Event-based Optimization of Schedules for Grid Jobs

Dalibor Klusáček

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy at
the Faculty of Informatics, Masaryk University

Brno, Czech Republic, 2011

Except where otherwise indicated, this thesis is my own original work.

Dalibor Klusáček
Brno, May 2011

Acknowledgments

I would like to express my deep thanks to Luděk Matyska and Hana Rudová for supporting me in my work and motivating me. It was my pleasure to work with them, as they continuously contributed with their expertise to my research and to this thesis.

I would also like to thank to all fellows from the Laboratory of Advances Networking Technologies at the Faculty of Informatics, Masaryk University, for their help. In no particular order I would like to thank especially to Igor Peterlík, David Antoš, Lukáš Hejtmánek, Jiří Filipovič, Miloš Liška and Matúš Madzin. Thanks guys, I really appreciate having the opportunity meeting and working with you for all these years.

I am deeply grateful also to my parents who have never given up to support me during my whole study. Last but not least, I would like to express my special thanks to my wife Vladimíra. She has always supported me and also came up with several great ideas that pushed my research forward.

Dalibor Klusáček

This work has also been kindly supported by the Ministry of Education, Youth and Sports of the Czech Republic under the research intent No. 0021622419, and by the Grant Agency of the Czech Republic with grant No. 201/07/0205. We also appreciate the support of EU CoreGRID NoE (FP6-004265), and the help of MetaCentrum, graciously supported by the research intent MSM6383917201. MetaCentrum team provided us the workload data and help us to interpret them correctly. Moreover, the access to the MetaCentrum computing facilities provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" LM2010005 funded by the Ministry of Education, Youth, and Sports of the Czech Republic is highly appreciated. The workload logs from the SDSC SP2 and SDSC DataStar were graciously provided by Victor Hazlewood. The HPC2N workload log was graciously provided by Ake Sandgren, while the LLNL Thunder workload log was graciously provided by Moe Jette. Synthetic data sets including job deadlines were graciously provided by Ranieri Baraglia and Gabriele Capannini.

Abstract

This thesis focus on the well known and complex problem of efficient job scheduling in large dynamic computational environments such as Grids. Current production scheduling systems are using queue-based algorithms that work well when few simple objectives are followed. However, it is very complicated to satisfy complex objectives involving, e.g., slowdown, response time, deadlines, resource utilization or fairness through a queue-based solution, since no, or quite limited planning ahead is often used. Therefore, the scheduling decisions for each job are taken in an "ad hoc" fashion, with limited consideration of the remaining jobs in the queue, or with limited consideration of previous and future steps. Since the scheduling is performed only with respect to the current state and a long term impact is not considered, the overall performance may easily degrade.

The main challenge of this thesis is to propose realistic approach that would allow to avoid such situations. For this purpose, we propose the application of schedule-based solution. Schedule represents a job to machine(s) mapping in time. Using schedule, the job execution is planned ahead, thus becomes predictable. Moreover, prepared schedule can be easily evaluated, therefore the quality of the solution can be exactly measured using selected optimization criteria. Furthermore, the schedule can be optimized, increasing the quality of the solution. The success of this approach is based on several principles that allow to build the schedule efficiently. The schedule is built using fast and simple scheduling policy. Then, the quality of such initial schedule is optimized using local search-based optimization metaheuristics. Both the policy and the optimization algorithm use so called incremental approach, which minimizes computational requirements of these techniques. Instead of time-expensive recomputation, the schedule is kept up to date subject to dynamically changing system through incrementally performed local modifications. When possible, existing gaps in the schedule are used to place new jobs. This so called gap filling approach follows the incremental approach while increasing the overall resource utilization. Also, the optimization plays an important role when fixing the schedule after some unexpected event, dealing with, e.g., machine failures or inaccurate job processing time estimates. Moreover, so called anytime approach is used to guarantee flexible runtime of optimization algorithms. When necessary, the potentially time-consuming optimization routines can be stopped at any time, guaranteeing minimal overhead. Finally, so called event-based approach is applied, meaning that the whole scheduling process is driven by incoming events. These events represent dynamically changing state of the system. Based on the event type, proper actions are proposed to keep the schedule up to date and consistent with the new situation. Different events are used to denote job arrivals and job completions or machine failures and restarts, etc. Together, the whole approach is designed to remain applicable in a real and dynamic environment. It is able to deal with dynamic job arrivals, machine failures and restarts, it can handle specific job requirements and it is tolerable to inaccurate processing time estimates, that complicate construction of efficient schedules.

