Author's personal copy

J. Parallel Distrib. Comput. 71 (2011) 963–973

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Online algorithms for advance resource reservations ☆

C. Castillo [a,*], G.N. Rouskas [b], K. Harfoush [b]

[a] IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, United States
[b] Department of Computer Science, North Carolina State University, Engineering Building 2, 890 Oval Drive, Raleigh, NC 27695, United States

ARTICLE INFO

Article history:
Received 1 July 2010
Received in revised form
24 October 2010
Accepted 12 January 2011
Available online 27 January 2011

Keywords:
Grid computing
Advance reservations
Scheduling
Resource allocation
Resource management

ABSTRACT

We consider the problem of providing QoS guarantees to Grid users through advance reservation of resources. 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. 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. Our main contribution is a timely, practical, and efficient solution to the problem of scheduling resources in emerging on-demand computing environments.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

Grids have emerged as an essential infrastructure for resource-intensive scientific and commercial applications [13,23] and evolved to become a cornerstone of Cloud computing [14]. Grid technology enables the sharing and dynamic allocation of distributed, high-performance computational resources while minimizing the associated ownership and operating costs; it also facilitates access to such resources and promotes flexibility and collaboration among diverse organizations. More recently, the concept of *on-demand computing* [22,6] (Cloud Computing) has emerged as a viable model in which a wide range of *finer grain* 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, Amazon has launched with great success Amazon EC2, a web service that provides resizable compute capacity [2] and more such service offerings are expected in the near future. This vision of computing as a utility is expected to change not only the way scientists and businesses work, but also the way they think about computing resources. However, its realization depends on the development of sophisticated resource management systems capable of allocating resources to users based on agreed upon quality of service (QoS) requirements [1], while satisfying certain system level objectives (e.g., high utilization, economic constraints, etc.) [4,5].

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 [20,21], portable batch system (PBS) [3], and load sharing facility (LSF) [26]; they typically run in batch mode, can be customized to specific policies, and attempt to balance the load among the 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 [23]. For instance, batch systems typically allocate resources to jobs as they become available, without consideration of applications that need to obtain results within a strict deadline [1]. 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 [20,21], but also employ *backfilling* operations, i.e., run jobs out of order, to make better use of the available resources. Unfortunately, backfilling often hinders the ability of the system to provide QoS guarantees. This is due to the fact that in order to optimize for utilization schedulers that support backfilling may have to bypass job priorities set by the system administrator [21].

Advance reservation of resources is one mechanism that Grid providers may employ in order to offer specific QoS guarantees to application users. Advance reservation, i.e., the ability of the scheduler to guarantee the availability of resources at a particular time in the future, increases the predictability of the system and it has been

☆ This work was done while the first author was a Ph.D. student at North Carolina State University.

* Corresponding author.
E-mail addresses: claris@us.ibm.com (C. Castillo), rouskas@ncsu.edu (G.N. Rouskas), harfoush@csc.ncsu.edu (K. Harfoush).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.01.003

Author's personal copy

an area of interest [12,1,23,31,13,25,33] in the Grid community. Although some schedulers, including Maui [20], provide some sort of advance reservation mechanisms, existing approaches to making reservations in the future lack sophistication, are expensive, and do not scale well. This lack of scalability is due primarily to two factors. First, as the number of resources in the Grid increases, the overhead of maintaining and updating the set of advance reservations can be significant, especially if appropriate attention is not paid to the design of the relevant data structures. For instance, common data structures used in practice consist of tables, therefore, basic update operations take time that is linear with the number of reservations. Second, making advance reservations tends to fragment the available resources. If this fragmentation is not taken into account by the scheduling algorithm, the result will be poor utilization and high job rejection rate; on the other hand, algorithms which attempt to utilize the fragmented capacity but are not properly designed will suffer from unacceptably high running times as the number of resources increases. For these reasons, incorporating QoS mechanisms into current Grid environments has proven to be a challenging task [1,13]. In practice, most systems tackle the complexity by limiting both the pool of resources available for advance reservation and the number of users with permission to request reservations. Recently, a 3-layered negotiation protocol for advance reservations for Grids was introduced in [30].

