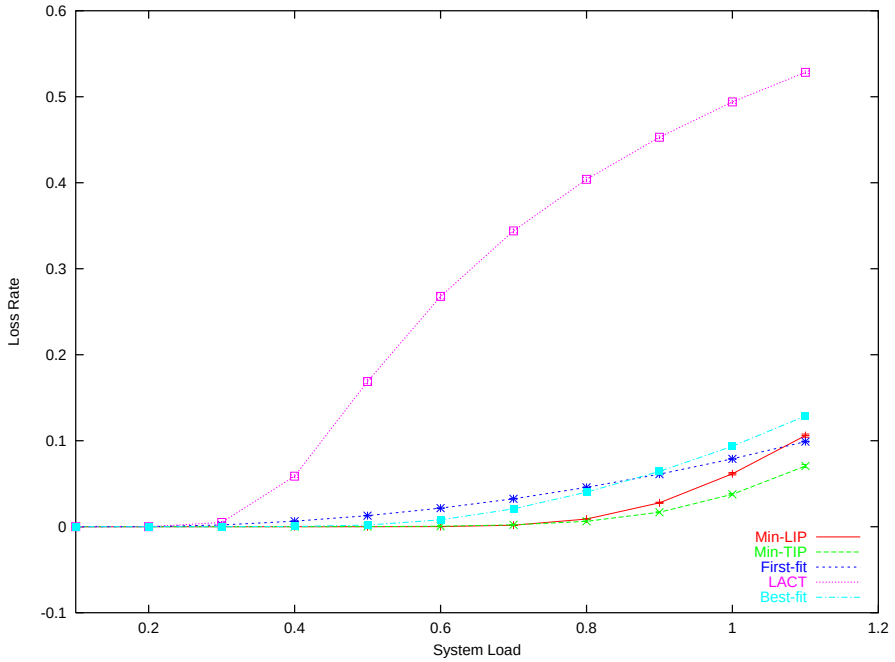


Figure 3.4: Loss rate against system load ρ, $n = 20, \bar{x} = 3.28, q = 0.1$

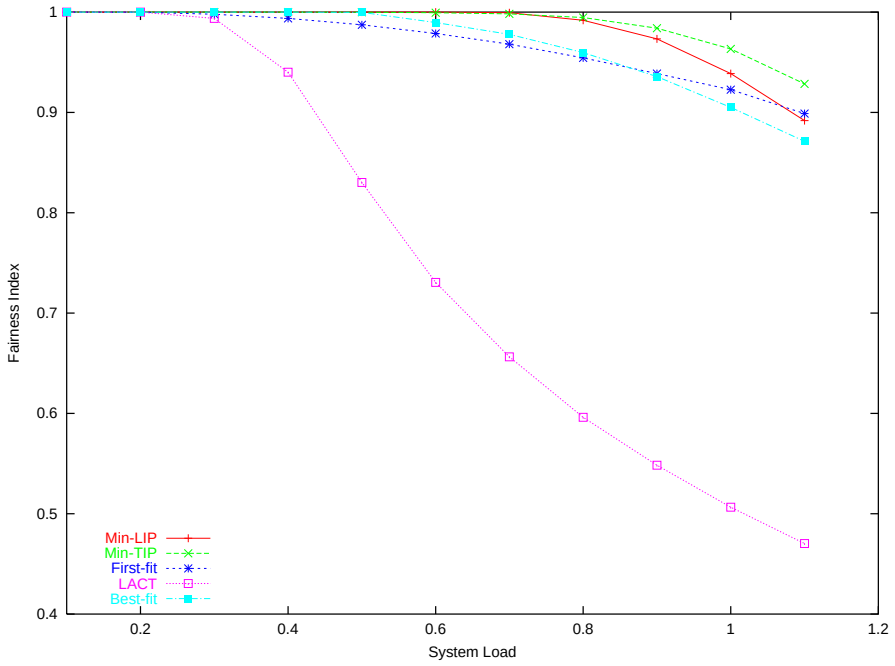


Figure 3.5: Fairness ratio against system load ρ, $n = 20, \bar{x} = 3.28, q = 0.1$

than the other four strategies at all but very low loads; this result is not surprising given the fact that this algorithm does not consider the idle periods in the servers. Under the other four

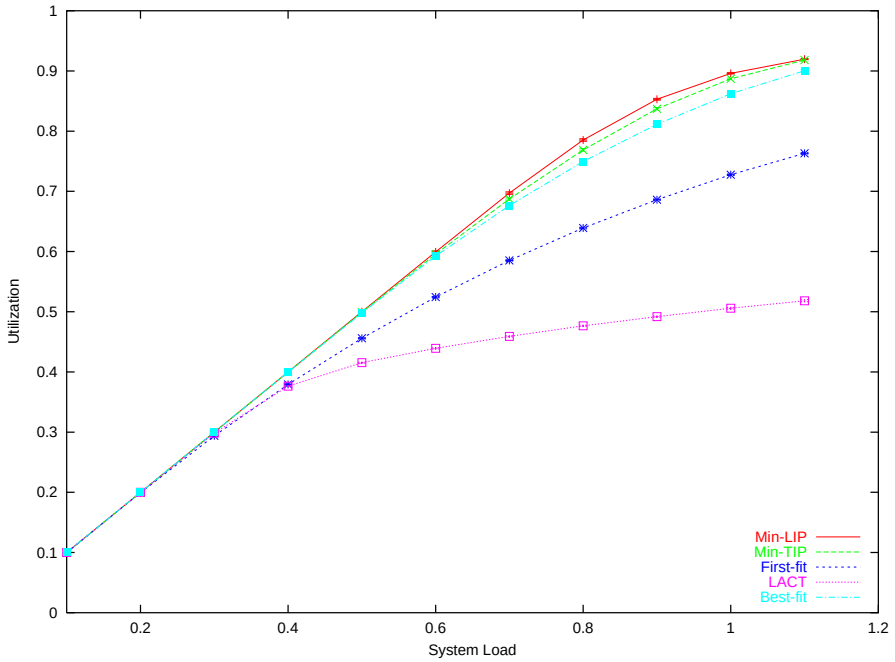


Figure 3.6: Utilization against system load ρ, $n = 20, \bar{x} = 3.28, q = 0.1$

strategies, jobs experience low loss rates even for load values close to 1; in fact, min-LIP and min-TIP have almost identical behavior with loss rates close to zero for loads up to $\rho = 0.8$. The best-fit strategy experiences low loss rate but performs slightly worse than min-LIP and min-TIP. This can be explained by the fact that in the best-fit strategy jobs are scheduled to start execution at the starting time of the idle period whenever possible. This results on the creation of small trailing idle periods, which may fail to accommodate incoming jobs; hence increasing the loss rate in the system. The first-fit algorithm also experiences low loss, but it performs worse than min-LIP or min-TIP for all load values less than 1. Therefore, min-LIP and min-TIP are clearly the best algorithms for typical operating regimes (i.e., at medium to medium-high loads). Note also that the loss rate for two algorithms is less than 10% even at a load of $\rho = 1.1$. This result can be explained by the fact that when the system is overloaded, large jobs have higher probability to be dropped than small jobs, under these two algorithms; hence at $\rho = 1.1$, the dropped jobs account for more than 10% of the offered load.

Figure 3.6, which plots the system utilization versus the load, confirms our observations regarding the relative performance of the five algorithms. As expected, utilization increases with the system load initially, but at some point the curves level off. LACT shows the lowest utilization, a result consistent with the high loss rates we observed in Figure 3.4. Min-LIP, best-fit and min-TIP again have the best performance, followed by first-fit. Moreover, followed

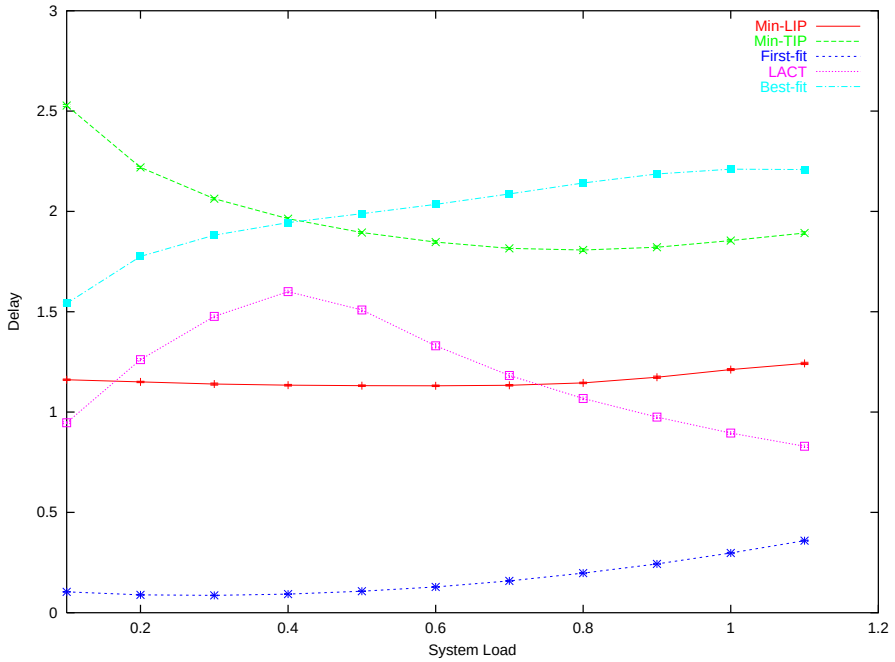


Figure 3.7: Average delay against system load ρ, $n = 20, \bar{x} = 3.28, q = 0.1$

closely by the best-fit curve, the behavior of the min-LIP and min-TIP curves is almost identical, with utilization increasing almost linearly with the load values. This result indicates that all three algorithms are capable of identifying and using idle periods to schedule jobs, thus ensuring that fragmentation of system capacity does not compromise overall performance. We also note that the difference in utilization between first-fit, on the one hand, and min-LIP, min-TIP and best-fit, on the other hand, is higher than the difference in loss rates would suggest. The higher difference in utilization can be explained by the fact that the first-fit strategy is less fair than the other three, and tends to drop larger jobs with higher probability; we will discuss this fairness issue in more detail shortly.

Let us now turn to Figure 3.7 which plots the average job delay against the system load. As we can see, jobs experience the lowest delay under the first-fit strategy. This result agrees with intuition: first-fit assigns a new job to the earliest feasible idle period, thus minimizing delay. The best-fit strategy, on the other hand, results in high delays for moderate to high loads. This is consistent with the results obtained for the loss rate (see Figure 3.4). We also observe that the average delay for min-LIP is higher than for first-fit but lower than under min-TIP. Recall that min-LIP first searches for the earliest feasible idle period in region R_2 (i.e., for an idle period starting after the job's ready time). Once such an idle period is found, the job is scheduled to start at the beginning of this period. Consequently, the starting time of the job

can be no earlier than under first-fit, hence the longer delay. On the other hand, min-TIP also searches first for the earliest idle period starting after a job's ready time. But unlike min-LIP, it schedules the job at the end of this idle period; shifting the job so that its completion time coincides with the end of the idle period causes higher delay than min-LIP. The average delay curve for the LACT algorithm lies between the corresponding curves for min-LIP and min-TIP for most system load values of interest. Note that the average delay for LACT increases up to $\rho = 0.4$, at which point LACT losses start to accelerate (refer to Figure 3.4). Beyond that point, average delay under LACT starts to decrease; however, this behavior is a side effect of the high losses incurred, rather than an indication of an inherent quality of the algorithm.

Overall, the average delay values in Figure 3.7 are relatively low, and correspond to a fraction of the mean job size $\bar{x} = 3.28$ for all algorithms. More importantly, average delay for the four strategies of interest (i.e., first-fit, min-LIP, min-TIP and best-fit) does not vary significantly with load, although it increases slightly at high loads. One exception is the min-TIP strategy which shows a moderate decrease in delay as ρ increases from low to moderate values. This behavior can be explained as follows. At low loads, min-TIP can find feasible idle periods starting after the jobs' ready time, and shifts the jobs to the end of these idle periods incurring a relatively high delay. At higher loads, on the other hand, and due to the relatively tight deadlines, it becomes more difficult to find such idle periods. In case of failure, min-TIP (similar to min-LIP) then searches for feasible idle periods that start before the jobs' ready time. Since these idle periods start earlier, the average delay under min-TIP tends to decrease with the load.

Figure 3.5 plots the fairness index, calculated by expression (3.1), against the system load. As we can see, the fairness index of the LACT algorithm suffers a precipitous drop starting at $\rho = 0.4$, the point where its losses begin to accelerate. This increase in unfairness is primarily due to the fact that larger jobs experience a significantly larger drop probability than smaller ones. The first-fit strategy is more fair than LACT, but it starts being unfair to jobs of larger size at loads as low as $\rho = 0.4$, compared to min-LIP and min-TIP; as a result, its utilization of the system capacity starts suffering from that point, as illustrated in Figure 3.6. Similarly, the best-fit strategy becomes less fair than the first-fit strategy as the load increases. The min-LIP and min-TIP strategies, on the other hand, achieve fairness index values close to 1 even at high system loads, with min-TIP slightly outperforming min-LIP. The fact that min-LIP and min-TIP remain fair across a wide range of load values is an important property of these algorithms, and indicates that they are capable of exploiting the idle periods in an effective manner. Moreover, their ability to treat all jobs fairly implies that users will not need to employ strategies such as

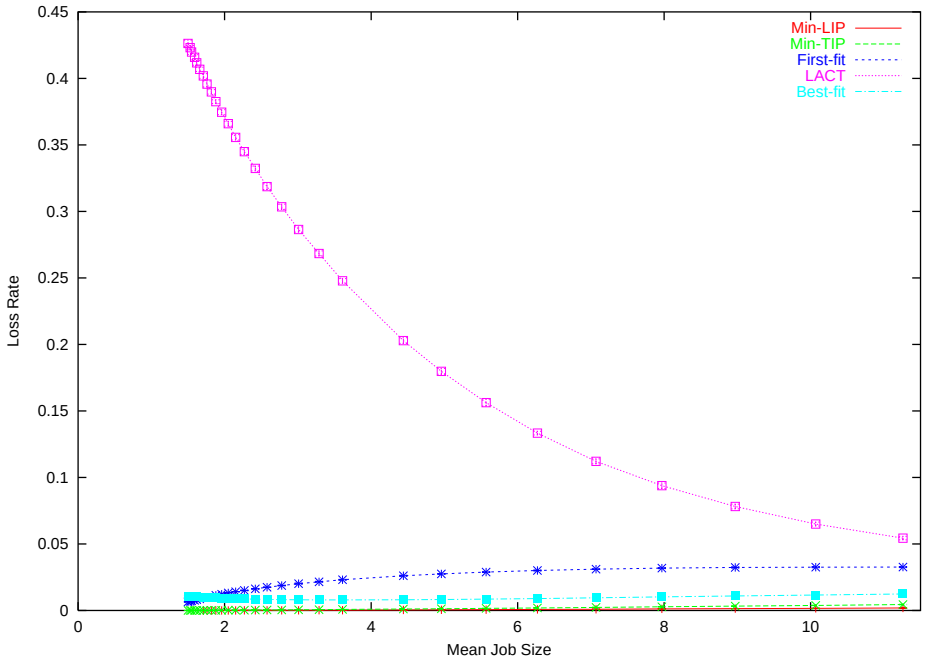


Figure 3.8: Loss rate against mean job size $\bar{x}$, $n = 20, \rho = 0.6, q = 0.1$

splitting a large job into several smaller ones, to avoid discrimination. Note that such strategies impose an additional overhead to the system in the form of additional memory usage (needed to store the additional idle periods created) and higher running time (due to the larger number of jobs requests, each request needing to search a larger data structure).

In addition to providing insight into the relative behavior of the five strategies due to the different optimization objectives considered, Figures 3.4-3.5 illustrate that properly designed scheduling algorithms can effectively overcome the obstacles of capacity fragmentation to deliver high performance in terms of metrics that reflect the requirements of both users and service providers. Specifically, the min-LIP and min-TIP algorithms cater to the user needs by ensuring that job deadlines are met in a fair manner while keeping both loss rates and average delay low; at the same time, they deliver high system utilization, an important goal for service providers.

The next three Figures 3.8-3.9 illustrate the behavior of the loss rate as we vary the values of three important system parameters, namely, mean job size $\bar{x}$, deadline tightness q, and number of servers n, respectively; the other parameters in the experiments take values as specified in the corresponding figure caption.