With the help of newly developed Alea simulator, the proposed approach has been exhaustively evaluated against selected queue-based scheduling algorithms using several data sets and objective functions. The evaluation has clearly demonstrated that the proposed solution significantly outperforms remaining algorithms in most cases. The proposed policies and optimization algorithms proved to be very flexible, successfully solving different optimization problems, involving multiple optimization criteria. Also, the solution was shown to be tolerable to the inaccurate processing time estimates. In case that the users were able to improve their job processing time estimates, our techniques provide better opportunities to increase the quality of the solution with respect to the queue-based algorithms.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

The concept of *Grid* was firstly introduced by Foster and Kesselman [59] who defined it as a distributed and decentralized computer environment composed of large number of heterogeneous resources which are managed by different owners like companies, universities or another business or scientific organizations. Such resources may be of different type such as computational machines, data storage nodes, databases, scientific tools, etc. These resources are interconnected by (high speed) communication network. Most of these resources are built from relatively cheap components like regular PCs. Also, other common hardware or existing infrastructure, e.g., the Internet, is usually used. As a matter of fact, the Grid environment is *changing dynamically in time*, mainly due to the changing load of the system, available network bandwidth and also due to other dynamic events such as resource or network failures and recoveries [59, 60].

The main goal of the *Grid technology* is to manage this large and heterogeneous environment while allowing an easy access to its resources for various users. It allows them to submit their jobs into the system, guaranteeing them nontrivial Quality of Service (QoS) while hiding the complexity of the system itself by providing powerful but simple interfaces for the end user of the Grid [59]. Moreover not only users but also resource owners should be satisfied, i.e., the Grid technology should be safe and efficient, thus allowing proper utilization of its resources. Clearly, if both users' and resource owners' goals are to be satisfied, multi-objective criteria have to be used. To meet these goals *automated and sophisticated scheduling techniques* should be applied. On the other hand, scheduling in such a highly dynamic, distributed, heterogeneous and decentralized environment is an extremely difficult task if good performance, nontrivial QoS, scalability, etc., are required.

Current scheduling techniques applied in Grids are mostly based on the queueing systems of various types which are designed with respect to specific needs of Grid technology. Since the Grids are built over existing infrastructure their physical as well as logical topology is often hierarchical [175]. On the lower level *Local Resource Management Systems (LRMS)* managing and scheduling jobs onto local resources of the institution are usually applied. On the higher level *meta schedulers* or so called *brokers* can be used. They usually manage different local sites, that each uses (often different) LRMS. Similarly to LRMS, they are using queue-based scheduling policies.

While simple objectives can often be reasonably satisfied with some queue-based scheduling policy, satisfying complex objectives involving slowdown, response time, deadlines, resource utilization, fairness, etc., is a hard task when the queue-based solution is used. The problem is that queue-based solutions often use no or quite limited planning ahead, thus the scheduling decision for each job is taken in an "ad hoc" fashion, with limited consideration of the remaining jobs in the queue. Therefore, the scope of scheduling is often bounded to the current state only and a long term impact of such scheduling decision can not be directly evaluated. This can degrade the quality and efficiency of such a solution. This problem is even more apparent when the common user of the system is not aware of or does not understand the applied scheduling policy. Nowadays, common users are sometimes forced to cheat when looking for good performance of their applications. For example, if the user requires fast or immediate response, he or she can try to execute the job by logging directly on the specific machine and starting the job from the command line. In such situation, the LRMS is bypassed which can discriminate honest users and — in general — leads to inefficient performance of the whole system. Therefore, such behavior is often penalized by the system administrator. Moreover, it is important to notice that in a long term scope such situation can lead to both user's and administrator's frustration.

1.1 Thesis Goals

The apparent way out is based on predictability [131, 52]. It means that the user obtains information when his or hers job will be executed. Based on such information the users can now plan ahead their work since the execution periods of their jobs are guaranteed. To allow such predictability, the execution must be planned ahead using a plan of execution called the *schedule*. Unlike queue-based solutions the schedule allows to maintain information about job to machine(s) mapping in time. Thanks to the prepared schedule, information concerning job execution is available *in advance*, before the actual job execution which is often favorable. Once the schedule is created, it can be *evaluated* with respect to the applied optimization criteria and further *optimized*, increasing the quality of the initial solution.

In this thesis, we introduce a model based on the use of a schedule when solving dynamic Grid scheduling problem. It combines fast and simple scheduling policies with advanced scheduling techniques [138, 68]. Scheduling policies construct the initial schedule while *local search-based* [68] optimization techniques improve the quality of such initial solution with respect to the applied objective criteria. When dealing with multiple objectives, schedule-based solution is very promising since it maintains a lot of information which can be used by scheduling algorithms. For example, for each job, specific machine(s) is selected in advance as well as the time interval when the job will be executed. Such schedule can be modified by optimization algorithm to better satisfy different criteria. The effect of these modifications can be simply evaluated by computing the values of applied objective functions. Then, the best solution can be chosen from more candidate solutions according to the values of objective functions.