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, hindering 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 paper we present a framework for designing effective and efficient scheduling algorithms that employ advance reservations to guarantee QoS to users. Specifically, 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. Using concepts from computational geometry [10], we show how to manage efficiently the fragmentation of resources due to advance reservations by maintaining an appropriate set of balanced search trees. We also present a set of scheduling strategies for making advance reservations. Each strategy corresponds to a different optimization objective, and requires that the information on the advance reservations be organized and maintained in a slightly different variant of the search tree structure. Our algorithms scale to large Grid systems, and simulation results demonstrate that they perform well across several performance metrics that reflect both user- and system-specific goals.

The rest of the paper is organized as follows. In Section 2 we describe the online scheduling problem we study in this work, and in Section 3 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 4 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 5 we present simulation results to evaluate the various strategies in terms of several performance metrics, and we conclude the paper in Section 6.

2. 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. 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. Fig. 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.

2.1. Online scheduling of real-time tasks

In real-time systems, the *Deadline Scheduling Problem* consists of scheduling a set of m independent tasks with release or ready times and deadlines on n identical processors. A task cannot start until its release time and it must be completed by its deadline [15]. Thus, the scheduler's task is to decide if there is a schedule such that each task can be completely executed within the interval of its release time and deadline. Much of the earlier work has been assuming that tasks are periodic, i.e., infinite sequence of identical activities, called instances, that are invoked within regular time periods. There have also been studies that consider tasks with dynamic arrivals, which can be further classified into aperiodic tasks and sporadic tasks. Aperiodic tasks arise from asynchronous events outside the system and follow specific random distributions; and sporadic tasks have random arrival times and hard deadlines. The problem being considered in our work is equivalent to the deadline scheduling problem for sporadic tasks and therefore we feel compelled to make a comparative analysis of the work done in both contexts—Grid and real-time scheduling. Real-time scheduling is a field that has been studied extensively in the past. In the following we provide a brief survey of this field; for a more comprehensive review, we refer to [32].

Author's personal copy

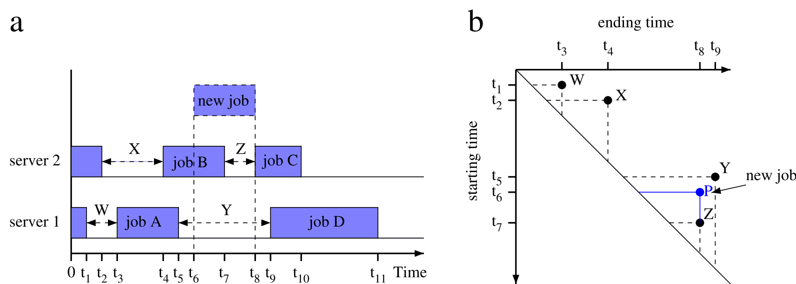

Fig. 1. (a) Advance reservations in a 2-server system: jobs scheduled and idle periods, (b) equivalent geometric representation of the schedule: idle periods as points in the plane.

Scheduling algorithms in real time systems can be categorized into a set of paradigmatic approaches [27]. This categorization is based on three aspects: (a) whether a system performs schedulability analysis, (b) if it does, whether it is done statically or dynamically, and (c) whether the result of the analysis itself produces a schedule or a plan according to which tasks are dispatched at run-time. Based on the categorization resulting from this work, the scheduling problem being considered here falls into the paradigm of *Dynamic Planning-based Scheduling*. This means that the feasibility of a given task is checked at run-time, i.e., a dynamically arriving task is accepted for execution only if it is found feasible. Such a task is said to be *guaranteed* to meet its time constraints. However, it has been shown that dynamic algorithms that do not *a priori* know the arrival times of tasks cannot guarantee optimal performance [11].

Real-time scheduling algorithms can also be classified into *offline* and *online* algorithms based on the knowledge they have about incoming tasks. In offline scheduling, the scheduler has complete knowledge of the task set and its constraints, such as deadlines and computation times. Scheduling decisions are based on fixed parameters assigned to tasks before their activation. Online scheduling algorithms, on the other hand, make their scheduling decisions at runtime. Online schedulers are flexible and adaptive, but they can incur significant overheads because of runtime processing [19]. A scheduler (offline or online) is said to be *optimal* for n processors if it constructs a feasible schedule for *every* task system that is feasible on n processors [16]. In the case of the Deadline Scheduling Problem, however, it has been shown that in general an optimal online scheduler does not exist if one or more task parameters are unknown [11]. Nevertheless, due to the unpredictable nature of the arrival time of tasks, online scheduling has been proposed as a viable mechanism to deal with the time constraints imposed by the tasks [16].