Figure 3.8 plots the loss rate for the five scheduling algorithms against the mean job size for $n = 20$ servers and system load $\rho = 0.6$. Min-LIP and min-TIP clearly outperform the other three algorithms, and their loss rate remains well below 1% across the range of mean

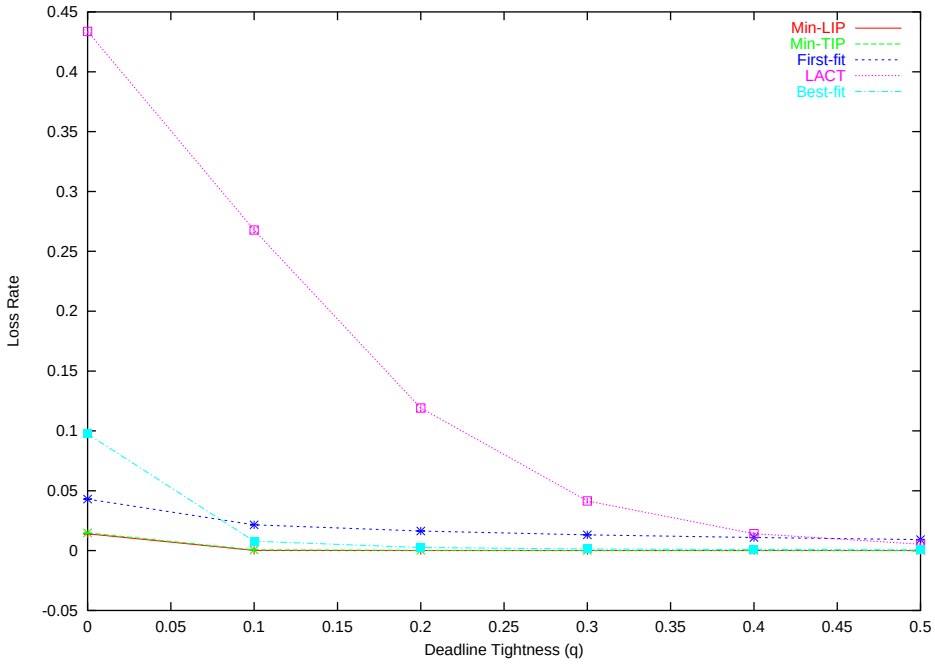


Figure 3.9: Loss rate against deadline tightness q, $n = 20, \bar{x} = 3.28, \rho = 0.6$

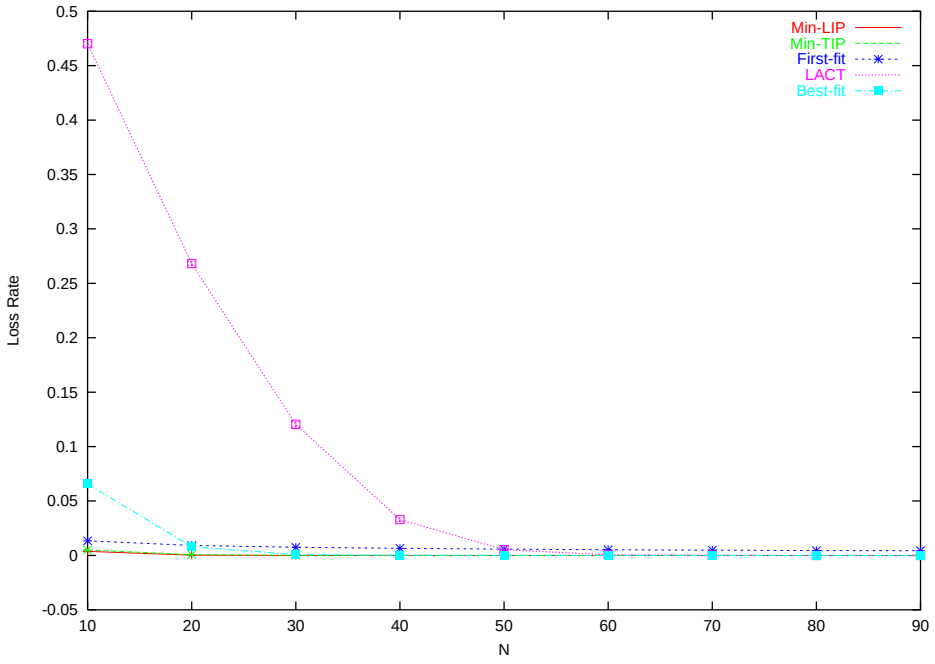


Figure 3.10: Loss rate against number of servers n, $\bar{x} = 3.28, \rho = 0.9, q = 0.1$

job size values shown in the figure; the performance of best-fit is also close. In fact, mean job size has little effect on the loss rate for these algorithms. We have also found that utilization

remains close to 60% for these two algorithms, and the fairness index close to 1. First-fit has a higher loss rate, which increases with the mean job size. Furthermore, we have found that the unfairness of first-fit also increases with the mean job size, to the degree that system utilization drops much more than the loss rate suggests, and in fact, it drops below the utilization of the LACT algorithm for $\bar{x} > 6$. Finally, the loss rate of LACT is the highest, but it *decreases* as $\bar{x}$ increases. While this behavior may seem counter-intuitive, it can be explained by noting that for constant load, increasing $\bar{x}$ implies a lower job arrival rate. Fewer job arrivals result in fewer idle periods, hence a lower degree of fragmentation of the available capacity. Since LACT performs worse with increasing degree of fragmentation, its performance improves as the mean job size increases.

In Figure 3.9 we plot the loss rate against the deadline tightness q. Recall that the larger the value of parameter q, the further in the future the deadline of each job lies, and the more flexibility an algorithm has in scheduling jobs. As we can see in the figure, the loss rate of the min-LIP and min-TIP strategies decreases as the value of q increases from 0 (the case of immediate deadlines) to 0.1; after that point, the loss rate remains at zero. The loss rate of best-fit and first-fit also decreases initially, and then remains low throughout the range of values of q. This behavior indicates that these four policies, which consider the idle periods when scheduling jobs, are effective throughout the range of deadlines considered in our study; their performance is affected, although not significantly, only when deadlines are very "tight." On the other hand, it is evident that the LACT algorithm is very sensitive to the tightness of the deadlines: its performance is poor when q is small, but it improves dramatically as the value of q increases, in which case the algorithm can push the starting time of jobs further in the future without missing their deadlines. Of course, this improvement in performance comes at the expense of significantly higher delay (not shown here due to space constraints).

Finally, Figure 3.10 plots the loss rate against the number n of servers in the Grid. The relative behavior of the various curves is similar to the one observed earlier: min-LIP and min-TIP clearly outperform the other four strategies and have loss rates close to zero at larger values of n, while LACT has by far the worse performance. In general, the loss rate decreases with the number of servers for all strategies, but shows a significant improvement for LACT. This behavior can be explained by noting that at constant load, as the number of servers increases, the degree of fragmentation tends to decrease, hence the performance of LACT improves. We also emphasize that the loss rate for LACT is an order of magnitude higher than the loss rates of all three; min-LIP, min-TIP and best-fit throughout the values of n used in this experiment.

3.5 Concluding Remarks

We have applied techniques from computational geometry to develop a suite of scheduling strategies that allocate resources in a Grid environment using a range of optimization criteria. We also presented efficient implementation of the various algorithms that scale to large Grid systems. We have presented results from extensive simulation experiments to demonstrate that our algorithms are simultaneously user- and system-centric: they are able to schedule resources to meet the deadlines imposed by users and maximize system utilization, while experiencing low job drop rates and low delays. Our algorithms also allocate resources to users in a fair manner. Our work provides a practical and efficient solution to the problem of scheduling resources in the emerging highly dynamic Grid environments.

Chapter 4

Efficient Scheduling Algorithm for Grids - heterogeneous

In the previous chapter we developed efficient algorithms for advance reservations in *homogeneous* Grids. These algorithms are effective in meeting time requirements (e.g., deadlines), may be adapted to employ several optimization criteria for scheduling jobs, and their low running times make them practical for large Grid environments. In this chapter we address the issue of meeting application time requirements in Grid environments with resources of *heterogeneous* capabilities (e.g, as in the case of computation servers with varying processing power).

We consider an environment where users submit jobs dynamically, and these jobs may start at a future time and must be completed within a certain deadline. We first investigate the impact of heterogeneity on the scheduling of resources, and conclude that heterogeneity of resources needs to be considered in order to achieve appropriate system and user performance. Based on this observation, we then develop an efficient heterogeneity-aware scheduling algorithm for advance reservations. We also describe how to apply techniques from computational geometry to develop data structures that allow the service provider to manage efficiently the set of advance reservations and handle effectively the resulting resource fragmentation.

Previous work in scheduling have considered resource heterogeneity in the context of Grids. In [48, 49, 59] authors show how laxity and fuzziness in the reservation requests may be exploited to address some of the drawbacks of advance reservations. Two of the most recent major works on advance reservations in Grids are [50] and [35]. In [50], the authors propose a multi-objective genetic algorithm formulation for selecting the set of resources to be provisioned that optimizes the application performance while minimizing the resource costs. In [35] a cost-

aware resource model is presented in which reservation for each application task is performed separately by negotiating with the resource provider. In [51] the authors present a broker service for the Grid resources that takes into account the fact that deadline and budget are specified, and then optimizes the usage of resources only by considering the current state of the resources but without any planning horizon.

The impact of resource heterogeneity has been investigated in contexts other than Grids. In [74] the authors exploit the heterogeneity found in an HPC (High Performance Computing) environment by dividing a task into subtasks and then mapping the latter to resources that best meet their requirements. This work assumes offline scheduling and does not support advance reservations; our work deals with online scheduling and allow users to schedule jobs in advance. In [75] the authors proposed a general framework to quantify the worst-case effect of increasing heterogeneity in models of parallel systems with finite total capacity. An important contribution of this work was a model to characterize resource heterogeneity which we adopt in our work.

The rest of the chapter is organized as follows. In Section 4.1 we describe the online scheduling problem being considered. In Section 4.2 we make a case for heterogeneity-aware algorithms in Grids. We perform a simple experiment to show that resource heterogeneity have a positive impact in system and user performance when scheduling algorithms are designed to accommodate for resource heterogeneity. In Section 4.3 we use novel techniques from computational geometry to represent the scheduling problem and facilitate the design of an efficient heterogeneity-aware scheduling algorithm. We provide details on the design and implementation of a heterogeneous-aware scheduling algorithm in Section 4.4. Several directions for further improving the performance of the scheduling algorithm that are subject of ongoing research are presented in Section 4.5. In Section 4.6 we evaluate the performance of our algorithm through simulation, and we conclude in Section 4.7.

4.1 Problem Description

Consider a scheduler $\mathcal{S}$ for a Grid with n servers which may be geographically distributed in a network. We consider a heterogeneous environment in that server i has service rate μ_i, where service rate refers to the amount of work a server can perform per unit of time. We also assume network delays are negligible. A user with job j requiring service submits a request to the scheduler. The request is characterized by a three-parameter tuple (r_j, l_j, d_j), where:

1. r_j is the *ready time* of the job, i.e., the earliest time the job can be made available to the Grid for processing;

2. l_j is the *size* of the job, i.e, the amount of work the job requires; and

3. $d_j (\geq r_j + l_j)$ is the *deadline* of the job, i.e., the latest time by which the job can be completed to provide any utility to the user.

The deadline is a measure of the quality of service required by the user. We assume that deadlines are *hard*, in that a user receives utility only if the job completes service by its deadline. Therefore, if $\mathcal{S}$ determines that the deadline cannot be met, it drops the job and notifies its user accordingly. Note that this restriction may be relaxed with minimal modifications to our algorithm; in Section 4.5 we describe a set of mechanisms that may be used to re-negotiate and re-plan advance reservations in order to minimize the number of jobs that are dropped.

In our model, the availability of resources is represented by time intervals during which servers are *idle*. We refer to these intervals as *idle periods* in this paper. We say that an idle period is *feasible* for a given job j if it can accommodate j within its deadline d_j. The feasibility of an idle period k for a given job j is determined by both the service rate of the server associated with the idle period and its duration. Therefore, we characterize an idle period k on a server i with service rate μ_i by a three-parameter tuple (st_k, et_k, c_k), where:

- st_k is the starting time of the idle period;

- et_k is the ending time of the idle period; and

- $c_k = \mu_i \times (et_k - st_k)$ is the *nominal capacity* of the idle period, i.e., the amount of work that server i can perform during idle period k.

Note that idle periods in slow (respectively, fast) servers may have a long (respectively, short) duration but small (respectively, large) nominal capacity. Moreover, the nominal capacity c_k of an idle period k represents the *maximum* job size that it can accommodate, assuming that the job is scheduled to start execution exactly at time st_k. As time progresses, the nominal capacity c_k of the idle period decreases at a rate equal to its server's rate μ_i. Consequently, if no job is allocated to the idle period by time $t = st_k$, then the maximum job size that it can accommodate decreases linearly at rate μ_i. Therefore, the nominal capacity of idle periods belonging to fast (respectively, slow) servers expires at a faster (respectively, slower) rate.

We consider the online scheduling problem whereby users submit service requests to $\mathcal{S}$ at random instants. We assume that $\mathcal{S}$ maintains a schedule which records, for each server

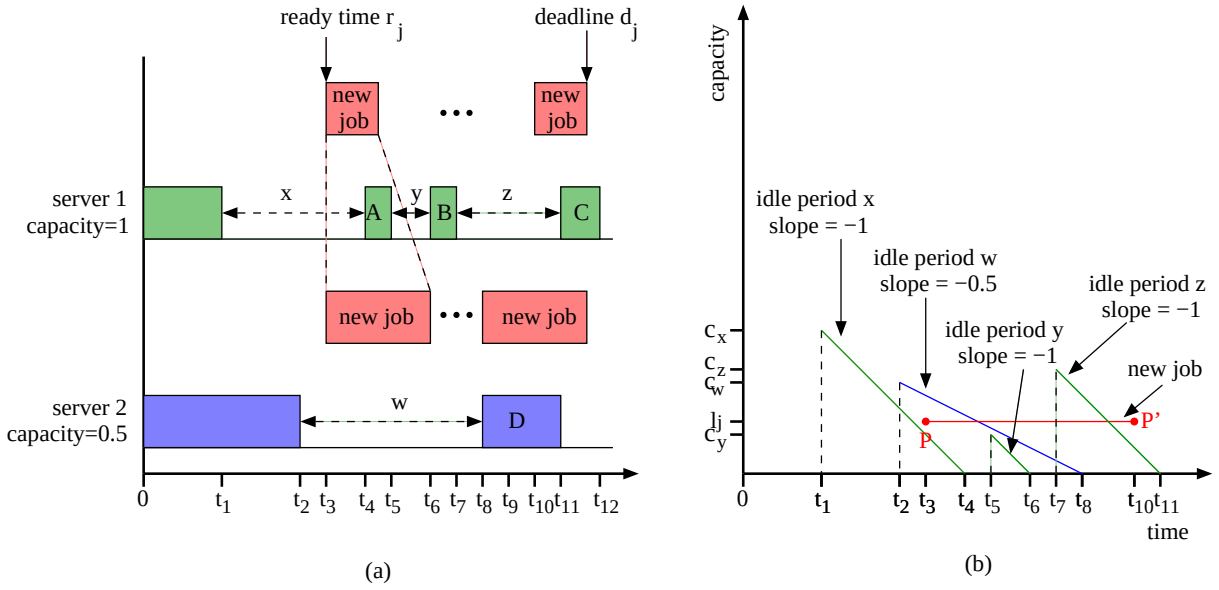


Figure 4.1: (a) Schedule of a 2-server system as a timetable, and (b) geometric representation of the idle periods and the new job.

i, the time periods in the future during which the server is reserved for jobs that have already been accepted to the system. In essence, this schedule represents the set of *advance reservations* that have been made, and it guarantees that server resources will be available to the accepted jobs at specific future times.

Figure 4.1(a) shows an example schedule for a 2-server system in which server i has rate $\mu_1 = 1$, and server 2 has rate $\mu_2 = 0.5$. The schedule is in the form of a *timetable*, and shows that at the current time (i.e., $t = 0$), there are four jobs scheduled for server 1: the job currently in service which will end at time t_1, job A which has reserved the server from time t_4 to time t_5, job B which has reserved the server from time t_6 until time t_7, and job C which is scheduled from time t_{11} to time t_{12}. Similarly, there are two jobs scheduled for server 2. The figure also shows a new job j requesting service. The job has ready time $r_j = t_3$ and deadline d_j. There are two representations of the new job. The representation at the top has a shorter duration and shows the new job as seen by server 1, while the one below has a longer duration (i.e., double that at the top) and shows the job as seen by server 2.

When a service request (r_j, l_j, d_j) for a new job j arrives, $\mathcal{S}$ immediately runs an algorithm to determine whether it is feasible to schedule the job so as to meet its deadline. If so, then $\mathcal{S}$ uses a set of criteria to select one of the (possibly multiple) servers that can handle this job, updates its schedule, and returns a reference to this server to the user; otherwise, the job is dropped. The scheduling decision impacts the performance perceived by users as reflected

by the fraction of jobs meeting (or missing) their deadlines and the response time of the jobs. It also impacts the overall system performance as reflected by the system utilization, which is a measure of how well the overall service capacity of the system is used. The challenge, therefore, is to develop efficient online scheduling algorithms that minimize the fraction of dropped jobs while maximizing utilization.

4.1.1 Computational Heterogeneity

To incorporate computational heterogeneity into our framework we use the model introduced in [75]. In this model the authors use *majorization* partial order to compare the imbalance, i.e., heterogeneity, of capacity distributions. The *majorization* partial order, $\succeq$, is defined as follows. Given two nonnegative vectors corresponding to the service rates of two n-servers systems $C = (\mu_1, \mu_2, \mu_3, \cdots, \mu_n)$ and $C' = (\mu'_1, \mu'_2, \mu'_3, \cdots, \mu'_n)$, we have $C' \succeq C$ when

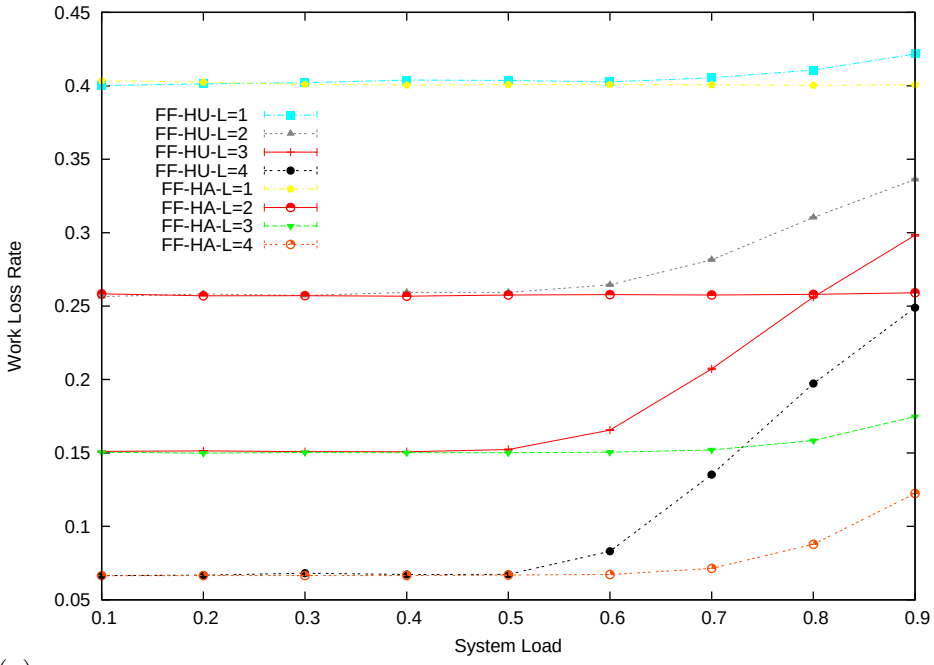
$$\forall k \sum_{i=1}^{k} \mu'_{[i]} \geq \sum_{i=1}^{k} \mu_{[i]} \text{ and } \sum_{i=1}^{n} \mu'_i = \sum_{i=1}^{n} \mu_i \tag{4.1}$$

where $\mu_{[i]}$ denotes the i-th largest component of C. We say that the computational capacity distribution C_A of a system A is more heterogeneous than the computational capacity C_B of a system B whenever $C_A \succeq C_B$.