The proposed model must cope with common real life situations such as dynamic job arrivals, machine failures and restarts. Clearly, such events must be properly handled and the schedule must be adjusted accordingly. Since the problem is not off-line [153], it is important to keep the runtime of scheduling algorithm in a decent level, preventing job starvation or delays in job execution. Therefore, the proposed scheduling techniques must use a time-efficient approach.

Moreover, construction of schedule requires additional information from the users. Clearly, some reasonable upper bounds of jobs' processing times must be specified, otherwise the schedule cannot be constructed. It can be expected that such estimates will be rather inaccurate [113, 32, 34]. Therefore, the proposed model must be able to deal with such situation and perform proper schedule corrections when the inaccuracy is detected.

Finally, the proposed model must be experimentally evaluated to demonstrate whether all important characteristics of the studied problem are properly maintained. The performance of the proposed scheduling algorithms must be experimentally evaluated with respect to the common queue-based algorithms, that represent current state-of-the-art approaches.

1.2 Thesis Contribution

The main contribution of this work is the proposal of *advanced scheduling algorithms* and the proposal of *realistic application* of these scheduling algorithms in dynamic and heterogeneous environments like Grids.

Firstly, the studied *Grid scheduling problem* is formally described, including description of several objective criteria. For this purpose, unified notation has been proposed and used throughout the whole thesis, allowing uniform description of both existing and proposed scheduling algorithms.

Also, the related work including uniform descriptions of existing state-of-the-art scheduling techniques has been recapitulated. The highest attention is dedicated to those techniques that are lately used for comparison when evaluating the performance of the proposed solution.

The main part of the thesis is dedicated to the presentation of the proposed solution. It is based on the construction of the *schedule* that allows to plan jobs' execution in advance. Moreover, as the schedule is built it can be *evaluated* using selected criteria and further optimized to increase the quality of the generated solution. More precisely, this thesis proposes a realistic *model* that incorporates all major features that represent severe complications when applying schedule-based solutions. The model incorporates *dynamic features* such as job arrivals, inaccurate processing time estimates or machine failures and restarts that all complicate the design of an efficient scheduling algorithm.

The major part of the proposed solution is related to the construction and maintenance of the schedule. It uses two fundamental techniques. Initial schedule is constructed using a fast *scheduling policy.* Such initial schedule is further optimized with a *local search-based optimization routine.* Applied scheduling techniques as well as the model itself exploit *several approaches* to establish realistic and efficient solution.

The *event-based approach* is used as a general paradigm, detecting dynamically arriving events that signal changes in the state of the system. For each such *event* an appropriate reaction is proposed and discussed. Here the goal is to keep the schedule up to date subject to such dynamic changes (events).

The scheduling policy and the optimization routine are subject to incremental, gap filling and anytime approach. *Incremental approach* guarantees that existing schedule is reused in the newly constructed schedule as much as possible which decreases the computational requirements and supports scalability. *Gap filling approach* aims to utilize existing "gaps" in the schedule with suitable jobs. It follows the incremental approach since the amount of modifications related to gap filling is very small. Finally, the *anytime approach* allows us to immediately stop optimization procedure when a higher priority request is delivered to the scheduler.

The model also proposes how the scheduling policy and the optimization routine can be used to *fix the schedule* upon dynamic changes in the resource pool such as machine failures or restarts. Similarly, if the jobs' processing time estimates are inaccurate *early jobs' completions* start to appear. Since the schedule is built using estimates, each such early job completion causes inconsistencies in the current schedule, creating gap(s) in the schedule. In such case, the model proposes a repair procedure based on the schedule optimization combined with the gap filling approach.

Beside the general model, this thesis also presents implementation details of the proposed solution, i.e., descriptions of the applied data structures, algorithms and additional routines that together implement the proposed solution. Also, the computational complexity of all algorithms and routines is closely analyzed.

The newly proposed solution is comprehensively evaluated through a set of experiments that demonstrate the suitability and performance of the proposed solution. To allow fair comparison a complex publicly available job scheduling simulator called *Alea* has been also developed as a part of this thesis. It incorporates all important features of the studied problem while implementing both the proposed schedule-based and several queue-based scheduling algorithms.

Last but not least, thanks to the helpful staff of the Czech national Grid infrastructure *MetaCentrum*, we were allowed to collect and prepare complex real life-based data sets that we have lately used during our experiments together with the commonly applied data sets from public workloads archives. These data sets are also publicly available to the scientific community[1].