Real-time scheduling algorithms can also be classified into preemptive or non-preemptive algorithms. Non-preemptive scheduling disciplines assume that tasks must run till completion once they start execution, i.e., not be preempted in favor of another task. For non-preemptive disciplines, the Deadline Scheduling Problem appears to be difficult to solve even for very restricted cases. For example, when $n > 1$, where n is the number of processors, the problem is readily seen to be NP-complete even when all tasks have the same release time and deadline [15]. In view of all the aforementioned observations, most schedulers found in the literature assume tasks can be preempted and function in offline mode. For instance, the well known EDF algorithm achieves optimality by using preemption and by assigning priorities to tasks. Furthermore, task preemption has been often assumed in order to support online scheduling.

Let us now review the computational complexity of some important offline algorithms for the Deadline Scheduling Problem. In [17] the authors gave an optimal offline scheduler for any number of processors reducing the scheduling problem to a network-flow problem. This scheduler implementation runs in $O(m^3)$ time, where m is the number of unfinished tasks. Faster optimal offline schedulers for special task systems have also been developed. An online algorithm is described in [29] which runs in $O(n^2m + mn \log m)$ time and *preemptively* schedules m independent tasks in n homogeneous processors. The same author also gave an $O(m \log mn)$ time offline scheduling algorithm for task systems with either one distinct release time or one distinct deadline [28]. An algorithm that also considers the Deadline Scheduling Problem is presented in [36]. One major assumption in this work is that tasks can be preempted, may require additional resources and have the same release time. The heuristics proposed have a running time of $O(nm^2)$ and fall into the dynamic-planning scheduling paradigm. A preemptive offline scheduling algorithm is also presented in [15] with a running time of $O(m \log m + mn)$, creating at most $O(m)$ and $O(mn)$ preemptions for nested systems and non-overlapping task systems, respectively. Seeking to leverage the low complexity of offline algorithms and the high flexibility of online algorithms, a hybrid preemptive scheduling algorithm is proposed in [19,18] for periodic and dynamic tasks.

From the results of the vast body of research addressing numerous variants of the Deadline Scheduling Problem in real-time systems, we can make the following general observations:

- Optimal offline scheduling algorithms exist. However, they require a priori knowledge of the characteristics of the workload.
- Online and dynamic scheduling algorithms offer great advantages in terms of adaptability and flexibility, two desired characteristics in Grid environments. However, in general, they do not offer optimal scheduling and hence we are left with heuristics to tackle the scheduling problem. Furthermore, they have high processing overheads.
- Non-preemptive scheduling problems are more difficult to tackle than their preemptive counterpart problems.
- To develop offline scheduling algorithms that are efficient on scheduling jobs, one must know specific characteristics of the workload, such as the tasks being non-overlapping or self-contained.

Our work differs from the existing literature in that to tackle the lack of efficiency in online scheduling algorithms for the Deadline Scheduling Problems we focus on two aspects that we believe have not been adequately addressed. Firstly, the design of data structures that enable the organization of resources so that they can be searched and updated in a computationally efficient manner that scales to large Grid systems. Secondly, the design and development of algorithms that provide the scheduler with the ability to allocate resources based on system and application criteria. Furthermore, we make the assumption that jobs may not be preempted, thereby increasing the difficulty of the problem space. Nevertheless, we believe that the algorithms we present here can be adapted handle preemption; however, such extensions are outside the scope of this article.

Author's personal copy

Several variants of this scheduling problem with advanced reservations and/or deadlines have been studied in Grid systems [31,25,34,7,8]. However, most of the heuristics are linear in the number of resources in the system and hence may not scale well to utilize the available system capacity efficiently [1,13]. In the next section, we present a new framework for developing efficient algorithms for this problem taking into account a range of optimization criteria.