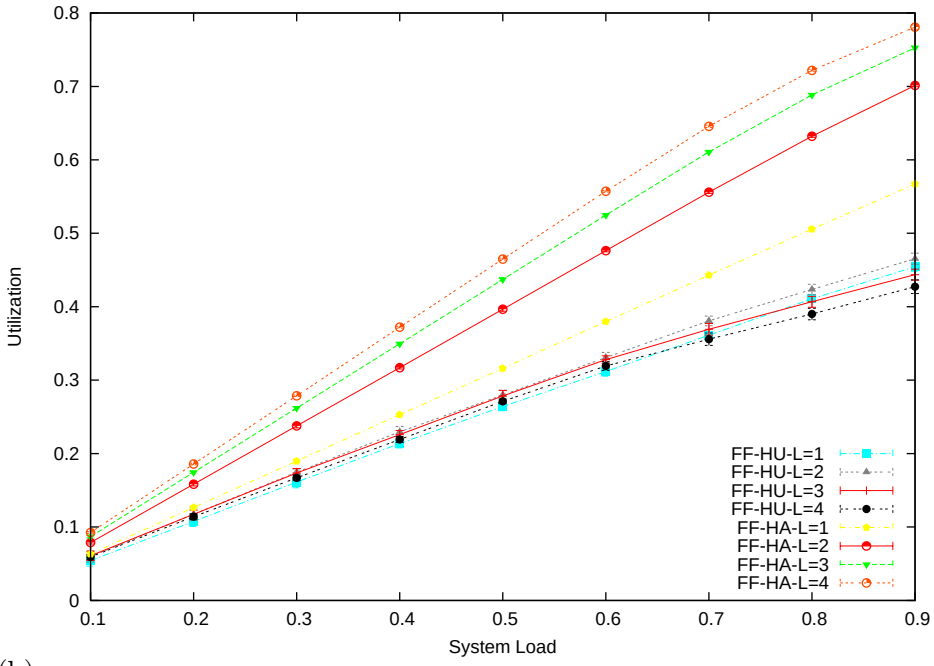
We say that a Grid system is (H, n)-heterogeneous, $H \ll n$, if the n servers are partitioned in H groups such that servers in group $h, h = 1, \cdots, H$, have the same service rate μ_h. Note that most existing Grids follow this model as they consist of a collection of clusters of identical processors. Thus, an (H, n)-heterogeneous system has n servers with H different rates. For a given (H, n)-heterogeneous system we may generate a range of service rate distributions that are more or less heterogeneous according to the majorization partial order in expression (4.1). We let L denote the levels of heterogeneity, i.e., the number of service rate distributions considered for a (H, n)-heterogeneous Grid, labeled in order of increasing heterogeneity:

$$(H, n)_L \succeq \cdots \succeq (H, n)_1 \succeq (H, n)_0 \tag{4.2}$$

where we use $(H, n)_0$ to denote the completely homogeneous system, i.e., one in which all n servers have the same rate μ. We use this model in the experimental studies we report in Sections 4.2 and 4.6.



(a)



(b)

Figure 4.2: Comparison of heterogeneous aware and unaware algorithms: (a) Work loss rate and (b) utilization against system load

4.2 The Case for Heterogeneity-Aware Algorithms

To investigate the impact of heterogeneity in resource allocation mechanisms in Grid environments we perform two different experiments. We refer to these experiments as heterogeneity-aware (HA) and heterogeneity-unaware (HU) experiments. In both experiments we consider the problem described in Section 4.1 and use the same scheduling algorithm and data structure; the only difference being that in one experiment we adapt the algorithm and data structure to accommodate heterogeneity.

More specifically we consider the well-known first-fit (FF) scheduling algorithm, and we use a linked-list data structure to store idle periods. In the heterogeneity-unaware (HU) experiment, all idle periods over all servers are stored in a single linked list in ascending order of their starting times. To schedule a new job, the FF algorithm searches the linked list and returns the first feasible idle period for the job; we refer to this algorithm as FF-HU. In the heterogeneity-aware (HA) experiment, the idle periods are stored in H linked lists, where H denotes the number of different rates in the system. Specifically, linked list $h, h = 1, \cdots, H$, stores the idle periods over all servers with rate μ_h in ascending order of their starting times. To schedule a new job, the FF algorithm considers the H lists in some order, and searches the first linked list for a feasible idle period; if no such idle period is found, the algorithm continues to search the next list in the order, and so on. The FF-HA algorithm terminates when the first feasible idle period is found, or when all the lists have been searched unsuccessfully. Clearly, the order in which the FF-HA algorithm considers the H linked lists will have an impact on performance.

We used simulation to compare the performance of the FF-HU and FF-HA algorithms; the details of the simulation setup are described in Section 4.6. Following the model of Section 4.1.1, we consider a (H, n)-heterogeneous Grid with $n = 120$ servers divided into $H = 3$ groups, with the server in each group $h, h = 1, 2, 3$, having the same rate μ_h. We created $L = 4$ (H, n)-heterogeneous systems by selecting the rate μ_h of each server group within each system so that $L = 4$ refers to the most heterogeneous system with respect to expression (4.1) and $L = 1$ to the least heterogeneous one.

Figure 4.2 plots the loss rate and utilization against system load, respectively. Each figure shows two sets of four plots, one set for the FF-HU algorithm and one for FF-HA; in this case, FF-HA considers the H lists of idle periods in increasing value of the rate μ_h of the corresponding servers. Each plot within a set corresponds to one of the $L = 4$ levels of heterogeneity, i.e., one of the (H, n)-heterogeneous systems obtained as we described above. We

have obtained results for other performance measures, e.g., waiting time, but do not include them here as they exhibit similar trends.

As we can see, for a given level of heterogeneity, the heterogeneity-aware algorithm (FF-HA) outperforms the heterogeneity-unaware one (FF-HU) across the spectrum of system loads. We also observe that the performance of each algorithm improves as the system becomes more heterogeneous, despite the fact that the total service rate is the same for all $L = 4$ heterogeneity levels. This phenomenon is due to the effect of statistical multiplexing, and is discussed in more depth in Section 4.6. These results, obtained with a basic scheduling algorithm and data structure, suggest that computational heterogeneity may have a significant impact on both user and system performance metrics and should be taken into account when designing scheduling algorithms. Nonetheless, taking heterogeneity into account comes with a price since it adds complexity to the problem and hence to the algorithms. For instance, although the worst-case running time of the FF-HA and FF-HU algorithms is the same (linear in the number of idle periods), the average running time of FF-HA can be significantly longer than that of FF-HU (since it may have to traverse several lists before it finds a feasible period that might be stored near the head of the single list maintained by FF-HU). The challenge, therefore, is to design scheduling algorithms that are both heterogeneity-aware and efficient; this is the subject of the next two sections.

4.3 A Geometric Model for Advance Reservations

In this section we employ techniques from computational geometry to model the problem we introduced in Section 4.1. We then use this model to develop an algorithm for advance reservation of resources, along with an associated data structure for storing and accessing efficiently the set of idle periods.

Without loss of generality, in the following discussion we make the assumption that the service rate μ_i of each processor i is such that $0 < \mu_i \leq 1$. This assumption allows us to define the size l_j of a job j as the amount of time for this job to complete on a server of rate $\mu = 1$. Clearly, the duration of the job on a server of rate $\mu_i < 1$ is then equal to l_j/μ_i.

4.3.1 Geometric Representation of Idle Periods and Jobs

We represent idle periods and jobs on the first quadrant of a Cartesian coordinate system in which the x axis represents *time* and the y axis represents *nominal capacity*. Figure 4.1(b) illustrates the geometric representation of the idle periods and new job of Figure 1(a). A job j

characterized by the tuple (r_j, l_j, d_j) is represented in this coordinate system as a line segment between two points $P = (r_j, l_j)$ and $P' = (d_j - l_j, l_j)$. Since, in Figure 4.1(a), the new job is defined by the tuple (t_3, l_j, d_j), the two endpoints of the line segment representation of this job in Figure 2(b) are $P = (t_3, l_j)$ and $P' = (t_{10} = d_j - l_j, l_j)$. As defined, point P represents the *earliest* possible starting time and required capacity for this job if it were scheduled on the fastest server, i.e., one with rate $\mu = 1$; similarly, point P' corresponds the *latest* possible starting time and required capacity for this job to be feasibly completed on the fastest server. Note that although we assume that servers may have different capacities, we use *a single* representation for each job j, namely the line segment with respect to the server of rate $\mu = 1$.

An idle period k characterized by the tuple (st_k, et_k, c_k) is also represented in the coordinate system as a line segment between two points, $k_1 = (st_k, c_k)$ and $k_2 = (et_k, 0)$. Recall that c_k denotes the nominal capacity of idle period k. Therefore, point k_1 represents the point in time (i.e., starting time) at which the idle period has the largest nominal capacity, and point k_2 the point in time (i.e., ending time) at which the idle period has reached zero capacity. The slope of the line segment representing idle period k is equal to $-\mu_i$, where μ_i is the rate of the server corresponding to this idle period; this representation clearly shows that the nominal capacity of the idle period decreases at rate μ_i. Consider, for example, idle period x in Figure 4.1(a) with starting time $st_x = t_1$, ending time $et_x = t_4$, and nominal capacity c_x. This idle period is represented in the plane by the line segment between the two points $x_1 = (st_x, c_x)$ and $x_2 = (et_x, 0)$. The slope of the line segment is -1, since the rate of server 1 is $\mu_1 = 1$. Idle periods y, z, and w are similarly represented by the line segments shown in Figure 4.1(b). Note also that the slope of the line segment corresponding to idle periods y and z is -1, while the one corresponding to w is -0.5 since the latter is on server 2 of rate $\mu_2 = 0.5$.

Feasibility Criteria. We may now use the above geometric representation to determine whether an idle period is feasible for a new job. Consider an idle period k with tuple (st_k, et_k, c_k) represented by the line segment defined by points k_1 and k_2, as explained earlier, and a new job j with tuple (r_j, l_j, d_j) that is represented by a line segment between points P and P'. Idle period k is feasible for job j if and only if both of the following conditions are satisfied.

1. *Starting time feasibility.* Let i be the server corresponding to idle period k, and μ_i be its service rate. For the idle period k to be feasible for the new job j, its starting time st_k has to be sufficiently early for the server to be able to complete the job before its deadline, i.e.:

$$st_k \quad \leq \quad d_j \; - \; \frac{l_j}{\mu_i} \tag{4.3}$$

Expression (4.3) is necessary but not sufficient for feasibility, since the idle period k may end early, before job j can complete on server i. Returning to Figure 4.1, we observe that idle period x satisfies the above condition with respect to the new job. However, the residual capacity of this idle period at the time the new job arrives is not sufficient to accommodate it.

2. *Capacity feasibility.* Assuming that the starting time feasibility is satisfied, an idle period k is feasible for a new job j if the line segment representing k lies above or intersects with, the line segment representing j. Equivalently, this condition is satisfied if the leftmost endpoint of the line segment representing the new job lies below the line segment representing the idle period. In Figure 4.1(b) we see that idle period y does not satisfy this condition as its line segment lies below the line segment representing the new job; hence, y is not feasible for the new job.

In Figure 4.1(b), the two conditions are satisfied for both idle periods w and z with respect to the new job represented by the line segment between points P and P'. Consequently, idle period w has enough capacity to accommodate the new job, as long as the latter starts before the time instant at which the corresponding lines intersect; similarly for idle period z.

Our objective is to develop techniques to identify efficiently feasible idle periods for each arriving job request, *without* having to examine all idle periods. As we have shown in [76], we can efficiently find idle periods that meet the starting time feasibility criterion by organizing the idle periods in an appropriate balanced tree structure that can be searched in logarithmic time. However, identifying idle periods that meet the capacity requirement, e.g., determining line segments lying above point P in Figure 4.1(b), requires that each idle period be examined separately. This is due to the fact that to perform this test the equation representing each line segment needs to be evaluated for the coordinates of the given point.

Next, we employ techniques from computational geometry to obtain an equivalent representation of idle periods and new jobs that allows us to develop an elegant solution to the problem of testing for the capacity feasibility criterion.

4.3.2 Duality Transform and Duality Plane.

Geometric duality [71] refers to the direct mapping between a point p (respectively, line l) and a line $p^\star$ (respectively, point $l^\star$). The duality transform maps objects from the *primal* plane to the *dual* plane. We now describe a simple duality transform we use in the remaining of this paper. Let $p := (p_x, p_y)$ be a point in the plane. The dual of p, denoted $p^\star$, is the line

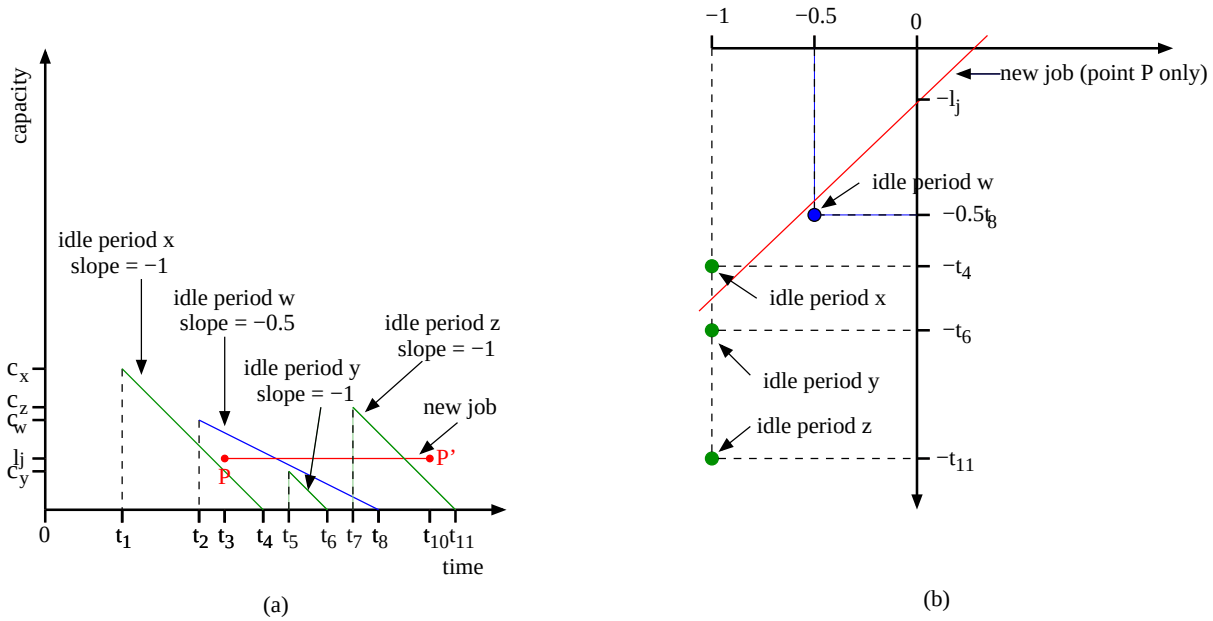


Figure 4.3: (a) Primal plane and (b) dual plane representations of the idle periods and new job of Figure 4.1(a)

defined as

$$p^\star := (y = p_x x - p_y) \tag{4.4}$$

where p_x and p_y are p's x and y coordinates, respectively. The dual $l^\star$ of a line $l := (y = mx + b)$ is the point p such that $p^\star = l$, that is,

$$l^\star := (m, -b) \tag{4.5}$$

where m and b are the slope and y-intercept of line l, respectively. One major advantage of this particular duality transform is that it is *order preserving*, that is, point p lies above line l if and only if point $l^\star$ lies above line $p^\star$ [71].

Let us now return to our original problem and the geometric representation of idle periods and jobs shown in Figure 4.1(b). We transform this primal plane to the dual plane by mapping the line l_k corresponding to an idle period k to a point $l_k^\star$, and the point P corresponding to the earliest time new job j can start execution, to a line $P^\star$. Using basic geometry principles we find that for any idle period k, the value of b in expression (4.5) is $\mu_i et_k$. Since the slope m of idle period k is $-\mu_i$, where μ_i is the rate of the corresponding server, expression (4.5) can be written as:

$$l_k^\star := (-\mu_i, -\mu_i et_k). \tag{4.6}$$

To find $P^\star$, we substitute p_x and p_y in expression (4.4) with r_j and l_j, respectively:

$$P^\star := (y = r_j x - l_j). \tag{4.7}$$

Figure 4.3(b) shows the dual plane corresponding to the primal plane in Figure 4.3(a); the latter figure is identical to Figure 4.1(b), and is repeated here for convenience. As we can see, the idle periods are now mapped to points in the dual plane. Specifically, all idle periods on server 1 of rate $\mu_1 = 1$ are now points with y coordinates equal to $-\mu_1 = -1$; similarly, the idle period on server 2 of rate $\mu_2 = 0.5$ has y coordinate equal to $-\mu_2 = -0.5$. Point P, on the other hand, which represents the earliest time the new job can start execution is represented on the dual plane as a line.

Consider now the capacity feasibility criterion we defined above. In the primal plane of Figure 4.3(a), it is clear that the idle period x is not feasible for the new job, as point P lies above the line segment representing x. Due to the order preservation of the duality transform, in Figure 4.3(b) we see that the point corresponding to idle period x also lies above the line representing point P. Similarly, idle periods y and w are feasible for the new job, and their corresponding points in the dual plane lie below the line representing point P. Therefore, checking for capacity feasibility in the dual plane requires checking whether the points representing idle periods lie below the line representing the new job. This test can be performed efficiently by organizing the idle periods (points) lying on the vertical line $x = -\mu$ (i.e., those corresponding to servers with rate μ) in a search tree structure, and searching for those with a y-coordinate less than that of the point at which the line representing the new job intersects the line $x = -\mu$; this search structure is described in the next section.

The observant reader will have noticed that, in the dual plane of Figure 4.3(b), the point representing idle period y lies below the line representing point P; however, a look at the primal plane of Figure 4.3(a) indicates that idle period y is *not* feasible. Note that extending the line segment representing idle period y in the primal plane would result in a line lying above point P, hence the dual plane representation is consistent in this regard. The issue here is that idle period y starts too late to be feasible, therefore it will not pass the starting time feasibility criterion above. Consequently, both the starting time and capacity feasibility criteria must be checked to ensure that an idle period is feasible.

4.4 Algorithm and Data Structure Description

We now introduce an efficient algorithm for finding a feasible idle period for a new job in a (H, n)-heterogeneous system with advance reservations. The algorithm is derived from the heterogeneity-aware FF-HA algorithm we described in Section 4.2, and will refer to it as FF-HA+. The FF-HA+ algorithm differs from FF-HA in that it maintains H balanced trees, rather than H linked lists, such that balanced tree $T_h, h = 1, \cdots, H$, stores information about the idle periods over all servers with rate μ_h. Similar to FF-HA, when a new job arrives, FF-HA+ searches the balanced tree structures in ascending order of server rate, and returns as soon as it finds a feasible idle period.

4.4.1 Balanced Tree Structure

The FF-HA+ algorithm maintains H 2-dimensional binary search trees to organize the idle periods in a (H, n)-heterogeneous system, one such tree $T_h, h = 1, \cdots, H$, for each distinct server rate value μ_h. Whenever the algorithm needs to search the idle periods available in servers associated with rate μ_h, the associated tree T_h is searched.

We will refer to the first and second dimension trees of T_h as T_h^{primal} and T_h^{dual}. As their name indicates, they organize the idle periods according to their parameterizations on the primal and dual planes, respectively. More specifically, tree T_h^{primal} is used to select idle periods that meet the starting time feasibility criterion, and tree T_h^{dual} is used to select among these idle periods the ones that meet the capacity feasibility criterion.

Let us now describe the 2-dimensional tree T_h more in detail. In tree T_h^{primal}, the actual idle periods are in the leaf nodes, arranged in ascending order of their starting time. A leaf node corresponding to idle period k stores the following information:

- the starting time of k;
- the ending time of k; and
- auxiliary data, such as the identity of the corresponding server.

Internal tree nodes store information regarding the idle periods in their subtree. This information is used to navigate the tree and locate idle periods appropriate for the new job. The information at an internal node v consists of:

- the median starting time of the idle periods stored in the subtree of T_h^{primal} rooted at v; and

- a pointer to the secondary priority search tree T_h^{dual} containing idle periods.