1.3 Thesis Structure

The rest of this thesis is organized as follows: In Chapter 2 we formally define the studied Grid scheduling problem, introducing unified notation which is used in the whole thesis. Using this notation, both existing and proposed scheduling algorithms can be described in an uniform fashion. The Grid scheduling problem is characterized by job and resource parameters and the dynamic behavior of the system that together determine several natural constraints which must be satisfied by any (feasible) solution of such a problem. We

[1] `http://www.fi.muni.cz/~xklusac/workload/`

also present several objective criteria that are commonly used to measure the quality of the generated solution. These criteria play an important role, since the general request is to produce "good" and "efficient" solutions. The chapter also discusses the complexity of the Grid scheduling problem and presents some general requirements concerning actual Grid scheduler.

In Chapter 3 we discuss the related work in the area of Grid scheduling. We provide basic overview of general scheduling techniques while focusing on the heuristic and meta-heuristic techniques. Next, the widely used queue-based scheduling approach is presented, followed by a uniform description of several popular queue-based scheduling policies. We also present the existing work on schedule-based systems. The problem of dynamically changing environment is discussed both for queue and schedule-based systems. A special attention is payed to the problem of job processing time estimates. The chapter concludes with an overview of existing techniques used for job processing time prediction, that can be used to refine otherwise highly inaccurate processing time estimates.

In Chapter 4 we discuss the proposed schedule-based approach in detail. We describe the applied event-based approach that manages the whole scheduling process. Beside that, incremental, gap filling and anytime approaches are proposed, allowing to build fast, efficient and scalable solution techniques. Next, several scheduling policies and local search-based optimization procedures are proposed to construct and optimize the schedule respectively. All important aspects of the solution such as applied data structures, pseudo-codes of algorithms or the expected computational complexity are discussed.

Chapter 5 provides detailed description and performance demonstration of the proposed job scheduling simulator *Alea*. This simulator is used to experimentally evaluate the proposed solution. The evaluation is presented in Chapter 6. We present several use cases which demonstrate the performance of the proposed solution under various setups that simulate different real life scenarios. The results include comparison with several queue-based algorithms that have been described in Chapter 3. The discussion on the flexibility and applicability of the proposed solution summarizes this chapter.

Finally, the work is concluded and the future work is presented in Chapter 7. This last chapter is followed by the list of bibliography referenced through the thesis, appendix, and a list of author's publications.

Chapter 2

Problem Description

In this chapter we define the problem of job scheduling in Grids or similar heterogeneous environments. We discuss all major features such as the definition of the Grid, the job, the resource, the objective function, or the system dynamics. We start with a Grid definition followed by the description of typical Grid scheduling approaches. Then we present a formal description of the studied problem, extending the existing standard notations.

2.1 Characteristics of Grid Scheduling

Grid is a distributed, decentralized, heterogeneous and a highly dynamic computer environment consisting of various resources which are interconnected by computer network [60]. By definition, it is a dynamically reconfigurable, scalable and autonomous infrastructure that provides location independent, pervasive, reliable, secure and efficient access to a coordinated set of services, such as computing power, storages, instruments, data, etc. [111]. Grid technology allows its users to submit their jobs to the Grid system where these jobs are executed, their results are collected and delivered back to the users. Nontrivial Quality of Service (QoS) such as fairness, fault tolerance, security etc., is guaranteed [60].

We now define some basic terminology concerning the Grid scheduling approaches. First, we have to distinguish between centralized and decentralized scheduling. By *centralized scheduling* we usually understand scheduling approach based on a single Local Resource Management System (LRMS) such as PBS Pro [88], LSF [191] or Sun Grid Engine [66]. In this case, a set of computers or clusters — usually belonging to one team or organization — is managed by a single scheduler which has a full control of all jobs and resources (see the *Centralized layer* in the bottom of Figure 2.1[1]). It should be mentioned that such solution does not satisfy completely the definition of the Grid since it is not working in a decentralized fashion.

Decentralized scheduling usually applies some hierarchical model as depicted by Figure 2.1. Here the scheduling process typically consists of separate decisions on different levels although some cooperation for effective solution is often required. Decentralized scheduling is often managed by one or more so called *Metaschedulers* or *Brokers* like

[1]Figure 2.1 is based on the scheme proposed in [175].