2.2. Resource scheduling with immediate deadlines

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 advanced 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 Fig. 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. In [37] the authors consider this same task model, i.e., the immediate deadline. The proposed scheduler, however, keeps track of the earliest available time (EAT) of each compute node in the system rather than the latest available time, i.e., denoted by LACT in our work. Note that this difference is well justified given that the scheduling problem considered in [37] does not support advance reservations. Due to the simplicity and low complexity of the LACT algorithm we use it as a baseline to perform a performance analysis of our algorithms.

An algorithm that considers the idle periods when making decisions was developed in [35] in the context of scheduling bursts in optical burst switched networks. The algorithm uses concepts from computational geometry [10] to represent the time intervals corresponding to idle periods as points in a plane, as illustrated in Fig. 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 Fig. 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 Fig. 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 [24] 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.

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

3. 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 Fig. 1. Assuming that the current time $t = 0$, Fig. 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 $(r_j = t_6, l_j = t_8 - t_6, d_j = t_{12})$. Fig. 2(b) is the geometric representation of this schedule. The fact that job j has a general deadline is represented in Fig. 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 this point.

Consider the new job j and its geometric representation in the plane, as shown in Fig. 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 Fig. 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 Fig. 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.

3.1. 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 [35] was developed for the special case of immediate deadlines. Recall also that the algorithm developed in [35] 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 Fig. 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 $l_{\min}$. Fig. 2(b) shows the partitioning of the plane into *horizontal* strips. Alternatively, one might partition the plane into *vertical* strips of

Author's personal copy

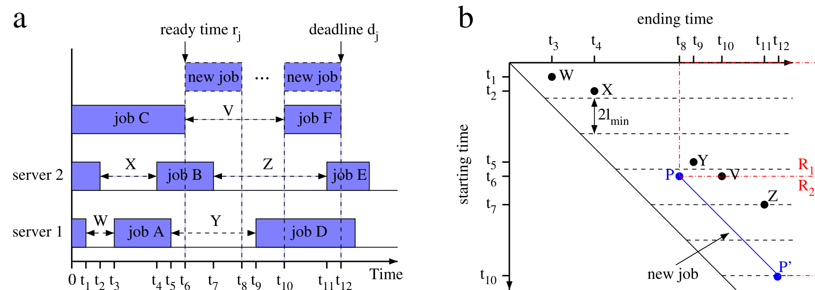

Fig. 2. (a) Jobs scheduled and idle periods in a 3-server system, (b) idle periods as points in the plane, plane partitioned into strips of width $2 \times l_{min}$, and feasible regions R_1, R_2 for the new job.

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, $h = 1, \ldots, H$, contains the idle periods falling within the h-th strip.

Rather than maintaining a single tree data structure as in [35], 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 Fig. 2(b), representing the earliest time the new job may start execution. 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. 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 Fig. 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 Fig. 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;

[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 $O((k + 1) \log n)$.

Author's personal copy

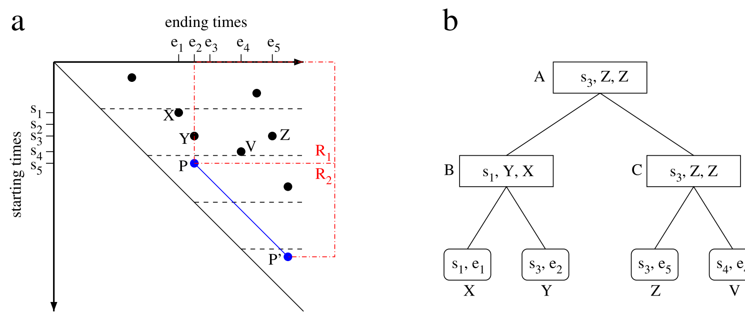

Fig. 3. (a) Schedule of advance reservations, (b) balanced tree structure storing the idle periods in the second strip from the top.

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.

4. 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.

4.1. Balanced tree structure for the Min-LIP strategy

Recall from Section 3.1 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 Fig. 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 $h, h = 1, \ldots, H$, 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 [9]. 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 Fig. 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
- a pointer to the idle period in v's subtree with the maximum length.

Fig. 3(b) shows the balanced tree T_h associated with the second strip from the top of the schedule shown in Fig. 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 Fig. 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 a 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 $h, h = 2, \ldots, H$, 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.