Tree T_h^{dual} stores the idle periods sorted in descending order of the y-coordinate of their dual representation, that is, of the corresponding point in the dual plane. Each intermediate node v in T_h^{dual} stores the following information:

- the median y-coordinate of the dual representation of the idle periods stored in the subtree rooted at v; and
- a pointer to the idle period in v's subtree with the maximum nominal capacity.

4.4.2 Searching the Balanced Tree Structure

Consider a request to schedule a new job j with parameters (r_j, l_j, d_j). The FF-HA+ algorithm searches the H balanced trees as we explained earlier, and returns the first feasible idle period found. We now describe how the search of balanced tree T_h is performed; this process is identical for all trees $T_h, h = 1, \cdots, H$. Specifically, the search proceeds in two steps:

1. In the first step, the algorithm traverses the tree T_h^{primal} and marks the intermediate nodes v whose subtrees contain idle periods that meet the starting time feasibility criterion.
2. In the second step, the algorithm searches the secondary trees T_v^{dual} at each intermediate node v marked during the first step, to locate the subset of idle periods that meet the capacity feasibility criterion.

Step 1: Search in T_h^{primal}. In this step, the algorithm identifies idle periods that meet the starting time feasibility criterion expressed in (4.3). To this end, we employ a standard search algorithm which starts at the root node and compares the quantity in the right-hand side of (4.3) to the median starting time stored at each internal node v. If the median starting time is smaller, then all the idle periods stored in v's left subtree meet the first feasibility criterion; the algorithm marks the left subtree and proceeds to search the right subtree. If the median starting time of the tree rooted at v is larger, then we can safely conclude that all the idle periods in the right subtree are infeasible and proceed recursively to search the left subtree of v. The algorithm returns the set of marked intermediate nodes as soon as it reaches a leaf, and proceeds to Step 2 described below. If no intermediate node is marked, the FF-HA+ strategy continues to search in the 2-dimensional tree T_{h+1} corresponding to the next larger value of server rate.

Step 2: Search in T_v^{dual}. In this step, the algorithm searches the idle periods meeting the starting time feasibility criterion, to identify the ones that also satisfy the capacity feasibility

criterion. To this end, the algorithm searches each of the subtrees rooted at the intermediate nodes marked in Step 1 and returns as soon as it finds one feasible idle period (if one exists). We will refer to T_v^{dual} as the secondary tree, i.e., the dual tree, associated with marked node v. The algorithm starts at the root of T_v^{dual} and compares the median y-coordinate stored at each internal node u to the y-coordinate of the point in the dual plane at which the line corresponding to the new job intersects the vertical line $x = -\mu_h$ (refer also to Figure 4.3(b)). If the latter value is smaller then it can be concluded that all the idle periods in the left subtree are above the line, and hence are infeasible; the algorithm then recursively searches u's right subtree. If the former value is smaller, then all the idle periods in the right subtree of u are feasible, and there may also exist feasible idle periods in its left subtree. In this case, the algorithm accesses the idle period with the *maximum capacity* in the right subtree by following the pointer stored at node u. If this idle period is feasible, the algorithm returns it and assigns it to the new job. Otherwise, the search continues recursively with the left subtree of u. If the algorithm reaches a leaf, then no feasible idle period exists in the given subtree and the algorithm continues searching the next tree marked in Step 1.

Running time complexity. In the worst case, the search algorithm marks an intermediate node at each level of the tree T_h^{primal} in Step 1. Given that it has to perform a standard search for each of these trees, the overall complexity is $O(\log^2 V_h)$ for 2-dimensional tree T_h, where V_h is the number of idle periods in the tree. Since the algorithm may have to search all H trees, the worst case complexity for FF-HA+ is $O(H \log^2 V)$, where $V = \max\{V_h\}$. As a comparison, the running time of FF-HA is $O(HV)$, i.e., linear in the number of idle periods, since it has to traverse H linked-list structures. Since H is typically a small constant, whereas the number V of idle periods can be quite large (especially for large systems with thousands of servers and for long time horizons for advance reservations), FF-HA+ is significantly more scalable than FF-HA.

4.5 Adaptability: Re-planning Capacity and Maximizing Utilization

In this section we describe two mechanisms that make it possible to exploit the efficiency of FF-HA+ in order to relax the hard deadline assumption and accommodate changes in resource availability; the implementation of these mechanisms is the subject of ongoing work within our group.

Replanning Capacity. In our work so far we have assumed that deadlines are hard, i.e.,

jobs are dropped if they can not be allocated within their deadline. It is possible to make the algorithm more flexible and increase the overall ability of the system to meet application QoS requirements by introducing a negotiation process. This process is invoked whenever the scheduler fails to allocate a job and attempts to reschedule existing reservations in order to allocate new incoming jobs whenever possible without affecting the QoS of previously scheduled jobs. This negotiation process may utilize a set of data structures and algorithms similar to the one we described in the previous section to organize, search, and modify existing reservations.

Our algorithm can also be adapted to handle efficiently changes in job demands. Consider, for instance, a job currently running on a server, and assume that it needs to execute for a longer period of time than the one it originally reserved (i.e., the original estimate of its running time was incorrect). In current systems, such jobs are either terminated or preempted and given low priority for scheduling. Given the low running time complexity of our search algorithm, there are several options to handling such situations: one can either invoke the negotiation process to reschedule the job that has reserved the server following the current job, or one can checkpoint the job, invoke the scheduling algorithm to find the next available feasible idle period for it, and then migrate the job to complete execution in another server.

Opportunistic Scheduling. To enable users and Grid administrators to exploit the variations of resource conditions to improve both application and system performance, the FF-HA+ algorithm may be extended to implement opportunistic scheduling. More specifically, new jobs that have no deadline requirements may use resources as they become available, and they may be preempted to accommodate new jobs with deadlines. Such an approach will increase utilization by filling idle periods that might not be used otherwise, and increases the flexibility of the system.

4.6 Performance Evaluation

In this section we present simulation results to demonstrate the performance of the FF-HA+ scheduling algorithm. We used the method of batch means to estimate the performance parameters we consider (and which we discuss shortly), with each batch consisting of thirty simulation runs and each run lasting until 10^6 jobs have been submitted to the Grid scheduler. We have also obtained 95% confidence intervals for all the results, which are shown in the figures.

In our simulation, we assume that job requests arrive following a uniform distribution in the range from one minute to 14 days [35]. The duration of each reservation request is randomly

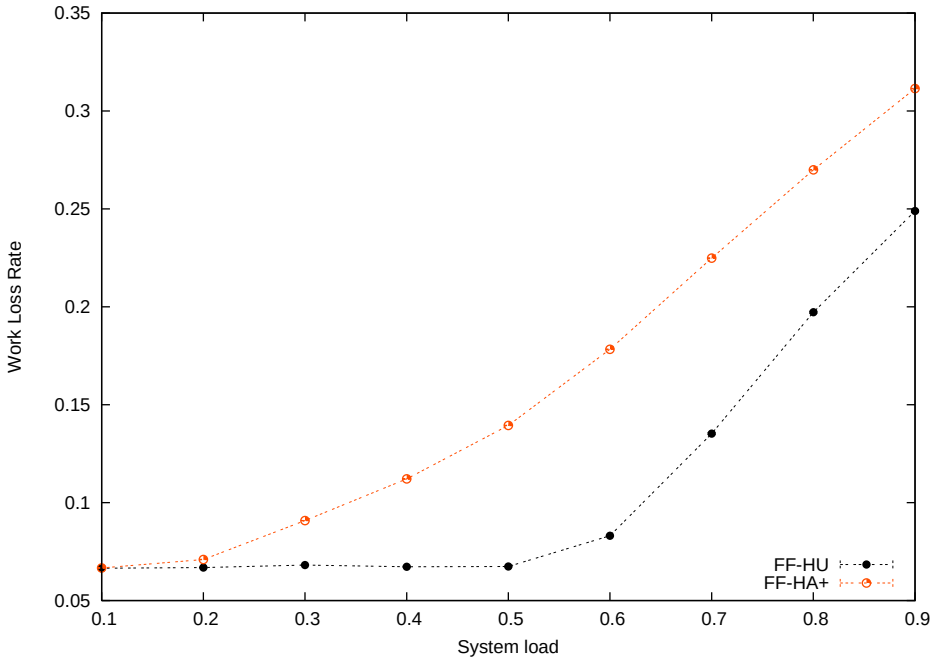


Figure 4.4: Comparison of FF-HA+ and FF-HU: (a) Work loss rate against system load

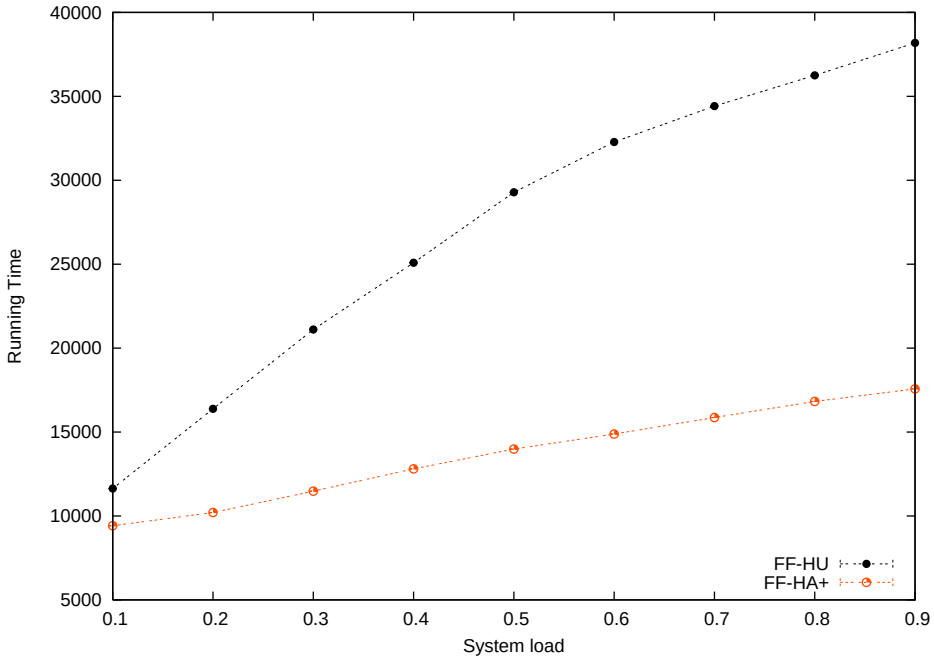


Figure 4.5: Comparison of FF-HA+ and FF-HU: Running time (milliseconds) against system load

selected so that 80% of the incoming jobs are smaller than 4 hours, and 20% are between 4 and 36 hours; the mean job size is 5.6 hours. These values were chosen based on the experience with

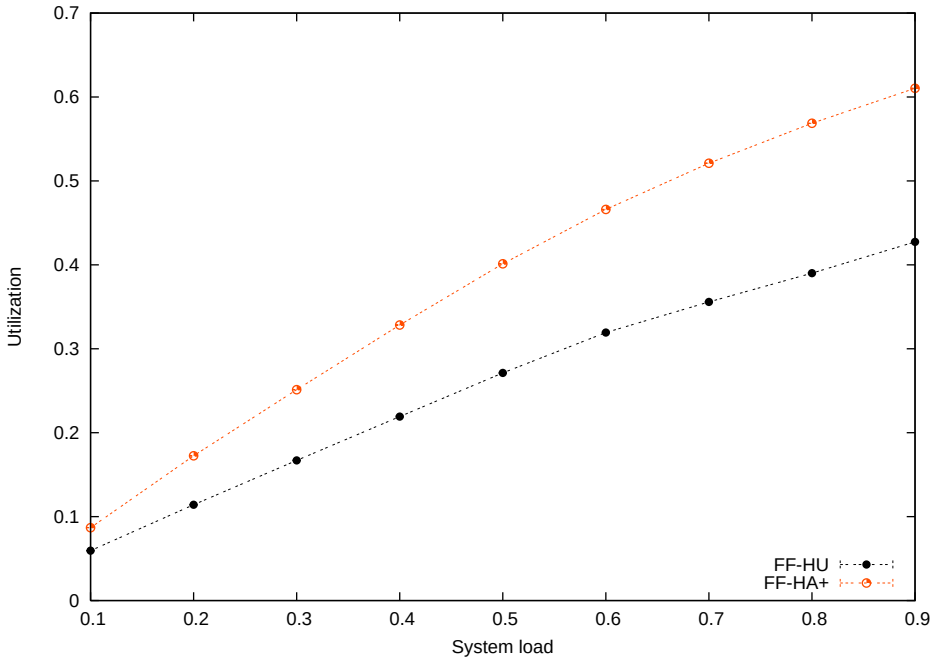


Figure 4.6: Comparison of FF-HA+ and FF-HU: Utilization against system load.

running real Grid workfolk applications as described in [**?**, 35]. We let the deadline d_j of job j be uniformly distributed in the interval $(r_j, r_j + q)$, where q corresponds to the "tightness" of the deadline; for most of our experiments we assume $q = 20$ hours unless stated otherwise.

We consider a (H, n)-heterogeneous system with $n = 120$ servers and $H = 3$ distinct service rates. We generated and studied $L = 4$ computational rate distributions such that $L = 1$ refers to the least heterogeneous system and $L = 4$ refers to the most heterogeneous one.

We use four performance metrics in our study. The *work loss rate* is the fraction of work that is dropped due to the fact that the deadline of the corresponding jobs cannot be met. The *system utilization* is the fraction of time the n servers are busy serving jobs. The *waiting time* is the mean amount of time that a job has to wait beyond its ready time until it starts execution; note that dropped jobs do not contribute to the average waiting time. Finally, the *algorithm running time* captures the efficiency of the search algorithm to schedule incoming jobs. To compute the running time we record the CPU time for each simulation corresponding to 10^6 jobs. Work loss rate and waiting time are measures of the QoS perceived by the user, system utilization is a measure of system performance, and running time determines the scalability of the system.

In our first experiment, we compare the FF-HA+ algorithm to the baseline algorithm FF-HU we described in Section 4.2. Recall that FF-HU strategy organizes idle periods in a

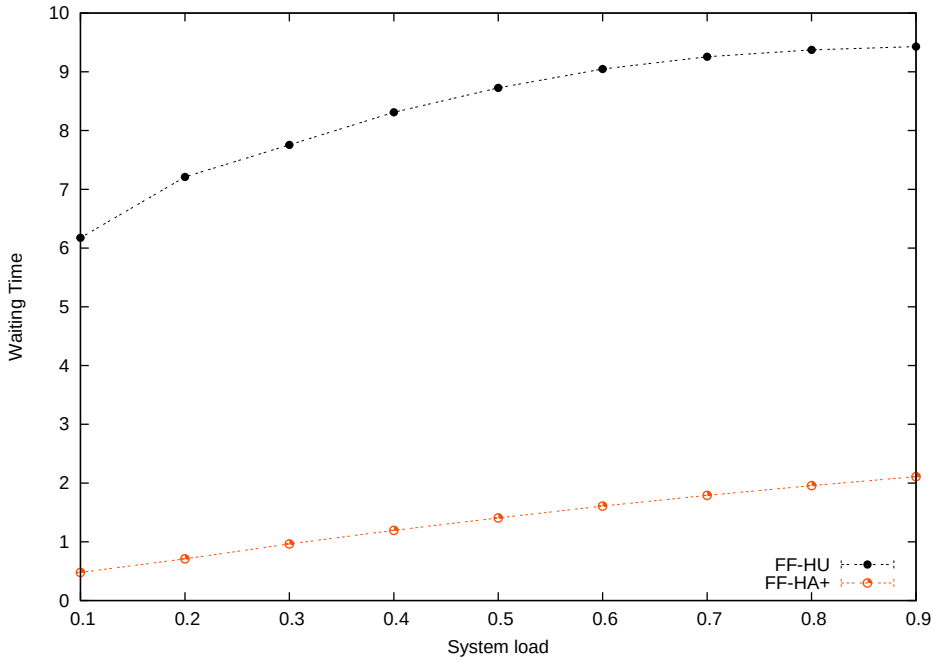
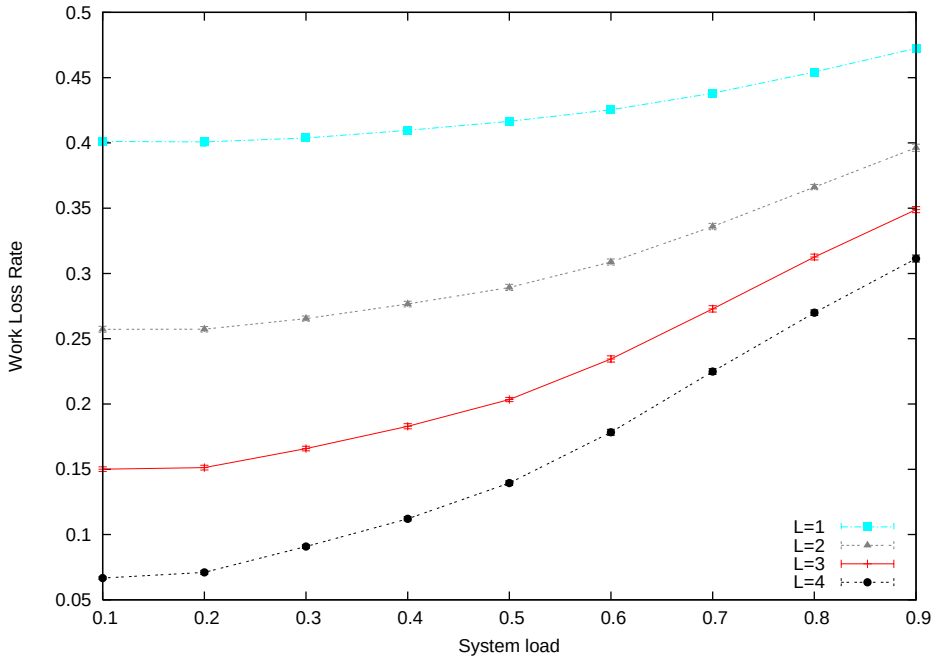


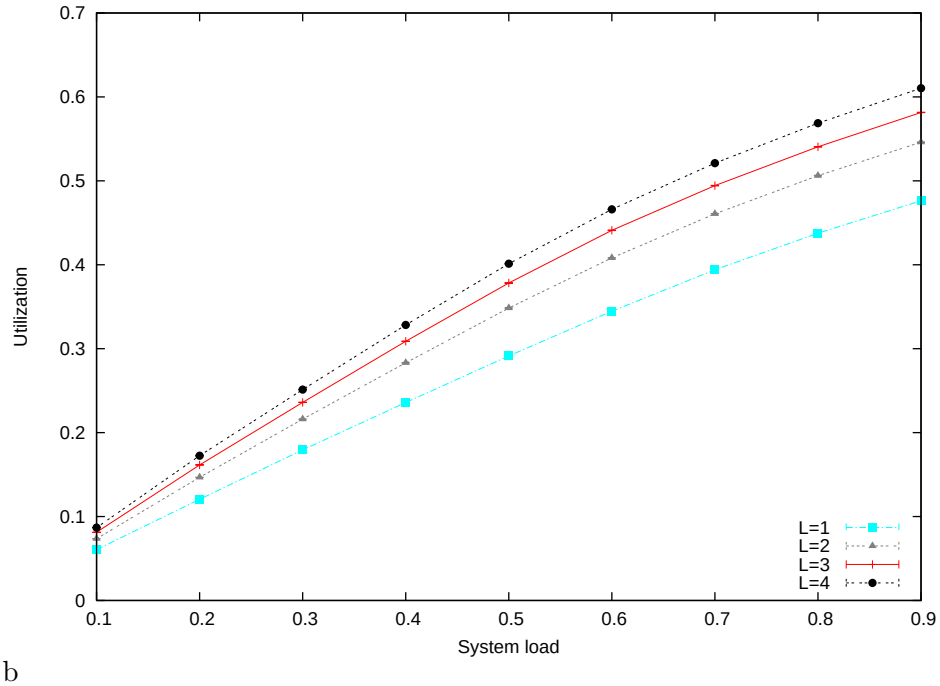
Figure 4.7: Comparison of FF-HA+ and FF-HU: Waiting time against system load.

single linked list ordered in ascending order of their starting time; the algorithm traverses the list and returns the first feasible idle period for a new job, i.e., the one with the earliest starting time. Note that idle periods with early starting times are at risk of expire unused if new jobs are not assigned to them. Therefore, this choice of a feasible idle period is expected to lead to low loss, since assigning a new job to the earliest possible feasible period allows idle periods starting later to be used for future job requests. On the other hand, the running time of the algorithm increases quickly with the size of the Grid system and the time horizon for making reservations. The FF-HA+ algorithm organizes the idle periods in balanced tree structures, hence it scales well to large Grid systems. However, it does not necessarily return the feasible idle period with the earliest starting time, hence we expect that its work loss rate will be higher than FF-HU. But we emphasize that FF-HA+ will always find a feasible idle period for a new job if one exists.