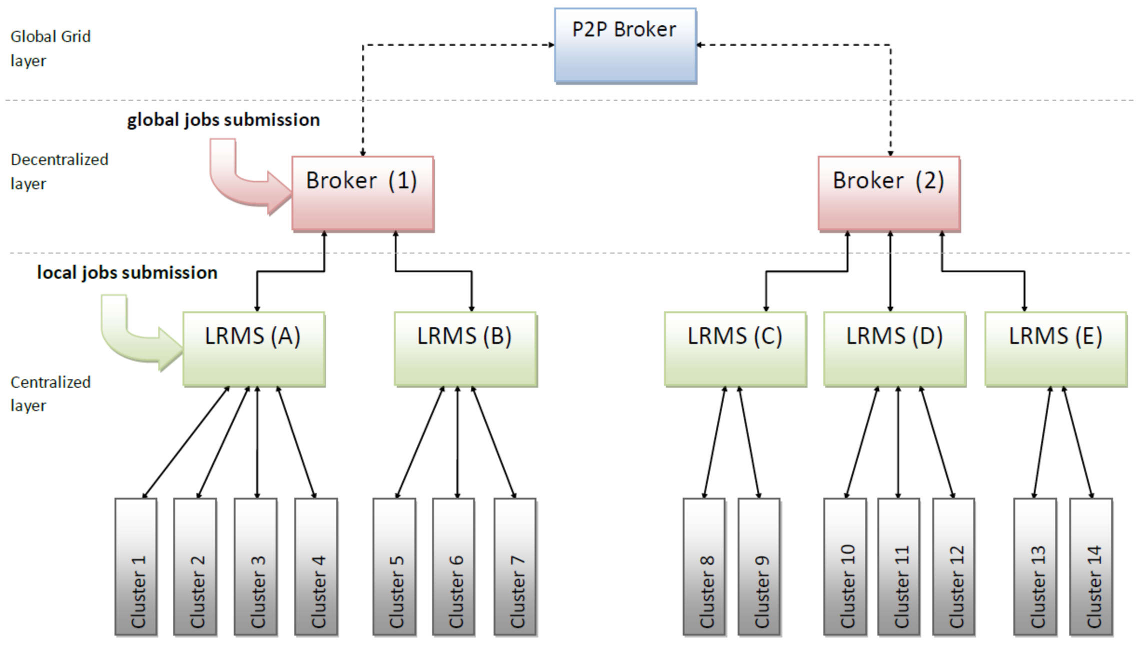


Figure 2.1: Hierarchy of Grid scheduling systems.

GridWay [82], Community Scheduler Framework [182], Moab [7] or Maui [6]. These brokers often manage different underlying centralized LRMSs [175]. Moreover, several brokers may communicate together, establishing a *global Grid* consisting of several local Grids or resources [175]. In this case, one or more so called P2P Brokers can be applied to interconnect different Grids, allowing truly global Grid infrastructure. In such an environment it is often possible to submit jobs at different levels — local users prefer their local LRMS while brokers are used by the users from the outside [175].

In this thesis, we consider centralized scheduling approach only. From now on we assume that the scheduler has full control of all jobs and resources. From this point of view, we will define and study problems similar to those that production LRMSs have to deal with.

We can also distinguish between *static* and *dynamic scheduling*. In the static case, we assume that all information about jobs, resources, Grid topology and other constraints are known before the start of the scheduling process and do not change in time. Such situation is not realistic in the Grid environment which is highly dynamic. Jobs and resources are appearing and disappearing in time, errors often occur and uncertain and imprecise information about jobs, resources or topology is available. If the scheduling process has to deal with such a dynamic system we are talking about *dynamic scheduling*. Some authors refer the same problem as an *online scheduling* [153]. Although the dynamic scheduling works in an online fashion we rather do not use the term online scheduling, since it is often used in relation to the *hard real-time systems* such as CPU schedulers [14, 77]. We will closely discuss these issues in Section 2.5.

Finally, we have to distinguish between local and global scheduling. *Local scheduling* is used to assign the process(es) of the job to the CPU(s) on a single machine [29], while *global scheduling* decides where to execute given job. Here, the selection of a suitable machine(s) is the key issue. Global scheduling is not responsible for the final mapping of the job's processes onto CPUs, this is done by the machine's local scheduler, i.e., operating system. Following these definitions, Grid scheduling systems perform global scheduling. Although global scheduling is not responsible for the final mapping of the job's processes onto CPUs, it decides when and on what machine(s) the job will be sent. Therefore, it can influence the behavior of the local machine's scheduler. Detailed discussion of these processes and their parameters will follow in Sections 2.2, 2.3 and 2.4.

As far as we know, there is no standard notation available for defining complex Grid scheduling problems. It has been shown in [56] that the widely used $\alpha|\beta|\gamma$ notation of Graham [71] is not sufficient. Still, our notation relies on the standard notations [29, 18] as much as possible.

2.2 Jobs

Job represents the user's application, which is executed on the resource(s). We consider u job owners (users) in the system that together submit n *jobs*. Let $\mathcal{J} = \{j_1, ..., j_n\}$ is the set of all jobs. The *owner* of a given job $j \in \mathcal{J}$ is denoted as o_j, where $o_j \in \{owner_1, ..., owner_u\}$. Each job j has known normalized processing time p_j (runtime). The resulting *processing time* $p_{i,j}$ depends on the speed of the machine i and it is equal to p_j/v_i, where v_i is the speed of the machine i. While the $p_{i,j}$ defines the actual runtime, users (job owners) are often requested to estimate the job runtime in advance. Such an estimate is denoted as *estimated processing time* ep_j.