Author's personal copy

4.2. 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 Fig. 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 $T_p, T_{p+1}, \ldots, T_{p'}$.

The min-LIP algorithm exploits the observation that any feasible idle period in region R_2 is optimal in order to examine each tree T_h, $h = p, \ldots, p'$, 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 Fig. 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). Recall that we maintain a circular queue storing the tree indices. Therefore, using modulo-H arithmetic we are able to locate the appropriate tree for the trailing idle period in constant time. The insert operation takes $O(\log n)$ time. Since the number of strips that fall within region R_2 is at most $k = \left\lceil \frac{d_j}{2l_{\min}} \right\rceil$, 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 $T_{p-1}, T_{p-2}, \ldots, T_1$. 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 Fig. 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, $h = p - 1, \ldots, 1$, in region R_1 is searched using a standard algorithm for red–black trees [9] 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 [9,10]. The number of strips within region R_1 is at most $m = \left\lceil \frac{r_j}{2l_{\min}} \right\rceil$, 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 Fig. 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 Fig. 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.

4.3. 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, $h = 1, \ldots, H$. 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:

Author's personal copy

- 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 Fig. 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 $m = \left\lceil \frac{r_j}{2l_{\min}} \right\rceil$.

Step 2: *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 $k = \left\lceil \frac{d_j}{2l_{\min}} \right\rceil$ strips, the overall complexity for searching b_{R_2} in Step 2 is $O(k \log n)$.

4.4. Implementation of other scheduling strategies

The scheduling strategies we defined in Section 3.2 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.

5. 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 three 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.

We compare five scheduling strategies: first-fit, min-LIP, min-TIP, best-fit and LACT. The LACT algorithm, which we described in Section 2, does not consider the idle periods created at each server, and hence suffers the effects of capacity fragmentation. As mentioned earlier, 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.

Fig. 4(a)–(c) plot the loss rate, average delay, and utilization 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 Fig. 4(a) 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 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 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 in the creation of small trailing idle periods, which may fail to accommodate incoming jobs; hence increasing the loss rate in the system. The

Author's personal copy

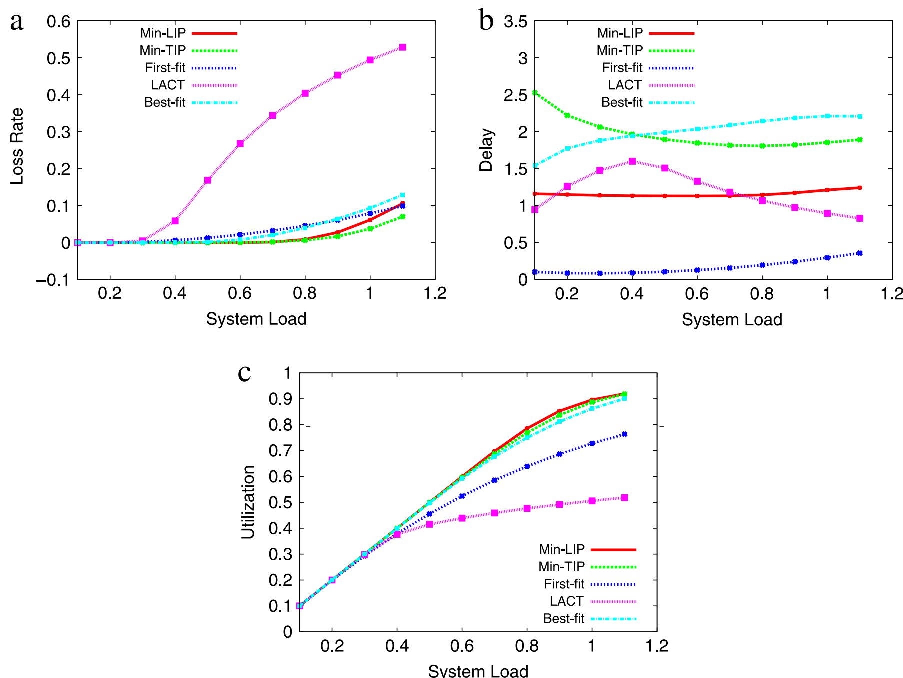

Fig. 4. (a)Loss rate vs. system load. (b) Average delay vs. system load. (c) Utilization vs. system load.