Figure 4.5 confirms the above observations. The figure plots the work loss rate and running time of the FF-HU and FF-HA+ algorithms against the system load. As we can see in Figure 4.5(a), the loss rate increases with the system load for both algorithms. The two strategies exhibit similar loss rates at low loads (when there are sufficient resources to schedule almost all jobs) and high loads (when the issue is the lack of resources, not the particular strategy used). However, the FF-HA+ strategy exhibits a higher loss rate at medium loads, as we expected. A careful examination of our experiments shows that FF-HU incurs less resource



a



b

Figure 4.8: The impact of heterogeneity: (a) work loss rate against load, (b) utilization against load

fragmentation that FF-HA+. This result is due to the fact that FF-HA+ returns the feasible idle period of maximum capacity among those in its subtree; while this choice was made to speed up the operation of the algorithm, the side effect is higher fragmentation. On the other hand, the running time of FF-HU is significantly higher than that of FF-HA+, especially at medium to high loads; again, this result is consistent with our discussion above.

The system utilization curves in Figure 4.7(a) suggest that FF-HA+ utilizes better the resources available in the system, i.e., the servers are busy performing work for a longer fraction of time than under FF-HU. However, since the loss rate for FF-HA+ is slightly higher, this results implies that FF-HA+ allocates more jobs to slow processors than FF-HU. A more careful examination of our results reveals that, under FF-HA+, processors with high service rate exhibit a higher fragmentation; since the capacity of processors with high service rate expires faster as time progresses, fragmentation of capacity on high-rate servers has a more detrimental effect on system performance, as exhibited by the higher loss rate of HH-FA+. Figure 4.7(b) plots the average waiting time that jobs have to wait beyond their ready time. We observe that jobs have to wait significantly longer under FF-HU compared to FF-HA+. In other words, although FF-HU schedules a larger fraction of jobs than FF-HA+, the start time of these jobs is pushed back resulting in longer waiting times.

Finally, Figure 4.8 investigates the impact of different levels of heterogeneity on performance. Figure 4.8 (a) plots the work loss rate against the load for $L = 4$ different levels of heterogeneity, where larger values of L imply higher heterogeneity; Figure 4.8 (b) is similar but plots system utilization against load. We can see that as resources become more heterogeneous, the loss rate and system utilization both improve, in many cases significantly so. This behavior follows from the fact that to increase resource heterogeneity in a given system while keeping the total service rate constant, as required by expression (4.2), the service rate of a few fast processors must increase further. In other words, a larger fraction of the total service rate is concentrated on fewer resources. Consequently, making the system more heterogeneous introduces a higher degree of statistical multiplexing, whereby fewer high capacity servers are responsible for serving larger number of customers. The results in Figure 4.8 then are consistent with the well-known fact from queueing theory that statistical multiplexing improves system performance.

4.7 Concluding Remarks

We have considered the problem of advance reservations for jobs with deadlines in a Grid system with heterogeneous resources. We have developed a geometric representation of idle periods and jobs that provides new insight and allows for efficient organization of the reservations. We have developed a scheduling algorithm with good performance that can scale to large Grid systems and long time horizons. We have also shown that resource heterogeneity may have a positive impact on performance if taken into account in the design of scheduling algorithms.

Chapter 5

Efficient Coallocation Scheduling Algorithm

One of the major benefits resulting from supporting advance reservations is the provision of resource co-allocation support to resource managers. In principle such simultaneous allocation can be achieved by leveraging algorithms as the ones presented in Chapters 3 and 4. However, the growing trend towards more virtualized environments has emphasized the need to provide automated solutions to support the management and coordination of multiple resources that are efficient, effective and scalable.

In light of these observations, this chapter presents an online co-allocation algorithm that is *efficient* in co-allocating resources while providing support for advance reservations and temporal range search, i.e., users can request a list of all resources available within a specific time window. To achieve this, we partition the temporal space into a set of *quanta* and use efficient 2-dimensional data structures to organize the co-allocations. Moreover, we perform an in-depth comparative analysis of our algorithm against the batch scheduling algorithm under real workloads. Our results indicates that online scheduling algorithms may achieve–under most conditions–higher utilization while providing smaller delays and better QoS guarantees; and all this can be achieved without adding much complexity. Furthermore, we show that our co-allocation algorithm scales to systems with large number of resources and heavy workloads.

The organization of this chapter is as follows. In Section 5.1 we give a formal description of the problem being considered. Two major applications of the co-allocation scheduling problem are described in Section 5.2. In Sections 5.3 and 5.4 we describe in detail the data structure and algorithm; and in Section 5.5 we analyze their performance. Finally we conclude this piece of work in Section 5.6.

5.1 Problem Description

Consider a scheduler $\mathcal{S}$ for a large distributed computing system with N servers, which may be geographically distributed in a network. A user with a job requiring service submits a request r to $\mathcal{S}$. The request is characterized by a four-parameter tuple (q_r, s_r, l_r, n_r), where:

1. q_r is the request time, i.e., the time the request is submitted by the user;
2. s_r is the earliest time the job can start execution. Note that when advance reservations are supported $s_r \geq q_r$. Otherwise, $s_r = q_r$;
3. l_r is the temporal size of the reservation, i.e., estimated duration of the job;
4. n_r is the spatial size of the reservation, i.e., the number of servers required for the given job.

We assume that $\mathcal{S}$ maintains a schedule which records, for each server, the time periods in the future during which the server is reserved for requests that have already been accepted by the system. In essence, this schedule represents the set of reservations that have been made, and it guarantees that server resources will be available to the accepted jobs at specific future times. Figure 5.1 shows an example schedule for a 4-server system. The schedule shows that at the current time (i.e., time t = 0 in the figure), there are two jobs scheduled for server 1: job A which is currently in service and will end at time t_4 and job B which has reserved the server from time t_{25} to t_{37}; similarly, two jobs have been scheduled for server 2, server 3 and server 4, respectively. Figure 5.1 also shows a service request R for scheduling a new job with $q_r = s_r = 17$, temporal size $l_r = 12$ and spatial size $n_r = 2$.

When a service request $r = (q_r, s_r, l_r, n_r)$ for a new job arrives, $\mathcal{S}$ immediately runs an algorithm to determine whether it is feasible to schedule the job. If feasible, $\mathcal{S}$ selects multiple servers that can handle this job, updates its schedule, and returns a reference to the n_r servers to the user; otherwise, the scheduler increases s_r by Δ_t units of time and retries scheduling the modified request. If after increasing s_r $maxRet$ times, the request r has not been met, the reservation request is put on hold and submitted to the scheduler for reconsideration at time $t = q_r + H$, where H represents the *time horizon* of the system, i.e., the last possible time at which the system can schedule requests.

The scheduling decision influences the performance perceived by users as reflected by the fraction of jobs scheduled at s_r (or later) and the turnaround times of the jobs. It also impacts the overall system performance as reflected by system utilization, which is a measure of

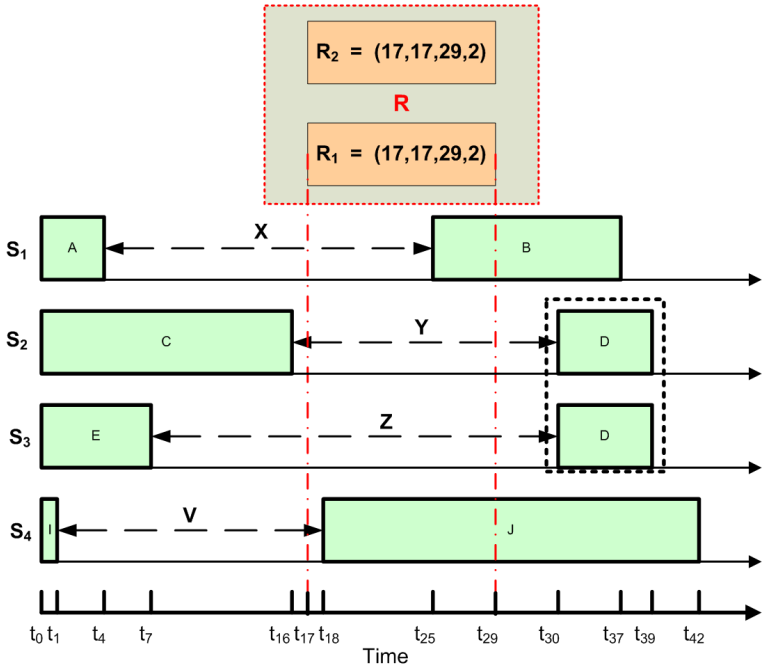


Figure 5.1: 4-server system with co-allocations and advance reservations

how well the overall service capacity of the system is being used. The challenge, therefore, is to develop efficient online co-allocation algorithms that minimize the fraction of delayed jobs and maximize the fairness experienced by users while maximizing utilization.

Note that the model just described can be easily extended to support advance reservations by defining $s_r > q_r$ as described in Section 5.5.

5.2 Applications

Co-allocation scheduling algorithms are of interest to many applications. In this section, we describe two applications of interest within the context of large scale distributed systems. These applications are being used extensively in real working environments and emphasize the importance of developing efficient co-allocation mechanisms.

5.2.1 Computing Resources in Virtual Computing Laboratory (VCL)

Computing over the Internet is becoming increasingly popular. Thanks to emerging infra-structures such as computational Grids and web services, it is possible to develop applications that support various Internet-wide-collaborations by seamlessly harnessing appropriate resources. The Virtual Computing Laboratory (VCL) [8] is one initiative established at North

Carolina State University, which is projected to utilize network and computing resources to connect all K-20 institutions across the state of North Carolina. One of the main challenges faced in this project is scalability in terms of resource provisioning and management given the large number of resources and users. Furthermore, in order to fulfill such a wide user population the VCL initiative needs to accommodate for multiple applications. Here we refer to two such applications more relevant to our work: desktop virtualization and high performance computing (HPC) provisioning.

Desktop virtualization provisioning consists of dispatching virtual desktops customized to a set of specific requirements for in-class and laboratory experiences. This initiative has the potential of saving money to schools from hardware and software ownership to IT human resources. Furthermore, it provides schools with access to cutting edge technology that can be leveraged to support advance educational techniques (e.g., learn by playing, virtual environments). A second major application of VCL is the provisioning of HPC resources for large scale jobs. This application targets more sophisticated and demanding users such as graduate students, scientists and faculty who rely on computing intensive experiments to carry out their research.

In both applications users request the resource manager a number of resources needed for a specific time window based on class schedules and/or deadlines. The resource manager then runs an algorithm to determine the availability of the resources and informs the user. If the request is granted, the manager sends the authentication information required for the user to gain access to the resources. Otherwise, it suggests alternative times at which the resources are available. Note that the algorithm proposed in this work supports both, on-demand (appropriate for batch schedulers and workloads with best-effort requirements) and advance reservations requests. Therefore, it is suitable for the VCL workload which is characterized by having mixed workloads with the majority being on-demand.

5.2.2 Grid Lambda Scheduling

Current research practices tend towards collaboration among institutions across countries that require high bandwidth connections to use multiple network-connected resources. This need has been recognized by the government and several initiatives have been funded and established to promote such collaboration (e.g., [40]). The realization of this vision depends on the development of scheduling capabilities that allow researchers to request and manage network resources on demand; and, the deployment of infrastructure that provides automated, scheduled

and rapid establishment of lambdas across administrative domains.

Critical to this vision is the scheduling of link wavelengths within each administrative domain along an end-to-end network path. One proposal [77] that has gained popularity is the use of PCE (Path Computation Element) [78] to provide with link wavelength scheduling. In this context, the link wavelength scheduling problem can be formulated as follows: Given a request–including a source and destination pair, range of wavelengths, a time window and the estimated length of the connection–the PCE entity finds link wavelength resources along a path from the source to the destination nodes within the administrative domain to satisfy the request. In essence this problem is equivalent to the co-allocation problem since the wavelengths on all links of the path must be allocated and de-allocated *simultaneously*.

The benefits of implementing the co-allocation scheduling algorithm proposed in this paper in the context of the aforementioned applications is two-fold. First, given the low complexity of the proposed algorithm, the resource manager can guarantee short response times, which in turn, improves the overall *efficiency* as perceived by the user and the system. Second, two important features of the co-allocation algorithm proposed are that it supports advance reservation and range search for resources. Range searching means that a request with a given time window, the algorithm returns all the resources available within that window specified. Thus, the algorithm enables users to refine their request and optimize for their requirements through the composition and execution of more complex requests.

5.3 Data Structure

An idle period is characterized by its starting time, ending time and server identification. In order to achieve efficiency of the algorithm we partition the temporal space into a set of quanta $q = 1m \cdots, Q$ where $Q = \lceil \frac{H}{l_q} \rceil$of size l_q (time units) each. In this scheme the leftmost and rightmost quantum contains the earliest and latest available idle periods, respectively. The rightmost quantum also represents the horizon of the system H. Each idle period is stored in each quantum it spans over. Figure 5.2(a), illustrates the concept: idle period x is stored in quanta t_0, t_{10} and t_{20}. One advantageous characteristic resulting from this partioning is that by tuning l_q, the number of idle periods stored in each quantum can be bounded to the number of servers in the system N, i.e., at most one idle period per server per quantum. For example, without losing generality, we could assume that there is a minimum reservation request size and it is equal to l_q. Note that small jobs could easily be packed together in order to satisfy this requirement. This in turn, has a beneficial impact on the running time of the co-allocation

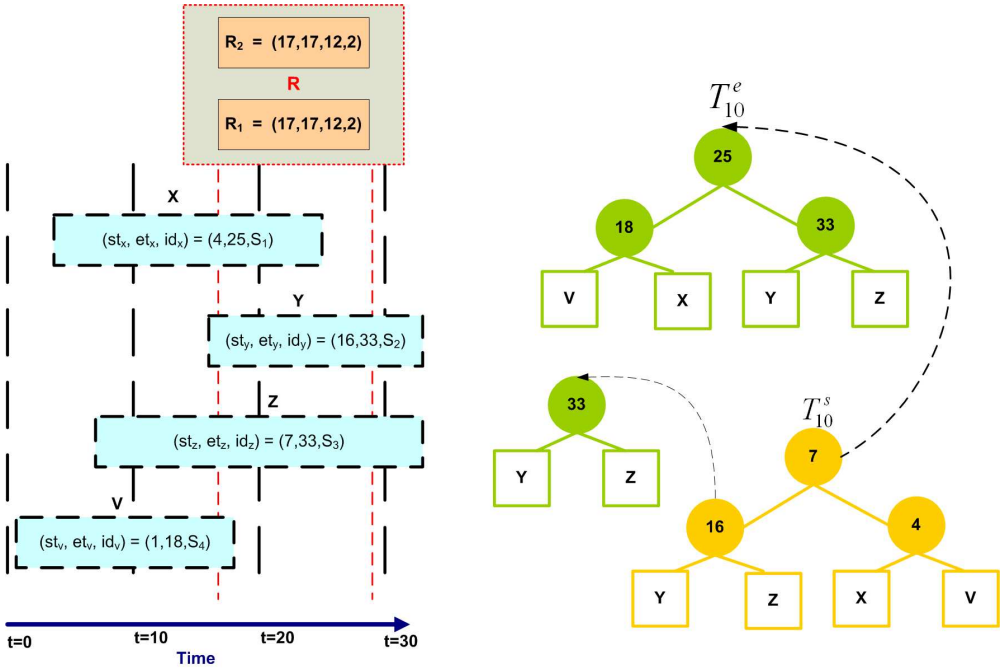


Figure 5.2: (a) Reservation request for 2 resources. Note that the remaining time of idle period u, $r_u$is expressed in terms of t_0; idle periods x, y and z are expressed in terms of quantum $_{10}$, and (b) 2-dimensional tree T_{10}^r containing the idle periods in quantum t_10. T_{10}^S stores the idle periods in descending order of their starting time.

algorithm as explained later in this section.

For each idle period i stored in a quantum q we keep the following information:

- the starting time st_i;
- the ending time et_i; and
- the server id id_i referring to an identification number for the server offering the idle period i.

For the co-allocation algorithm we use a 2-dimensional tree T_q to store the idle periods within each quantum $q, q = 1, \cdots, Q$. This data structure is depicted in Figure 5.2(b). In the tree corresponding to T_q's first dimension, ${t_q}^s$, idle periods are stored in the leaf nodes and arranged in descending order of their starting time. The information stored at each of the internal nodes u of ${t_q}^s$ consists of:

- the median starting time of the idle periods stored in the subtree ${t_q}^s$ rooted at u;
- the size of the subtree rooted at u; and

- a pointer to a secondary binary search tree ${t_q}^e$.

Tree ${t_q}^e$ corresponds to T_q's second dimension and stores the idle periods in u's subtree in ascending order of their ending time. The information stored at each node v of tree ${t_q}^e$ consists of:

- the median ending time of the idle periods stored in the subtree ${t_q}^e$; and
- the size of the subtree rooted at v.

Note that as the time advances the quanta containing idle periods with ending times smaller than the current time expire. Our approach of partitioning the temporal plane into quanta and maintaining a separate tree structure for the idle periods within each quantum makes it easier to handle expired idle periods. Let us assume that the system starts operation at time $t = 0$, and that we maintain Q *quanta*, each of width l_q. Since the scheduling horizon (i.e., the time in the future during which a job can be scheduled) is $Q \times l_q = H$, no idle period can end at time $t' < t + Q \times l_q$ where t is the current time.