Job requires one or more CPUs per period (called per period usage $usage_j$ of job j [19]). Job requiring one CPU is called *sequential* while job requiring more than one CPU is called *parallel*. Parallel jobs are sometimes divided into two subgroups: rigid and moldable [74]. A *moldable job* is defined as a parallel job that can be run on a variable number of CPUs, i.e., for a given job j the $usage_j$ is not known until a final scheduling decision is taken [74]. On the other hand, *rigid job* has fixed $usage_j$ value which is known in advance and cannot be changed by the scheduler [74].

Beside the parallel and sequential jobs another major category is a *workflow*. Workflow is a job that consists of several tasks which may have defined dependencies (precedence constraints) between each other. Those dependencies may result in a complicated structures although one of the most common description is based on the Directed Acyclic Graph (DAG) notation [85, 193]. In such situation, each task in the workflow can start its execution only when all preceding tasks in DAG are already finished. However, this notation only covers simpler workflows. If the workflow consists of, e.g., two tasks that interact together over the time, then the DAG notation is not sufficient and more powerful "non-DAG" [193] models must be used. In this thesis, all parallel jobs are considered as rigid and workflows are not used at all. Also, we consider neither preemptions of the jobs nor migrations from one machine to another.

Job arrives at time r_j which is the *release date* of the job (sometimes called arrival time) and the earliest time when the job j can start its processing. S_j denotes the actual start time of the job while C_j is the actual *completion time* of the job j.

Since no preemptions are allowed, Formula 2.1 holds[2] for every job. Let $\mathcal{R}_j$ specifies the set of machines assigned to the job j and $usage_{i,j}$ refers to the number of CPUs assigned to the job j on machine i. As a consequence the Formula 2.2 expresses the request that every job must obtain the requested amount of CPUs.

$$C_j = S_j + \min(p_{i,j}, ep_j) \qquad \forall j \in \mathcal{J} \tag{2.1}$$

$$\sum_{i \in \mathcal{R}_j} usage_{i,j} = usage_j \qquad \forall j \in \mathcal{J} \tag{2.2}$$

Let $\mathcal{CPU}_j$ is the set of CPUs that the job j uses during its execution. Since no job migrations are allowed, $\mathcal{CPU}_j$ remains the same during the whole job execution. Moreover, two different jobs must use different CPUs if their executions overlap in time, i.e., their sets of assigned CPUs must be disjunctive. This request is guaranteed by Formula 2.3.

$$\mathcal{CPU}_{j_1} \cap \mathcal{CPU}_{j_2} = \emptyset \qquad \forall j_1, j_2 \in \mathcal{J} \text{ s.t. } j_1 \neq j_2 \wedge ((S_{j_1} \leq C_{j_2} \wedge S_{j_1} \geq S_{j_2}) \vee (S_{j_2} \leq C_{j_1} \wedge S_{j_2} \geq S_{j_1})) \tag{2.3}$$

The *due date* d_j (or deadline) may be used to represent the expected completion time of the job. Moreover, each job j may be assigned with a *weight* w_j defining its "importance" or "priority" among other jobs.

2.3 Resources

In this section we define the computational resources. We were inspired by the resource-constrained project scheduling problem [19] and resource-constrained machine scheduling [16] when defining resources and their capacity.

There are m *machines* characterized by the speed and the number of CPUs. More precisely, machine i ($1 \leq i \leq m$) has cap_i CPUs which is also called *capacity of the machine*[3]. Each machine i has the *speed* v_i. Machines having the same speed v_i and the same capacity cap_i can be grouped into a *cluster* k. Grouping of the machines into clusters is common for machines located within the same local area network. There are l clusters and $\mathcal{M}_k$ defines the set of machines in cluster k. Clearly, $\mathcal{M}_{k1} \cap \mathcal{M}_{k2} = \emptyset$ if $k1 \neq k2$. Therefore, Formula 2.4 defines the set of all machines in the system and Formula 2.5 defines the total number of all CPUs in the system (*cpus*).

[2]In real systems, job j is killed when the time limit ep_j is reached thus the actual runtime never exceeds this value as shown in Formula 2.1.