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.

Let us now turn to Fig. 4(b) 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 Fig. 4(a). 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 Fig. 4(a)). 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.

Fig. 4(c), 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 Fig. 4(a). Min-LIP, best-fit and min-TIP again have the best performance, followed by first-fit. Moreover, followed closely by the best-fit curve, the behaviour 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 tends to drop larger jobs with higher probability. This is due to the fact that it is more unlikely that the scheduler can find feasible idle periods for large jobs.

Overall, the average delay values in Fig. 5(a) 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.

In addition to providing insight into the relative behaviour of the five strategies due to the different optimization objectives considered, Fig. 4(a)–(c) 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.

Author's personal copy

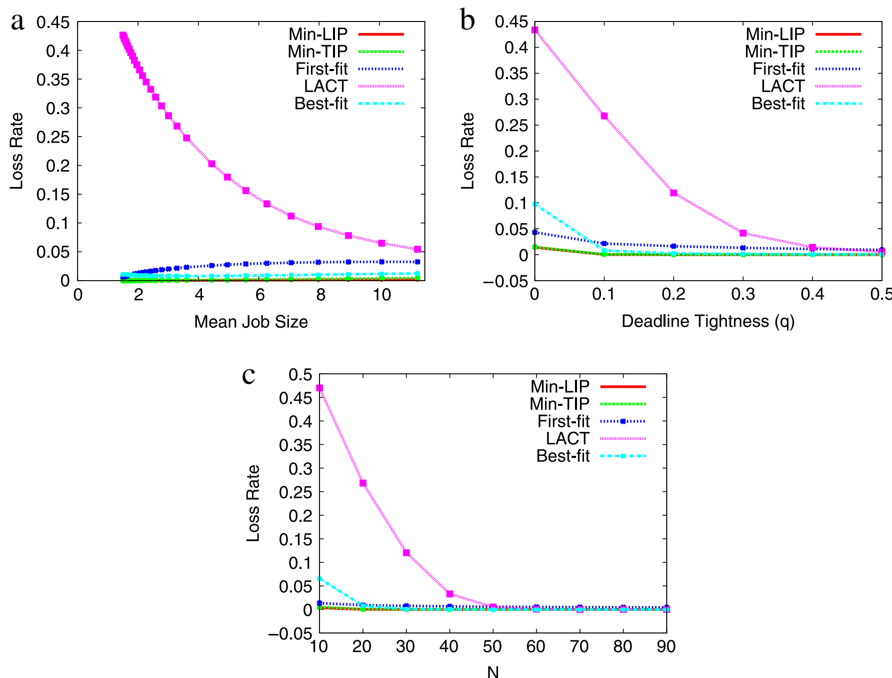

Fig. 5. (a)Loss rate vs. mean job. (b) Loss rate vs. deadline. (c) Loss rate vs. number of servers.

Specifically, the min-LIP and min-TIP algorithms cater to the user needs by ensuring that job deadlines are met 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 Fig. 5(a)–(c) 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.

Fig. 5(a) 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 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. First-fit has a higher loss rate, which increases with the mean job size. 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 Fig. 5(b) 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, Fig. 5(c) 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 worst 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.

6. 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 work provides a practical and efficient solution to the problem of scheduling resources in the emerging highly dynamic Grid environments.

References

[1] R.J. Al-Ali, K. Amin, G. Von Laszewski, F. Omer, D.W. Walker, M. Hategan, N. Zaluzec, Analysis and provision of QoS for distributed Grid applications, Journal of Grid Computing 2 (2) (2004) 163–182.