Consider the leftmost quantum with index $q = 1$. Initially, the latest time at which an idle period in this quantum may end(expire) is at time $t' = (Q+1) \times l_q - \epsilon$, and it corresponds to the scheduling at time $t = l_q - \epsilon$ of a request with $s_r = Q \times l_q$ time units in the future. Therefore, at time $t = (Q+1) \times l_q$ the tree corresponding to quantum with index $q = 1$ is discarded sine all idle periods recorded in the tree have expired. At the same time, a new empty tree is created to record idle periods falling in the new quantum with index $q' = H$. This discard operation is repeated every l_q time units thereafter. All the operations involved in discarding a tree can be performed in $O(1)$ time with no extra memory cost.

5.4 Online Co-Allocation Scheduling Algorithm

Consider a reservation request r with requesting time, starting time, ending time and number of resources of q_r, s_r,l_r and n_r, respectively.

Given this request, the co-allocation algorithm needs to find n_r feasible idle periods in order to satisfy the request r. An idle period is feasible for a given reservation request if its starting time (ending time, respectively) is smaller (larger, respectively) than the starting time (ending time, respectively) of the reservation request. We denote the ending time of a reservation request by $e_r = s_r + l_r$. To find n_r feasible idle periods that satisfy the user request the algorithm proceeds in two phases as follows:

Phase 1: The algorithm first searches for every idle period i with $st_i \leq s_r$ in the tree ${t_q}^s$ associated with the quantum q containing s_r. We refer to such idle periods as *candidate* idle periods since they *only* meet one condition of the feasibility requirement. Recall that they also need to end after e_r in order to be feasible idle periods.

The algorithm starts at the root of the tree and follows left or right subtree based on the value of s_r: If the median starting time of the tree is greater than s_r, the algorithm ignores the left subtree and continues searching recursively in the right subtree. This is because all the idle periods stored in the left subtree have starting times that are larger than the starting time needed by the reservation request (s_r) and hence, do not classify as *candidate* idle periods. If the median starting time is smaller than s_r, the algorithm marks the right subtree and continues to search in the left subtree in a recursive fashion. Note that all the idle periods stored in the right subtree start earlier than the reservation request and therefore, they can all be safely considered as candidate idle periods for the given request. The algorithm stops when it reaches a leaf node. If the idle period stored in the exit leaf node classifies to be a candidate, it is marked. In addition, the algorithm keeps track of the number of candidate idle periods by adding up the size parameter of all marked trees into a temporal counter C_r^s. This enables the algorithm to recognize if there are at most n_r candidate idle periods *before* proceeding to the next phase and thus, improving its efficiency.

Let us refer back to the example presented in Figures 5.1 and 5.2. In Figure 5.2 (b) the algorithm marks the right subtree containing idle periods X and V since both classify as candidates. It then continues to search in the left subtree, marks the tree containing idle period Z and stops when it reaches the leaf containing idle period Y. As mentioned earlier, depending on the value of ${C_r}^s$, the search algorithm continues to the next phase. More specifically, if ${C_r}^s \geq n_r$ it means that there are enough candidate idle periods to be able to satisfy the request and hence, it is safe to proceed to the next phase. Otherwise, the starting time of the request s_r is increased by Δ_t time units and the search algorithm tries to schedule the *modified* request again. Following with our previous example, by the end of phase 1, ${C_r}^s = 4$ and the algorithm continues to phase 2. The value of Δ_t is specificized by the schedule administrator and can be tuned to optimize system and user performance. For instance, applications with high QoS requirements can request the scheduler to retry scheduling their workloads more aggressively, i.e., by choosing small values of Δ_t, in order to minimize the waiting time. This approach would require policy-based mechanisms that are out of the scope of this work. Our algorithm retries to schedule a given request *maxRet* number of times, after which it increases the request time of the reservation by the *horizon* of the schedule, i.e., $q_r = q_r + H$. In essence, the scheduler

puts the request on hold and reconsider it later at time $t = q_r + H$.

Phase 2: If at least n_r *candidate* idle periods are found in Phase 1, the algorithm searches for those idle periods that also meet the condition $et_i \geq e_r$,as this condition guarantees their feasibility for the request being considered. To do this, an algorithm similar to the one presented in Phase 1 is invoked for each of the secondary trees ${t_q}^e$ associated with the trees previously marked, following the reverse order in which they were marked. The algorithm proceeds as follows: Starting at the root, if the median sending time of the tree is larger than e_r it marks the right subtree and continues searching in the left subtree. A temporal counter ${C_r}^e$ is also used in this phase to keep track of the number of feasible idle periods found for r. If the median ending time is smaller than e_r, the algorithm ignores the left subtree and proceeds its search recursively in the right subtree. The algorithm stops whenever it reaches a leaf node or when ${C_r}^e \geq n_r$, whichever happens first. ${C_r}^e \geq n_r$ means that there are n_r feasible idle periods for the request being considered, and in this case the search algorithm invokes an instance of the in-order traversal algorithm. This routine traverses the marked subtrees and ends as soon as n_r idle periods have been retrieved. Similar to Phase 1, if ${C_r}^e < n_r$ the algorithm has failed to find n_r feasible idle periods to satisfy the request and hence s_r is increased by Δ_t units of time.
Following with our running example in Figures 5.1 and 5.2, the algorithm first checks the subtree containing Z and then proceeds to the subtree containing Y. Finally, after verifying that the condition ${C_r}^e = 2$ has been met, the algorithm invokes an in-order traversal subroutine to retrieve idle periods Y and Z.

For each idle period allocated for a given request our algorithm needs to update the data structure by (1) removing the idle period for each of the trees it spans over; and, (2) adding any new idle period created. Recall that at most two new idle periods will be created for each idle period i allocated per request r: $x = (st_x, s')$ and $y = (e'_r, et_y)$. To insert (remove, respectively) a new idle period into the data structure the algorithm first compute the quanta it spans over and then insert (remove, respectively) the idle period in each of them. Note that the updating process can easily be implemented as a background process (multithreading) thus maximizing the efficiency of the system.

Range Search As we mentioned earlier, our algorithm allows for range search of resources. This feature is often required since it permits users to strategically select resources that optimize for their applications and/or meet specific preferences. For example, a user that is interested in reserving resources within a time window $[t_a, t_b]$ may submit a request such that $s_r = t_a$, $l_r = (t_b - t_a)$ and $n_r \geq 1$. The scheduler runs a simplified version of the algorithm which returns the set of resources available (if any) without updating the data structures in the scheduler.

Table 5.1: Features of workload used in the performance evaluation.

Workload	No. of processors (n_r)	No. of jobs	Avg estimated l_r (hours)
CTC	512	39,734	5.82
KTH	128	28,481	2.46
HPC2N	240	202,825	4.72

Later, the user contacts the scheduler to commit the resources resulting from her selection. Due to space constraints we do not include this feature and corresponding performance evaluation in this paper. Instead, we plan on another publication to cover this aspect of the work.

5.4.1 Algorithm Complexity - Worst Case Analysis

In this chapter we have mainly focused on designing an algorithm that is efficient in co-allocating resources. We exploit the logarithmic complexity of balanced binary search trees to achieve this goal. In phase 1 the co-allocation algorithm can mark *at most* $\log v$ subtrees, where v is the number of idle periods in the quantum containing s_r and may be bounded to N. Assuming that for a given request the algorithm advances to phase 2, it takes $\log v$ operations in each of the subtrees marked in Phase 1 for the algorithm to determine if it can satisfy the request. In the worst case the algorithm retrieves the n_r feasible idle periods by invoking an in-order traversal subroutine which requires $O(n_r)$ operations. In addition, the algorithm needs to update each of the trees containing the n_r feasible idle periods. This steps accounts for $O(n_r \times Q \times \log v^2)$. Therefore, the overall complexity of our co-allocation algorithm per quantum is $O(n_r \times m \times \log v^2 + n_r)$. In the worst-case scenario the algorithm is invoked multiple times per request, i.e, as many times as s_r can be increased; we discuss our experience with this parameter in Section 5.5.

5.5 Performance Evaluation

We use simulation-based experiments to evaluate the performance of our algorithm. The experiments are based on discrete event-based simulations, where a workload is needed to drive the simulation. We utilize three real workloads obtained from the Parallel Workload Archive [79] to investigate the performance of our co-allocation algorithm; all of them represent medium-large size Grid systems in use today, have been fully sanitized and have been used in numerous research works [79]. Table 5.1 summarizes each of these workloads. All three systems implement some variation of a batch scheduler where jobs are placed into one or multiple queues waiting for resources to become available before execution.

Table 5.2: Terminology used in co-allocation model

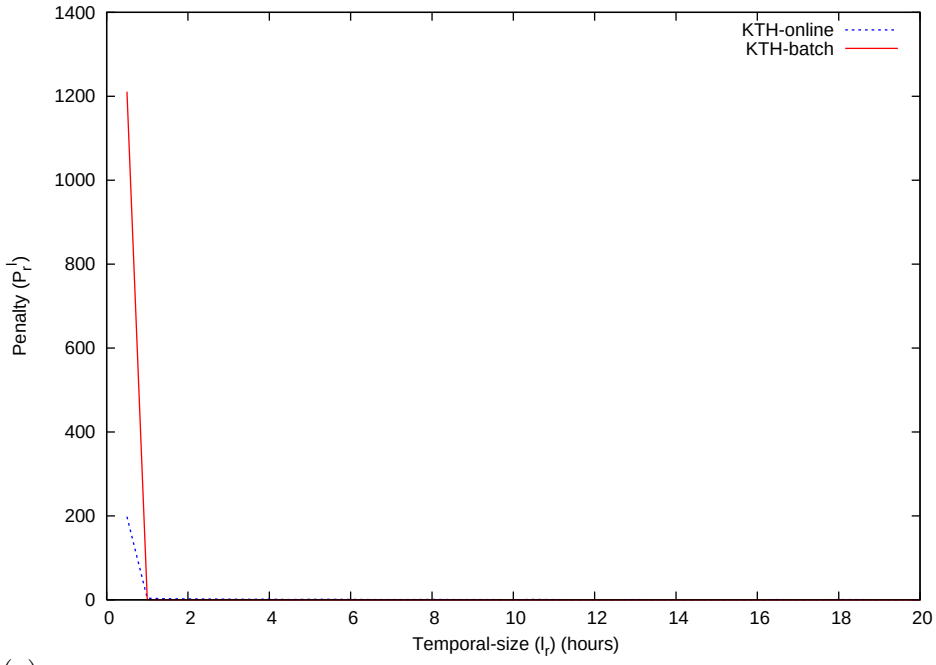
Symbol	Description
$\mathcal{S}$	Scheduler
id_i	Id of server offering idle period i
H	Horizon of the system
N	Total number of servers in the system
Q	Number of quanta ($\lceil \frac{H}{l_q} \rceil$)
s'_r	Time at which reservation starts
r	Reservation represented by tuple (q_r, s_r, l_r, n_r)
T_q	2-dimensional tree associated with quantum q
s_r	Earliest time r can start
t_q^s	T_q's first dimension containing idle periods sorted in function of their starting time
l_r	Duration of r
t_q^e	T_q's second dimension containing idle periods sorted in function of their ending time
e_r	$s_r + l_r$
l_q	Length of quantum
n_r	Number of resources requested by r
${C_r}^s$	Counter with no. of candidates idle periods for reservation request r in Phase 1
q_r	Request time for reservation r
${C_r}^e$	Counter with no. feasible idle periods for reservation request r in Phase 2
st_i	Starting time of idle period i
$maxRet$	Maximum number of times the algorithm tries to reschedule
et_i	Ending time of idle period i
Δ_t	Time units increase for each schedule retrial

Each log entry in the traces corresponds to a single job and contains information about the job such as starting time, expected running time, submission time, ending time, number of processors, user id, computer id, and waiting time. Recall that in our model a request r is represented by the four-parameter tuple (q_r, s_r, l_r, n_r). Therefore, for the purpose of our study we extracted the same four parameters per log entry. Parameter $maxRet$ was set to $\frac{Q}{2}$, i.e., half the number of quanta and Δ_t to 15 minutes for our experiments. Other larger values were tried but no significant improvement in performance was obtained for those values.

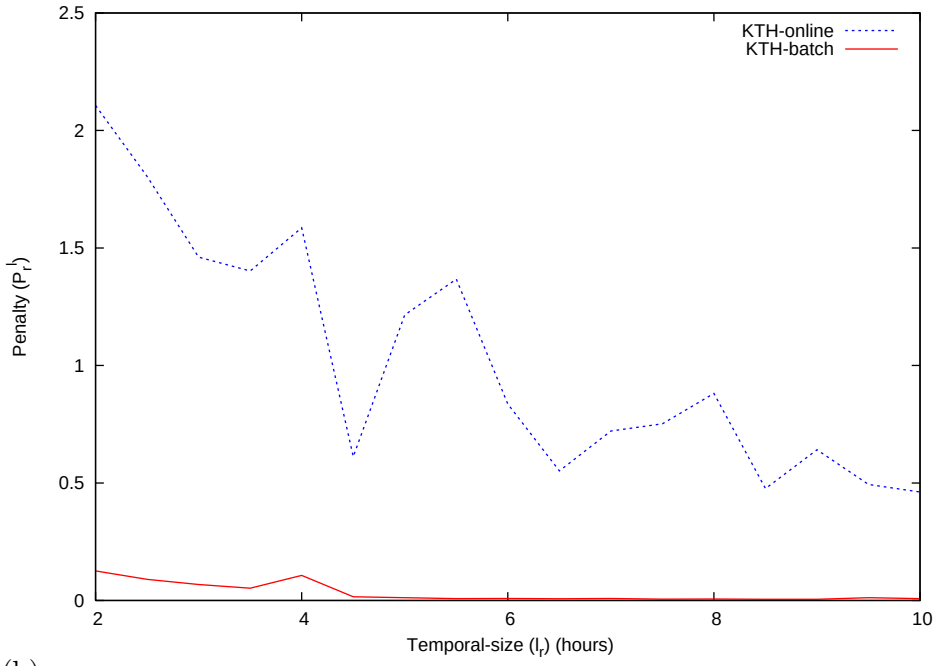
In our study we use three performance metrics:

- **Waiting Time (W_r)** is a measure of the QoS perceived by the user and refers to the time between the earliest time the job can start execution (s_j) and the actual time it starts execution, denoted by (s'_j).

- **Temporal-Penalty (${P_r}^l$)** is a measure of the fairness experienced by the user and is defined as: ${P_r}^l = \frac{W_r}{l_r}$. In other words, P_r captures the waiting time as a function of the duration of the job. Intuitively, lower values correspond to a more fair treatment of jobs.



(a)



(b)

Figure 5.3: (a) Penalty (P_j) (KTH). (b) Zoom-in of mid-tail.

Ideally $P_r < 1.0$. In practice, however, this usually does not hold for small jobs. This will be further explained when we discuss the results later in this section.

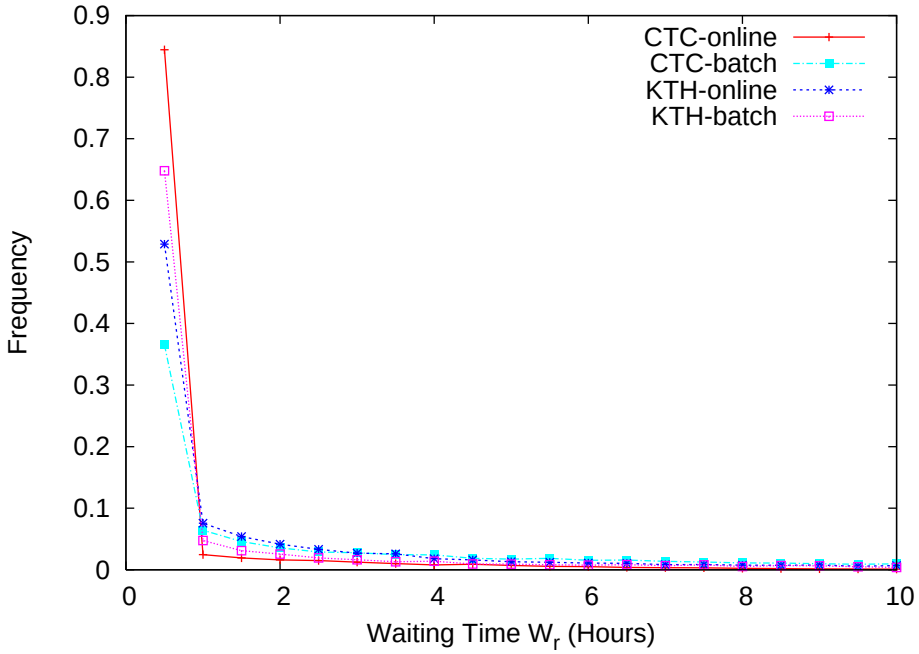


Figure 5.4: Waiting time (W_j) distribution (CTC and KTH).

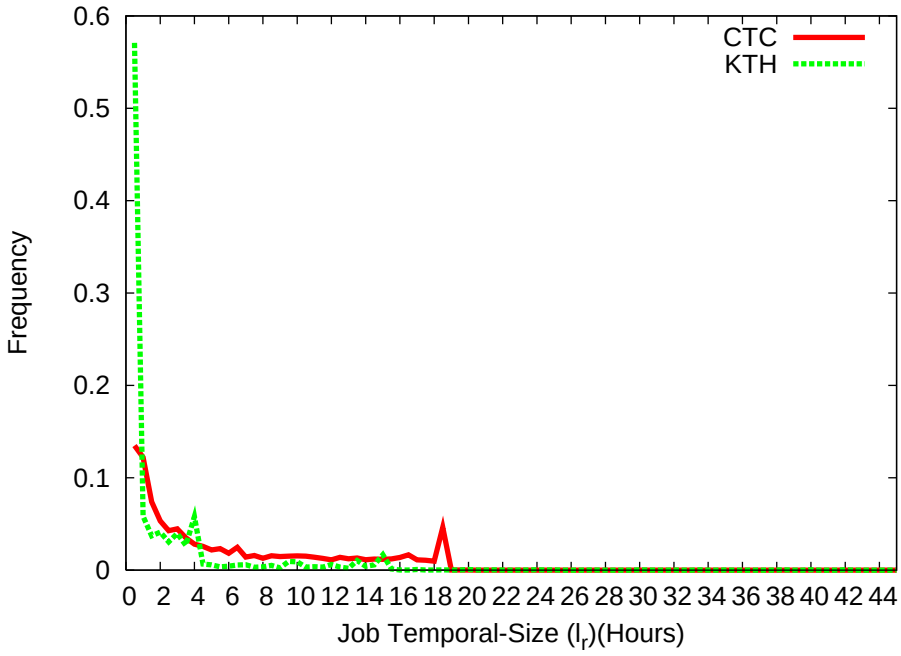


Figure 5.5: Temporal-size distribution (l_r) for workloads CTC and KTH.

- **Spatial-Penalty ($P_r{}^n$)** also measures the fairness experienced by the user and is represented by the average W_r as a function of the spatial-size of the job (n_r). Intuitively, the

larger the number of resources needed by a given job the harder it is for $\mathcal{S}$ to schedule the job and therefore, the longer the waiting time.