[3]Machine is a renewable resource [19] meaning that job consumes (portion of) its capacity but as soon as the job finishes, the capacity is *renewed*.

$$\mathcal{M} = \bigcup_{k=1}^{l} \mathcal{M}_k \tag{2.4}$$

$$cpus = \sum_{i \in \mathcal{M}} cap_i \tag{2.5}$$

All machines within a cluster k use the Space Sharing processor allocation policy[4]. Let $cpus_k$ is the total number of CPUs in cluster k (see Formula 2.6). Then the Space Sharing policy allows parallel execution of $j_1, \dots j_s$ jobs at the cluster k if Formula 2.7 holds.

$$cpus_k = \sum_{i \in \mathcal{M}_k} cap_i \tag{2.6}$$

$$cpus_k \geq \sum_{j \in \{j_1, \dots j_s\}} usage_j \tag{2.7}$$

Therefore, for a given parallel job j several machines within the same cluster can be co-allocated to process the job. Job j can only be executed on cluster k if Formula 2.8 is true. Clearly, Formula 2.8 is a special case of Formula 2.7,

$$cpus_k \geq usage_j \tag{2.8}$$

On the other hand, machines belonging to different clusters cannot be co-allocated (used for scheduling of the same parallel job). This implies that all machines where the job is processed belong to the same cluster, thus all have the same speed. An assignment of jobs to machines with different speed is not supported[5]. Also processing time of the job $p_{i,j}$ is the same for all machines i where the job j is executed.

2.4 Specific Job Requirements

So far we have defined typical parameters that are used to describe jobs and machines. Here the given job can be executed on any cluster satisfying Formula 2.8. However, this is a very basic constraint which is often not sufficient in the real life. In heterogeneous environments, users often specify some subset of machines or clusters that can process their jobs. This subset is usually defined either by the resource owners' policy (user is allowed to use such cluster), or by the user who requests some properties (library, software license, execution time limit, etc.) offered by some clusters or machines only. Also, the combination of both owners' and users' restrictions is possible [103].

[4]Some authors prefer the name Space Slicing policy [54] to emphasize that preemption is not used. To remain consistent with later chapters, we use the name Space Sharing in this thesis.

[5]This is a feature of our current model. In reality, it may be possible to place machines with different speeds into one cluster or even co-allocate several clusters to execute one parallel job.

First of all, each cluster may have additional parameters that closely specify its properties. These parameters typically describe the architecture of the underlying machines (Opteron, Xeon, ...), the available software licenses (Matlab, Gaussian, ...), the operating system (Debian, SUSE, ...), the list of queues allowed to use this cluster (each queue has a maximum time limit for the job execution, e.g., 2 hours, 24 hours, 1 month), the network interface parameters (10Gb/s, Infiniband, ...), the available file systems (nfs, afs, ...) or the cluster owner (Masaryk University, Charles University, ...). For simplicity, we expect that all the machines within a cluster k supports the same properties. These properties are denoted as a set of supported properties $\mathcal{SUPP}_k = \{prop_{k_1}, .., prop_{k_q}\}$, meaning that cluster k supports q different properties. Different clusters may support different properties.

Corresponding information is often used by the user to closely specify job's characteristics and requirements. Those are typically the time limit for the execution, which may be given explicitly by the user (ep_j) or the ep_j value is derived from the known queue time limit which is set by the system administrator. Users may also further specify which cluster(s) is suitable for their jobs by specifying the required machine architecture, the requested software licenses, the operating system, the network type or the file system, etc. In another words, by setting these requirements, user may prevent the job from running on some cluster(s). In real life, there are several reasons to do so. Some users strongly demand security and full control and they do not allow their jobs (and data) to use "suspicious" clusters which are not managed by their own organization. Others need special software such as Matlab or Gaussian which is not installed everywhere. Some clusters are dedicated for short jobs only and a user wanting more time is not allowed to use such cluster, and so on.

All these requests are often combined together and have to be included into the decision making process to satisfy all specific job requirements. Formally, each job j may specify a set of requested properties $\mathcal{REQ}_j = \{prop_{j_1}, .., prop_{j_r}\}$, meaning that j requests r different properties. From this point of view, cluster k is suitable to execute the job j only if Formula 2.9 is satisfied.

$$\mathcal{REQ}_j \subseteq \mathcal{SUPP}_k \tag{2.9}$$

If no suitable cluster is found, the job has to be canceled.

Clearly, the specific job requirements cannot be used when the corresponding cluster parameters are not known. Without them, consideration of "job-to-machine" suitability is irrelevant. Therefore, whenever the term *specific job requirements* is referenced in this thesis, it means that both additional job and cluster parameters are applied, possibly decreasing the number of suitable clusters for the job execution.

2.5 System Dynamics

In this section we describe an important feature that seriously affects the design and application of scheduling algorithms. It is the *system dynamics*. It means that the state of the system is not static but it is changing dynamically in time. Hence, also the studied problem is dynamic since it changes over the time. Therefore, from now on we will also consider the *time parameter* t ($t \geq 0$), which represents the time. We will closely discuss two major influences that affect the state of the system. Those are machine failures and restarts and the users' activity.