Author's personal copy

[2] Amazon EC cloud. http://www.amazon.com/gp/browse.html?node=3435361.
[3] B. Bode, D.M. Halstead, R. Kendall, Z. Lei, D. Jackson, The portable batch scheduler and the Maui scheduler on Linux clusters, in: Proceedings of the Usenix Conference, 2000.
[4] R. Buyya, D. Abramson, J. Giddy, Nimrod/g: an architecture for a resource management and scheduling system in a global computational Grid, in: Proceedings of HPC'00-Asia, May 2000, pp. 283–289.
[5] R. Buyya, D. Abramson, S. Venugopal, The Grid economy, Proceedings of the IEEE 93 (3) (2005) 698–714.
[6] E.-K. Byun, J.-W. Jang, W. Jung, J.-S. Kim, A dynamic Grid services deployment mechanism for on-demand resource provisioning, in: Proceedings of Cluster Computing and the Grid, 2005.
[7] E. Caronand, P.K. Chouhan, F. Desprez, Deadline scheduling with priority for client-server systems on the Grid, in: IEEE/ACM International Workshop on Grid Computing, 2004, pp. 410–414.
[8] H.-L. Chan, T.-W. Lam, K.-K. To, Nonmigratory online deadline scheduling on multiprocessors, SIAM Journal on Computing 34 (3) (2005) 669–682.
[9] T. Cormen, C. Leiserson, R. River, C. Stein, Introduction to algorithms, second ed., McGraw-Hill Book Company, 2001.
[10] M. de Berg, M. van Krefeld, M. Overmars, O. Schwarzkopf, Computational Geometry: Algorithms and Applications, second ed., Springer-Verlag, 2000.
[11] M.L. Dertouzos, A. Ka-Lau Mok, Multiprocessor on-line scheduling of hard real-time tasks, IEEE Transactions on Software Engineering 15 (12) (1989) 1497–1506.
[12] E. Elmroth, J. Tordsson, A Grid Resource Broker Supporting Advance Reservations and Benchmark-Based Resource Selection, in: Lecture Notes in Computer Science, vol. 3732, Springer-Verlag, 2005, pp. 1077–1085.
[13] I. Foster, C. Kesselman (Eds.), The Grid 2: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 2003.
[14] I. Foster, Y. Zhao, I. Raicu, S. Lu, Cloud computing and Grid computing 360-degree compared, in: IEEE Grid Computing Environments, GCE08, 2008, pp. 12–16.
[15] K.S. Hong, J.Y-T. Leung, Preemptive scheduling with release times and deadlines, Journal of Real-Time Systems 1 (1989) 265–281.
[16] K.S. Hong, J.Y-T. Leung, On-Line scheduling of real-time tasks, IEEE Transactions on Computers, 4 (10) 1326–1331.
[17] W. Horn, Some simple scheduling algorithms, Naval Research Logistic Quarterly 21 (1) (1974) 177–185.
[18] D. Isovic, G. Fohler, Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints, in: Proceedings of the IEEE 21st Real-Time Systems Symposium, November 23–27, 2000, Orlando, Florida.
[19] D. Isovie, Handling sporadic tasks in real-time systems: combined offline and online approach, http://www.mrtc.mdh.se/publications/0308.pdf.
[20] D. 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.
[21] D. Jackson, Q. Snell, M. Clement, Core algorithms of the Maui scheduler, Lecture Notes in Computer Science 2221 (2001) 87–102.
[22] W. Leinberger, V. Kumar, Information power Grid: the new frontier in parallel computing? IEEE Concurrency 7 (4) (1999) 75–84.
[23] M. Maheswaran, K. Krauter, R. Buyya, A taxonomy and survey of Grid resource management systems for distributed computing, Software: Practice and Experience 32 (2) (2002) 135–164.
[24] E. McCreight, Priority search trees, SIAM Journal of Computing 14 (2) (1985) 257–276.
[25] R. Min, M. Maheswaran, Scheduling advance reservations with priorities in Grid computing systems, in: Proceedings of PDCS'01, 2001, pp. 172–176.
[26] Platform Computing Corporation. http://www.platform.com.
[27] K. Ramamritham, J. Stankovic, Scheduling algorithms and operating systems support for real-time systems, IEEE Transactions on Software Engineering 82 (1) (1994) 56–67.
[28] S. Sahni, Preemptive scheduling with due dates, Operations Research 27 (5) (1979) 925–934.
[29] S. Sahni, Y. Cho, Nearly online scheduling of a uniform system with release times, SIAM Journal on Computing 8 (2) (1979) 275–285.
[30] M. Siddiqui, A. Villazon, T. Fahringe, Grid capacity planning with negotiation-based advance reservation for optimized QoS, in: Proceedings of Conference in Supercomputing, vol. 2006, 2006, pp. 103–118.
[31] W. Smith, I. Foster, V. Taylor, Scheduling with advanced reservations, in: Proceedings of IPDPS'00, 2000, pp. 127–132.
[32] J.A. Stankovic, M. Spuri, K. Ramamritham, G.C. Buttazzo, Deadline Scheduling for Real-Time Systems. EDF and Related Algorithms, in: Series: The Springer International Series in Engineering and Computer Science, vol. 460, 1998, Hardcover.
[33] A. Sulistio, R. Buyya, A Grid simulation infrastructure supporting advance reservation, in: Proceedings of PDCS'04, November 2004, pp. 1–7.
[34] A. Takefusa, H. Casanova, S. Matsuoka, F. Berman, A study of deadline scheduling for client-server systems on the computational Grid, in: Proceedings of HPDC, 2001, pp. 406–415.
[35] J. Xu, C. Qiao, J. Li, G. Xu, Efficient burst scheduling algorithms in optical burst-switched networks using geometric techniques, IEEE Journal on Selected Areas in Communications 22 (9) (2004) 1796–1811.
[36] W. Zhao, K. Ramamritham, J.A. Stankovic, Preemptive scheduling under time and resource constraints, IEEE Transactions on Computers C-36 (8) (1987) 949–960.
[37] W. Zhao, K. Ramamritham, J.A. Stankovic, Scheduling tasks with resource requirements in hard real-time systems, IEEE Transactions on Software Engineering SE-13 (5) (1997).