We organize our performance evaluation in two parts. First, we investigate the performance of our on-line algorithm against the performance obtained from the traces where batch scheduling is used. Second, we study the impact of introducing support for advance reservations in our algorithm.

5.5.1 On-line Co-Allocation vs. Batch Scheduling

In this part of the performance evaluation we investigate the difference in performance between our on-line co-allocation algorithm and the batch scheduling algorithm. To achieve this we run our simulation driven by the workload traces and assume that $q_r = s_r$. In other words, the co-allocation algorithm tries to schedule the request as soon as it is received. If it fails to satisfy the request then it increments s_r as described in Section 5.4.

Figure 5.3 (a) plots the temporal-penalty ($P_j{}^l$) experienced by jobs when using both scheduling algorithms for KTH workload. We observe that small jobs experience a higher temporal-penalty – a factor of six – under the batch scheduler as compared to our online co-allocation algorithm. A more careful look at the mid-tail of both curves ($2\ hours \leq l_r \leq 10\ hours$) in Figure 5.3 (b) shows that larger jobs are more penalized under our online algorithm. Similar results were found for the other two workloads: for the lack of space and to preserve graph clarity, we are omitting those results in here. These results are somehow counter-intuitive considering that most batch schedulers implement some sort of backfilling, i.e., allow small jobs to leap ahead in the queue as long as they don't delay the reservation corresponding to the job at the head of the queue, and therefore it is expected that small jobs would experience a relative lower degree of penalty. A more detailed analysis of the traces reveals that our algorithm is more efficient in finding idle periods to allocate incoming small jobs without delaying them much. This is reflected by the small number of times that the algorithm needs to reschedule (by increasing s_r) small jobs as observed in our simulations. We further discuss this observation later in this section. On the other hand, batch schedulers find it difficult to recognize resources available to fit small jobs due to the high fragmentation of resources and the dominant presence of jobs that are large in spatial and temporal dimension.

Figure 5.4 (a) shows the waiting time (W_r) distribution for both algorithms under two workloads, CTC and KTH. Let us first consider the curves for the CTC workload. Under our online scheduler most jobs have a waiting time smaller than 2 hours. This is an improvement

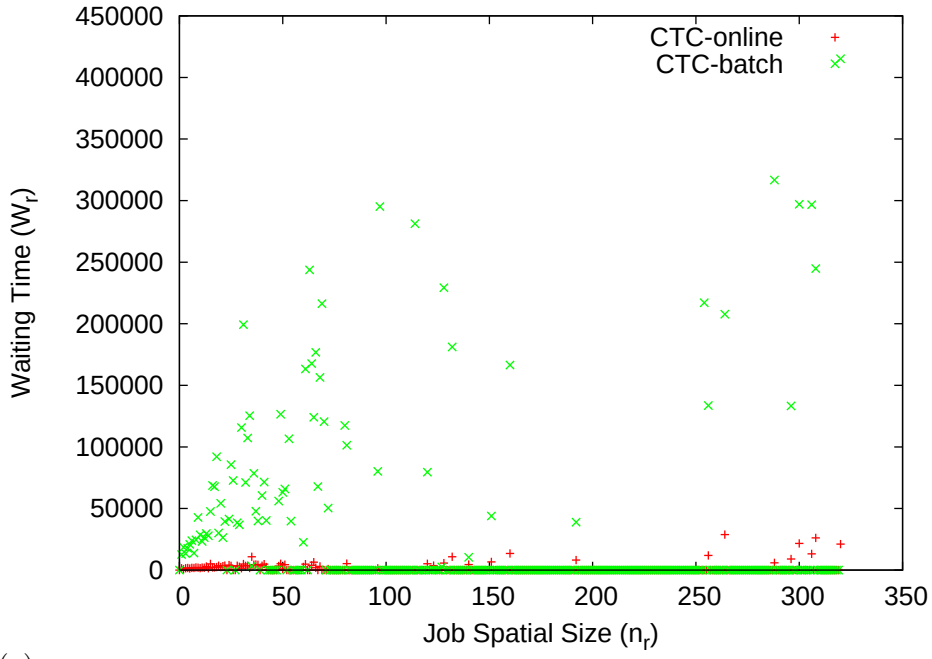
of nearly a factor of two when compared to the batch scheduler. Furthermore, the tail length of both curves differ by hundreds of hours, with the maximum waiting times being 19 hours for our online scheduler and much higher 674 hours for the batch scheduler. The results for the KTH workload are slightly different in that more jobs wait for less than 1 hour under the batch scheduler as compared to our algorithm. However, the waiting time distributions even out before 2 hours. Similar to the CTC workload, the length of the tails differ by a couple of hundred hours with the maximum waiting time for the online scheduler and the batch scheduler being 75 hours and 272.5 hours, respectively.

We make two major observations regarding these results. First, as shown in Figure 5.4 (b), most jobs in the KTH workload have a duration smaller than 2 hours. This results in higher resource fragmentation, impeding the scheduling algorithm from finding feasible idle periods to allocate for incoming jobs. This is in contrast to the CTC workload where at most 14% of all jobs are smaller than 2 hours. This suggests that the performance of our algorithm is not oblivious to the workload, which is a common observation in most scheduling algorithms. Second, the large difference in the tail length for both workloads suggests that by keeping a look-ahead view till the horizon H our algorithm can pack incoming jobs more efficiently and hence improve the utilization of the system.

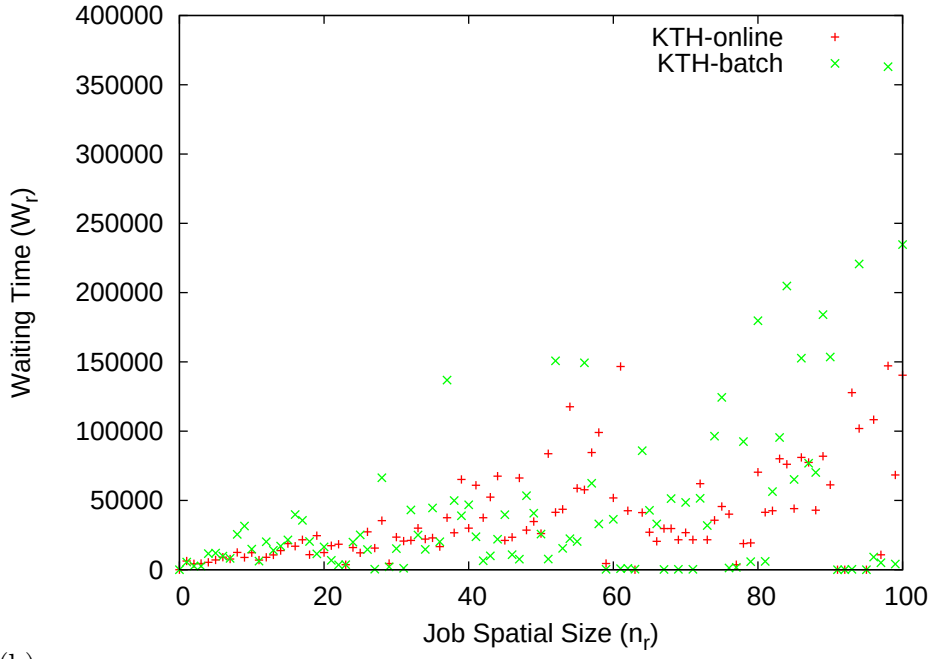
To finalize this part of our evaluation, in Table 5.3 we present the number of retrials, i.e., number of times the scheduler increments s_r per request, as a function of the spatial size for workloads CTC and KTH. In order to obtain results that are statistically significant we calculate the average number of retrials as a function of n_r in groups of 50 servers. For instance, the first column corresponds to the average number of retrials for jobs such that $0 < n_r \leq 50$. Blank spaces in the table represent cases in which there were no requests with n_r values within the corresponding range. It can be observed from the Table that as n_r increases the number of retrials increases as well. This can be explained from the fact that as jobs demand more resources the fragmentation in the system increases and it becomes harder for the algorithm to schedule incoming requests. We also observe a larger number of retrials for the KTH workload as compared to CTC workload. This is due to the fact that KTH exhibits higher fragmentation as a result of KTH's temporal size distribution (see Figure 5.4 (b)).

Table 5.3: Number of retrials as a function of the spatial size of the workloads for CTC and KTH

Workload/(n_r)	(0:50]	(50:100]	(100:150]	(150:200]	(250:300]	(350:400]
CTC (No. of retrials)	2.96	5.34	7.22	13.25	—	127.44
KTH (No. of retrials)	10.27	60	120	—	—	—

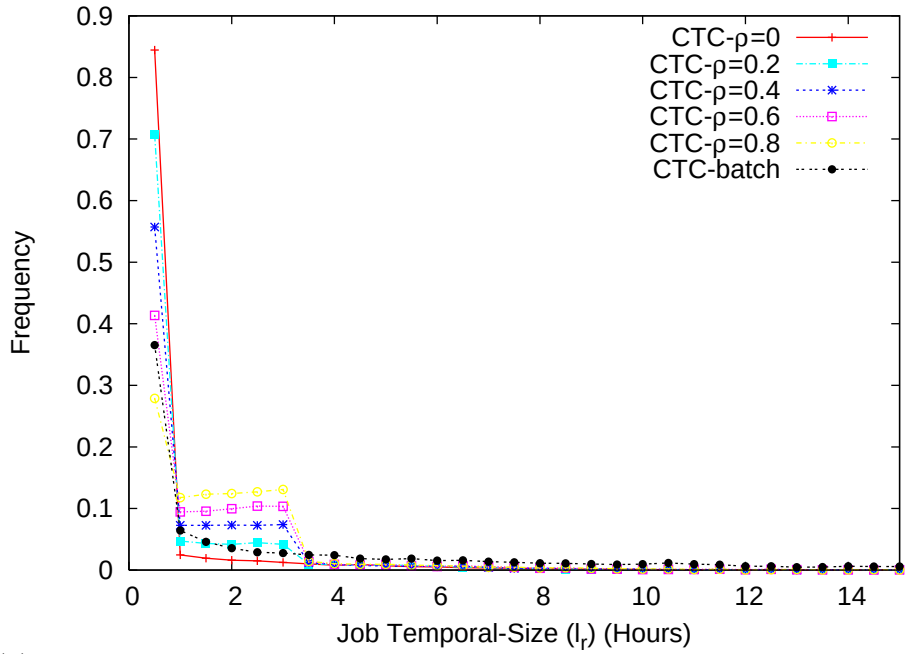


(a)

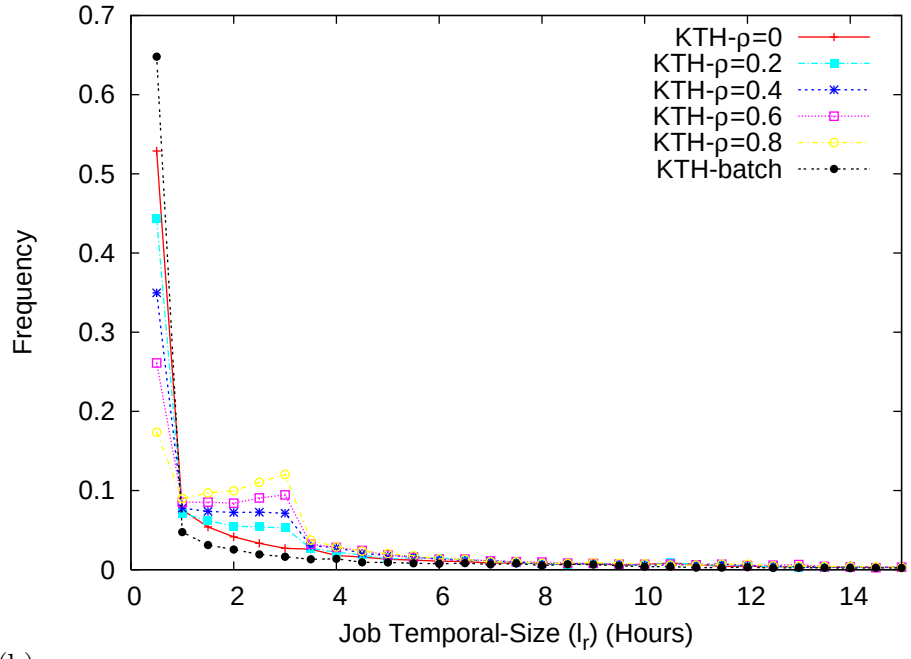


(b)

Figure 5.6: (a) Average Waiting Time (W_t) as a function of job spatial-size for the CTC workload. (b) Average Waiting Time (W_t) as a function of job spatial-size for the KTH workload.



(a)



(b)

Figure 5.7: (a) Waiting time distribution (W_t) for CTC workload. (b) Waiting time distribution (W_t) for KTH workload.

5.5.2 On-line Co-Allocation with Advance Reservations

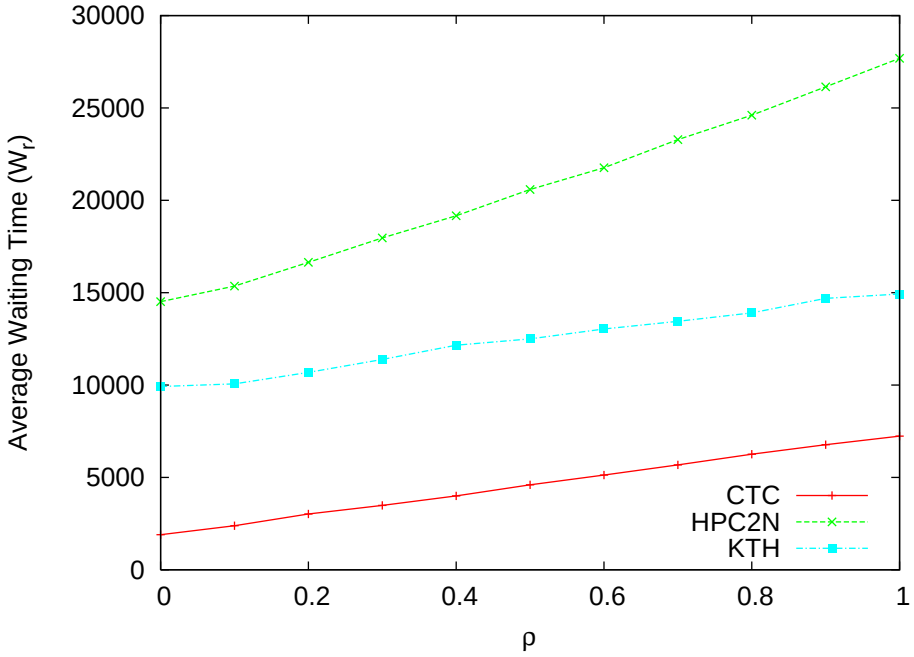


Figure 5.8: Average waiting time (W_r) between s_r and s'_r for workloads CTC, KTH and $HPC2N$.

Due to the fact that advance reservations are not widely implemented in existing systems, there are no workload traces in the Parallel Workload Archive [79] that represent the advance reservation model. In order to evaluate the performance of our algorithm we generated advance reservation requests by randomly choosing jobs from the workload traces according to a desired proportion of advance reservations in the experiment. We denote the fraction of jobs with advance reservations in the system by ρ. For any advance reservation request we randomly set its requested start time (s_r) to be within zero to three hours in the future, as in [27].

Figure 5.7 shows the waiting time distribution W_t for different values of ρ for workloads CTC and KTH, respectively. In both graphs we observe a peak around 3 hours: this is a consequence of setting the requested start time to be within zero to three hours as mentioned earlier. We observe that as ρ increases in the range [0:3] hours the W_t distribution varies for both CTC and KTH. However, the tail lengths for both remains the same. This observation indicates that the QoS perceived by the user is oblivious to the number of advance reservations in the system. We also notice that under the CTC workload our algorithm outperforms the batch scheduling algorithm for multiple values of ρ. This is in contrast to the results plotted in Figure 5.7 (b) for KTH where batch scheduling performs better than our co-allocation algorithm for all values of ρ. Nevertheless, as discussed for Figure 5.4 the tail for the batch scheduler is

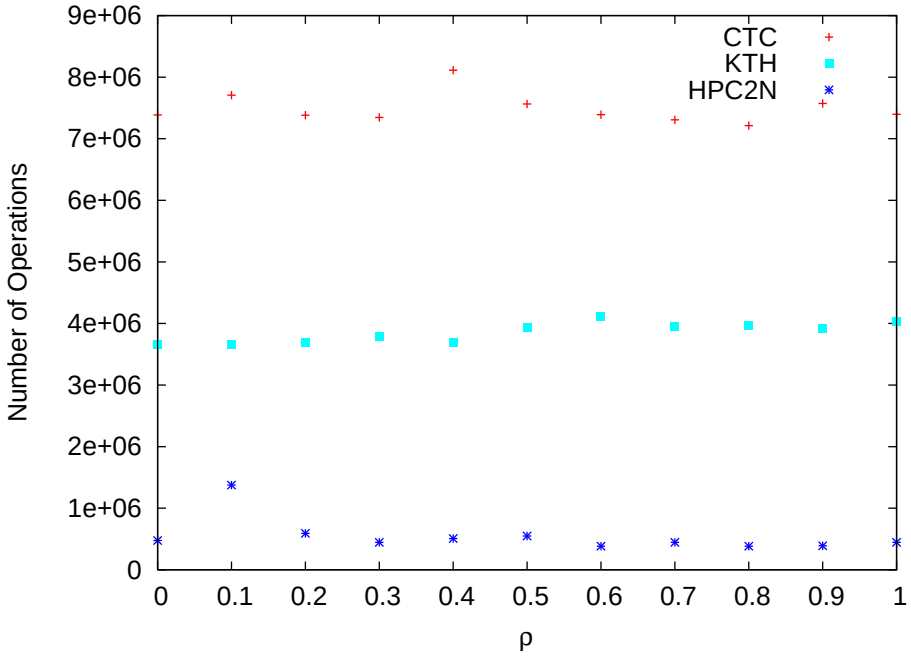


Figure 5.9: Number of operations as a function of ρ for workloads CTC, KTH and $HPC2N$.

significantly longer than for the online algorithm.

Figure 5.8 presents the average waiting time W_r against ρ. We vary ρ in the range $0 \leq \rho \leq 1$, where $\rho = 0$ corresponds to the case in which advance reservations are not supported and shown by our previous experiment; and, $\rho = 1$ represents all the jobs using advance reservations. We observe that W_r increases as ρ increases. This follows intuition since by increasing ρ we effectively increase the waiting time of more jobs in the system and hence, higher overall waiting time is observed.

Figure 5.9 depicts the average number of operations performed by the scheduling algorithm to schedule a request r as a function of ρ under the three workloads. The graph shows that our algorithm scales well as the fraction of advance reservations increases. The reasoning behind this observation is that when performing advance reservations it is more likely that the algorithm will find resources available without having to search in multiple quanta, resulting in few retrials. On the other hand, when scheduling incoming jobs immediately, i.e., $s_r = q_r$ the scheduler is more likely to search further quanta after searching in the quantum containing s_r. Therefore, even though increasing ρ increases resource fragmentation in the system, and hence larger binary search trees are expected, the number of operations remains relatively constant due to the few number of quanta searched and the balancing feature of the binary search trees being used.

It is worth mentioning that events such as servers going down for maintenance are difficult to infer from the workload traces. Nevertheless, we feel that such events have little impact on the results due to the variety of workloads and their large sizes. Evidence of this is the high consistency found across the three workloads for different performance metrics.

5.6 Concluding Remarks

In this work we have considered the problem of co-allocation of resources in large scale distributed systems. We have developed an online co-allocation algorithm that is *efficient* in co-allocating resources while providing support for advance reservations (QoS guarantees) and range search. We achieve this by partitioning the temporal space into a set of quanta and by using efficient 2-dimensional balanced search trees to organize the co-allocations. We have also performed an in-depth comparative analysis of our algorithm against conventional batch schedulers under real workloads. Our results provide some insightful conclusions indicating that online scheduling algorithms may achieve–under most conditions–high overall rate of utilization, while providing smaller delays and better QoS guarantees without adding much complexity. We also showed that our co-allocation algorithm scales to systems with large number of resources and heavy workloads.

Chapter 6

Summary and Future Work

In this thesis we have provided a practical and efficient solution to the problem of scheduling resources in the emerging highly dynamic Grid environments. More specifically, we have developed efficient algorithm implementations that support advance reservations and co-allocation of resources.

To tackle the problem of supporting advance reservations in homogeneous and heterogeneous environments we employed concepts from computational geometry and devised efficient balanced search trees. By doing this, we were able to design algorithms that are efficient and flexible in handling advance reservations. A comprehensive performance evaluation study using simulation was conducted to demonstrate that the proposed strategies perform well across several user and system performance metrics.

We also developed an online co-allocation algorithm that is efficient in co-allocating resources by means of partitioning the temporal space and by using 2-dimensional balanced search trees to organize the availability of resources. A comparative analysis of our algorithm against batch scheduling was also performed. We concluded that online scheduling algorithms may achieve–under most conditions- high utilization, while providing smaller delays and better QoS guarantees without adding much complexity.

6.1 Future Work

Our work can be extended in several directions.

1. **Extensions to Cloud environments.** More specifically, we will research ways of introducing predictability into the Cloud by applying advance reservations. By doing this we hope to devise algorithms that make the Cloud more proactive and less reactive.

2. **Workloads of non-deterministic duration.** In thesis we assume that users know *a priori* the duration of jobs submitted into the Grid. We would like to relax this assumption by over or under estimating the length of the reservations according to historical data of workloads and system utilization.

3. **In depth experimental and theoretical investigation of batch scheduling vs. advance reservations.** The results obtained from our work on resource co-allocation led us to the conclusion that advance reservations might achieve better performance as compared to conventional batch scheduling. We would like to perform an in depth study to help us shed some lights on this observation.

Bibliography

[1] Amazon. Amazon Elastic Computing Cloud EC2. `http://www.amazon.com/gp/browse.html?node=201590011`.

[2] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, 2003.

[3] Jarek Nabrzyski, Jennifer M. Schoff, and Jan Weglarz. *State of the Art and Future Trends*. Kluwer Academic Publishers, 2004.

[4] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of Grid resource management systems for distributed computing. *Software Practice and Experience*, 32(2):135–164, February 2002.

[5] Mohan K. Ramamurthy and Kelvin K. Droegemeier. Linked Environments for Atmospheric Discovery (LEAD): A cyberinfrstructure for mesoscale meteorology research and education. In *Conference of Interactive Information Processing Systems for Meteorology, Ocenoegraphy and Hydrology*, 2004.

[6] Ewa Deelman, Carl Kesselman, Gaurang Mehta, Leila Meshkat, Laura Pearlman, Kent Blackburn, Phil Ehrens
, Albert Lazzarini, Roy Williams, and Scott Koranda. GriPhyN and LIGO, Building a virtual data grid for gravitational wave scientists. In *IEEE International Symposium on High Performance Distributed Computing*, pages 225–234, Washington, DC, USA, July 2002. IEEE Computer Society.

[7] Phosphorus. `http://www.fz-juelich.de/jsc/grid/PHOSPHORUS/`.

[8] Sam Averitt, Michale Bugaev, Aaron Peeler, Henry Schaffer, Eric Sills, Sarah Stein, Josh Thompson, and Mladen Vouk. The virtual computing lab. In *International Conference*

on Virtual Computing Initiative, pages 1–16, Research Triangle Park, NC, May 2007. IBM Corporation.

[9] Daniel S. Katz, Joseph C. Jacob, G. Bruce Berriman, John Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, and Gurmeet Singh. A comparison of two methods for building astronomical image mosaics on a grid. In *International Conference on Parallel Processing Workshops (ICPP)*, pages 85–94, Oslo, Norway, June 2005. IEEE Computer Society.

[10] Philip Maechling, Hans Chalupsky, Maureen Dougherty, Ewa Deelman, Yolanda Gil, Sridhar Gullapalli, Vipin Gupta, Carl Kesselman, Jihic Kim, Gaurang Mehta, Brian Mendenhall, Thomas Russ, Gurmeet Singh, Marc Spraragen, Garrick Staples, and Karan Vahi. Simplifying construction of complex workflows for non-expert users of the southern california earthquake center community modeling environment. *ACM SIGMOD Record*, 34(3):24–30, 2005.

[11] William Leinberger and Vipin Kumar. Information power Grid: The new frontier in parallel computing? *IEEE Concurrency*, 7(4):75–84, 1999.

[12] Eun-Kyu Byun, Jae-Wan Jang, Wook Jung, and Jin-Soo Kim. A dynamic Grid services deployment mechanism for on-demand resource provisioning. In *5th. IEEE International Symposium on Cluster Computing and the Grid*, volume 2, pages 863–870, Cardiff, UK, 2005. IEEE Computer Society.

[13] Sun Microsystems. Sun grid. `http://www.sun.com/service/sungrid/`, April 2006.

[14] Rashid J. Al-Ali1, Kaizar Amin, Gregor von Laszewski, Omer F. Rana1, David W. Walker1, Mihael Hategan, and Nestor Zaluzec. Analysis and provision of QoS for distributed grid applications. *Journal of Grid Computing*, 2(2):163–182, 2004.

[15] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational Grid. In *4th. International Conference on High Performance in Asia-Pacific Region*, pages 283–289, Beijing, China, 2000. IEEE Computer Society Press.

[16] Rajkumar Buyya, David Abramson, and Srikumar Venugopal. The grid economy. In *the IEEE*, volume 93, pages 698–714, March 2005.

[17] Avraham Leff, James T. Rayfield, and Daniel M. Dias. Service-level agreements and commercial grids. *IEEE Internet Computing*, 7(4):44–50, July 2003.

[18] Dang Minh Quan and Odej Kao. On architecture for sla-aware workflows in grid environments. In *International Conference on Advanced Information Networking and Applications*, pages 287–292, Washington, DC, USA, 2005. IEEE Computer Society.

[19] David Jackson. New issues and new capabilities in HPC scheduling with the Maui scheduler. `http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF01/Jackson\_Utah.pdf`.

[20] David B. Jackson, Quinn Snell, and Mark J. Clement. Core algorithms of the Maui scheduler. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, Lectures in Computer Science, pages 87–102, London, UK, 2001. Springer-Verlag.

[21] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor – A hunter of idle workstations. In *IEEE 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[22] San Diego Super Computing Center. Catalina Batch Scheduler. `http://www.sdsc.edu/catalina/`.

[23] I.B.M. . Loadleveler. `http://www-03.ibm.com/systems/clusters/software/loadleveler/index.html`.

[24] Brett Bode, David M. Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *ALS'00: Proceedings of the 4th conference on 4th Annual Linux Showcase & Conference, Atlanta*, pages 27–27, Berkeley, CA, USA, 2000. USENIX Association.

[25] Platform Computing Corporation. LSF. `http://www.platform.com`.

[26] Erik Elmroth and Johan Tordsson. A grid resource broker supporting advance reservations and benchmark-based resource selection. *Applied Parallel Computing*, 3732:1061–1070, 2006.

[27] Warren Smith, Ian Foster, and Valerie Taylor. Scheduling with advance reservations. In *14th. IEEE International Parallel and Distributed Processing Symposium*, pages 127–132, Cancun, Mexico, May 2000.

[28] Rui Min and Muthucumaru Maheswaran. Scheduling advance reservations with priorities in Grid computing. In *Parallel and Distributed Computing Systems*, pages 172–1176, 2001.

[29] Anthony Sulistio and Rajkumar Buyya. A Grid simulation infrastructure supporting advance reservation. In *Parallel and Distributed Computing Sysems*, pages 1–7, 2004.

[30] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. Performance impact of resource provisioning on workflows applying. Technical, University of Southern California, 2005.

[31] Henan Zhao and Rizos Sakellariou. Advance reservation policies for workflows. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 47–67, June 2006.

[32] Marek Wieczorek, Mumtaz Siddiqui, Alex Villazon, Radu Prodan, and Thomas Fahringer. Applying advance reservation to increase predictability of workflow execution on the grid. In *2nd IEEE International Conference on e-Science and Grid Computing*, pages 82–82, Washington, DC, USA, December 2006. IEEE Computer Society.

[33] Andrew Stephen Mcgough, Ali Afzal, John Darlington, Nathalie Furmento, Anthony Mayer, and Laurie Young. Making the Grid predictable through reservations and performance modelling. *Compute Journal*, 48(3):358–368, 2005.

[34] Lionel Eyraud-Dubois, Gregory Mounie, and Denis Trystram1. Analysis of scheduling algorithms with reservations. In *IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[35] Mumtaz Siddiqui, Alex Villazón, and Thomas Fahringer. Grid capacity planning with negotiation-based advance reservation for optimized qos. In *ACM/IEEE Conference on Supercomputing*, page 103, New York, NY, USA, 2006. ACM.

[36] Dean Kuo and Mark Mckeown. Advance reservation and co-allocation protocol for grid computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 164–171, Washington, DC, USA, 2005. IEEE Computer Society.

[37] Ian Foster and Alain Roy. Quality of service architecture that combines resource reservation and application adaptation. In *International Workshop on Quality of Service*, pages 161–188, June 2000.

[38] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt, and Alain Roy. A distributed resource management architecture that supports advance reservations and co-

allocation. In *7th International Workshop on Quality of Service*, pages 27–36, London, UK, 1999.

[39] Gregor von Laszewski, Joseph A. Insley, Ian Foster, John Bresnahan, Carl Kesselmany, Mei Suy, Marcus Thiebauxy, Mark L. Rivers, Steve Wang, Brian Tiemanz, and Ian McNultyz. Real-time analysis, visualization, and steering of microtomography experiments at photon sources. In *9th. SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, US, March 1999.

[40] Control Dynamic Multi-Terabit Core Optical Networks: Architecture, Protocols and Management (CORONET). `http://www.darpa.mil/STO/solicitations/CORONET/index.htm`.

[41] I.B.M. IBM Blue Cloud. `http://www-03.ibm.com/press/us/en/pressrelease/22613.wss`.

[42] Abhijit Bose, Brian Wickman, and Cameron Wood. Mars: A metascheduler for distributed resources in campus grids. In *5th. IEEE/ACM International Workshop on Grid Computing*, pages 110–118, Pittsburgh, PA, US, November 2004. IEEE Computer Society.

[43] J. M. Alonso, V. Hernandez, and G. Molto. Towards on-demand ubiquitous metascheduling on computational grids. In *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 84–90, Napoli, Italy, February 2007. IEEE Computer Society.

[44] Sathish S. Vadhiyar and Jack Dongarra:. A metascheduler for the grid. In *11th IEEE International Symposium on High Performance Distributed Computing*, page 343, Edinburg, Scotland, 2002.

[45] Borja Sotomayor, Kate Keahey, Ian Foster, and Tim Freeman. Enabling cost-effective resource leases with virtual machines. In *IEEE International Symposium on High Performance Distributed Computing*, pages 1–3, Monterey, CA, US, June 2007.

[46] T. Fahringer, R. Prodan, Rubing Duan, F. Nerieri, S. Podlipnig, Jun Qin, M. Siddiqui, Hong-Linh Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid application development and computing environment. In *IEEE/ACM International Workshop on Grid Computing*, pages 122–131, Washington, DC, USA, 2005. IEEE Computer Society.

[47] Andrew Stephen Mcgough, Ali Afzal, John Darlington, Nathalie Furmento, Anthony Mayer, and Laurie Young. Making the grid predictable through reservations and performance modelling. *The Computer Journal*, 48(3):358–368, 2005.

[48] Umar Farooq, Shikharesh Majumdar, and Eric W. Parsons. Impact of laxity on scheduling with advance reservations in Grids. In *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 319–324, Washington, DC, USA, September 2005. IEEE Computer Society.

[49] T. Roblitz, F. Schintke, and Reinefeld. Resource reservations with fuzzy requests. *Journal on Concurrency and Computation: Practice and Experience*, 18(13):1681–1703, November 2006.

[50] Gurmeet Singh, Carl Kesselman, and Ewa Deelman. A provisioning model and its comparison with best-effort for performance-cost optimization in Grids. In *International Symposium on High Performance Distributed Computing*, pages 117–126, New York, NY, USA, June 2007. ACM.

[51] Rajkumar Buyya and Srikumar Venugopal. The gridbus toolkit for service oriented grid and utility computing: An overview and status report. In *International Workshop on Grid Economics and Business Models*, April 2004.

[52] Robert L. Henderson. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294, London, UK, April 1995. Springer-Verlag.

[53] GENIAS Sofware GmBH. Codine: Computing in distributed networked environments. `http:www.genias.de/genias/english/codine.html`, 1995.

[54] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn:. Parallel job scheduling – a status report. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, Lectures Notes in Computer Science, Springer, pages 1–16, Cambridge, MA, US, 2004. Springer Berlin / Heidelberg.

[55] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1294 of *Lecture Notes in Computer Science*, pages 1–34, London, UK, April 1997. Springer-Verlag.

[56] Steve J. Chapin, Dimitrios Katramatos, John F. Karpovich, and Andrew S. Grimshaw. The legion resource management system. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 1659, pages 162–178, San Juan, Puerto Rico, 1999. Springer-Verlag.

[57] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource co-allocation in computational grids. In *8th. IEEE International Symposium on High Performance Distributed Computing*, pages 219–228, Redondo Beach, 1999.

[58] Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *International Workshop on Job Scheduling Strategies for Parallel Processing*, Lectures in Computer Science, pages 62–82, Orlando, Florida, US, 1998. Springer-Verlag.

[59] Thomas Robitz, Florian Shintke, and Jan Wendler. Elastic grid reservations with user-defined optimization policies. In *Workshop on Adaptive Grid Middleware*, September 2004.

[60] Ammar H. Alhusaini, Viktor K. Prasanna, and C. S. Raghavendra. A framework for mapping with resource co-allocation in heterogeneous computing systems. In *Proceedings of the 9th Heterogeneous Computing Workshop*, page 273, Washington, DC, USA, 2000. IEEE Computer Society.

[61] Ammar H. Alhusaini, C. S. Raghavendra, and Viktor K. Prasanna. Run-time adapation for grid environments. In *Heterogeneous Computing Workshop*, pages 864–874, San Francisco, CA, US, April 2001.

[62] Carsten Ernemann, Volker Hamscher, Uwe Schwiegelshohn, and Ramin Yahyapour. On advantages of of grid computing for parallel job scheduling. In *International Workshop in Grid Computing*, pages 219–231, Berlin, Germany, May 2002.

[63] Anca I. D. Bucur and Dick H. J. Epema. Scheduling policies for processor coallocation in multicluster systems. *IEEE Trans. Parallel Distrib. Syst.*, 18(7):958–972, 2007.

[64] Weizhe Zhanga, Albert M. K. Chengb, and Mingzeng Hu. Multisite co-allocation algorithms for computational grid. In *20st. International Parallel and Distributed Processing Symposium*, pages 1–8, Rhodes Island, Greece, April 2006.

[65] Lizhe Wang, Wentong Cai, Bu-Sung Lee, Simon See, and Wei Jie. Resource co-allocation for parallel tasks in computational grids. In *CLADE '03: Proceedings of the International*

Workshop on Challenges of Large Applications in Distributed Environments, pages 88–95, Washington, DC, USA, June 2003. IEEE Computer Society.

[66] Joerg Decker and Joerg Schneider. Heuristic scheduling of grid workflows supporting co-allocation and advance reservation. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 335–342, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Atsuko Takefusa, Satoshi Matsuoka, Henri Casanova, and Francine Berman. A study of deadline scheduling for client-server systems on the computational Grid. In *10th IEEE International Symposium on High Performance Distributed Computing*, pages 406–415, Washington, DC, USA, 2001. IEEE Computer Society.

[68] Eddy Caron, Pushpinder Kaur Chouhan, and Frederic Desprez. Deadline scheduling with priority for client-server systems on the grid. In *5th. IEEE/ACM International Workshop on Grid Computing*, pages 410–414, Washtington, DC, US, November 2004. IEEE Computer Society.

[69] Ho-Leung Chan, Tak-Wah Lam, and Kar-Keung To. Non-Migratory online deadline scheduling on multiprocessors. In *5th. Annual ACM-SIAM symposium on Discrete algorithms*, pages 970–979, Philadelphia, PA, US, 2004. Society for Industrial and Applied Mathematics.

[70] Jinhui Xu, Chunming Qiao, J. Li, and Guang Xu. Efficient burst scheduling algorithms in optical burst-switched networks using geometric techniques. *IEEE Selected Areas in Communications*, 22(9):1796–1811, November 2004.

[71] Mark de Berg, Marc van Kreveld, M .Overmars, and O. Schwarzkopf. *Computational Geometry–Algorithm and Applications*. Springer-Verlag, 2 edition, 2000.

[72] Edward M. McGreight. Priority search trees. *SIAM Jornal of Computing*, 14(2):257–276, 1985.

[73] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Riverst, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, 2 edition, 2001.

[74] Howard Jay Siegel and Shoukat Ali. Techniques for mapping tasks to machines in heterogeneous computing systems. *Journal of Systems Architecture*, 46(8):627–639, 2000.

[75] P. Brighten Godfrey and Richard M. Karp. On the price of heterogeneity in parallel systems. In *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 84–92, New York, NY, USA, August 2006. ACM.

[76] Claris Castillo, George N. Rouskas, and Khaled Harfoush. On the design of online scheduling algorithms for advance reservations and qos in grids. In *21sts IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, Long Beach, California, US, April 2007.

[77] Tsegereda Beyene, Yufeng Xin, Mosaddaq Turabi, and Khalid Raza. Pce based grid networking. In *IEEE Symposium on Computers and Communications*, pages 769–774, Aveiro, Portugal, July 2007.

[78] A. Farrel, J. P. Vasseur, and J. Ash. A path computation element (pce)-based architecture. `http://tools.ietf.org/html/rfc4655`, August 2006.

[79] Parallel Workload Archive. `www.cs.huji.ac.il/labs/parallel/workload`.