C. Castillo is a Research Staff Member in IBM Research since 2008.

Her research interest revolves around resource management in large scale distributed systems. She is particularly interested in middleware technologies for data intensive applications.

Previously, she was a research assistant in the Department of Computer Science at North Carolina State University where she obtained her Ph.D. Prior to that, she obtained her Master Degree from the Department of Electrical Engineering in the same institution. She received her undergraduate degree in Electrical Engineering from the University of Panama, Panama, where she also worked as a lecturer.

On the professional side, she has worked as a consultant contractor for Centauri Technologies Corporation in Panama and interned at Intel Research, IBM Research (T.J. Watson) and Cisco Systems Inc. in the United States. She is a member of IEEE and ACM.

G.N. Rouskas is a Professor of Computer Science at North Carolina State University.

He received his Diploma in Computer Engineering from the National Technical University of Athens (NTUA), Athens, Greece, in 1989, and his M.S. and Ph.D. degrees in Computer Science from the College of Computing, Georgia Institute of Technology, Atlanta, GA, in 1991 and 1994, respectively.

His research interests include network architectures and protocols, optical networks, network design, and performance evaluation. He is co-editor of the book "Next-Generation Internet Architectures and Protocols" (Cambridge University Press, 2011), author of the book "Internet Tiered Services" (Springer, 2009), and co-editor of the book "Traffic Grooming for Optical Networks" (Springer 2008).

He is founding co-editor-in-chief of the Optical Switching and Networking Journal, he is on the editorial board of the IEEE/OSA Journal of Optical Communications and Networking, and he has served on the editorial boards of IEEE/ACM Transactions on Networking, Computer Networks, and Optical Networks.

He is the TPC co-chair for ICCCN 2011, and he has served as TPC or general chair for numerous conferences, including the IEEE GLOBEC 2010 ONS Symposium, BROADNETS 2007, IEEE LANMAN 2004 and 2005, and IFIP NETWORKING 2004.

He is the recipient of several research and teaching awards, including a 1997 NSF CAREER Award. He is a Distinguished Lecturer for the IEEE Communications Society in 2010–2011.

K. Harfoush obtained his Ph.D. in Computer Science from Boston University in 2002. He is currently Associate Professor in the Department of Computer Science at North Carolina State University, which he joined in 2002. His research interests are in the general areas of network modeling and design, Internet Measurement, Peer-to-Peer systems and network security. He is leading the Network Design and Traffic Engineering group at North Carolina State University.

Prof. Harfoush is a recipient of the prestigious NSF CAREER award, and serves on the program committees for numerous conferences including IEEE INFOCOM and IEEE ICNP. He is a member of ACM and IEEE since 2002.