2.5.1 Machine Failures and Restarts

The Grid consists of many machines that naturally are not immune against occasional failures or necessary maintenance and upgrade periods. Also the Grid has not fixed size concerning the number of connected machines. It is a common feature that machines are added or removed from the Grid as the time passes. *Machine failure* means that either one or more machines within a cluster are not available to execute jobs for some time period. Such failure may be caused by various reasons such as the power failure, the disk failure, the software upgrade, etc. However, we do not differentiate between them in this work. We only focus on the result of a failure, meaning that all jobs that have been — even partially — executed on the failed machine are immediately killed.

Machine restart means that the failure terminated and the machine becomes available for the job processing. Previously killed jobs are not automatically resubmitted. For simplicity, machine failures or restarts can be easily used to "simulate" the permanent removal of some old machine or the addition of a new machine respectively.

As soon as machine failures or restarts are considered in the problem, the Formulas 2.4–2.8 may give incorrect results. It is caused by the fact that the numbers of machines (m) and clusters (l) as well as $\mathcal{M}_k$ (set of machines in cluster k) do not reflect changing number of *available* (non-failed) machines and clusters. Therefore we will modify existing formulas to cover dynamic changes in machine availability. Let m^t and l^t denote the number of available machines and the number of available clusters at time t respectively. Let $\mathcal{M}_k^t$ is the set of available machines of cluster k at time t. Then Formulas 2.4–2.8 can be rewritten as shown in the following Formulas 2.10–2.14.

$$\mathcal{M}^t = \bigcup_{k=1}^{l^t} \mathcal{M}_k^t \qquad \forall t \text{ s. t. } t \geq 0 \tag{2.10}$$

$$cpus^t = \sum_{i \in \mathcal{M}^t} cap_i \qquad \forall t \text{ s. t. } t \geq 0 \tag{2.11}$$

$$cpus_k^t = \sum_{i \in \mathcal{M}_k^t} cap_i \qquad \forall t \text{ s. t. } t \geq 0 \tag{2.12}$$

$$cpus_k^t \geq \sum_{j \in \{j_1, \ldots j_s\}} usage_j \qquad \forall t \text{ s. t. } t \geq 0 \tag{2.13}$$

$$cpus_k^t \geq usage_j \qquad \forall t \text{ s. t. } t \geq 0 \tag{2.14}$$

Such modifications guarantee that only machines available at the given time t will be used[6]. All formulas now provide correct results for any time t, since we require that both $\mathcal{M}^t$ and $\mathcal{M}_k^t$ sets contain only *currently available machines* with respect to the time t.

2.5.2 Users Activity

In the same unpredictable fashion as failures influence the number of available machines, users' actions have impact on the number of jobs currently present in the system. New jobs arrive following their arrival time r_j meanwhile other jobs have already been completed. Let n_c^t is the number of jobs already completed before the current time t. Let n_f^t is the number of jobs that have not yet arrived, i.e., their $r_j > t$. Then, at any given time t the number of jobs currently present in the system is equal to n^t, such that $0 \leq n^t \leq n$ and $n^t = n - n_c^t - n_f^t$.

The arrival time is determined by the user in most cases. In geographically small systems, the user activity is usually higher during the day causing higher load of the system compared with the off-peak night hours [51].

2.5.3 Dynamic Priority

In Section 2.2, the job weight (priority) w_j has been defined. This priority may be fixed during the whole lifetime of the job or it may be dynamically adjusted. This is frequently done by the schedulers to guarantee fairness, to avoid starvation, etc. Therefore, we should introduce new symbol w_j^t to reflect the influence of time t on the current value of the job weight. However, we kindly ask the reader to accept the existing symbol w_j, allowing us to keep the notation as simple as possible[7].

2.6 Optimization Criteria

So far, we have described common constraints that must be met by the scheduling process. There is one aspect left, which is very important — the quality of the generated solution. There are several common criteria that are usually monitored in the real life. These *optimization criteria* are used when making scheduling decisions and represent the goals of the scheduling process. Typically, the goal is to increase the resource usage and the number of successfully completed jobs or to minimize the response time. These criteria are expressed by the value of *objective function* which allows to measure the quality of computed solution and compare it with a different solution. In the following text we present several useful optimization criteria and formal descriptions of corresponding objective functions. We start with the criteria that are often applied to measure the system's performance. Next, job and fairness related criteria will be presented while the last part of this section will describe a criterion used to measure the time requirements of the scheduling algorithm.

[6]For simplicity, we assume that there is some reliable information service that provides correct and up to date information concerning available machines.

[7]From now on, w_j can be understood as a variable, not a constant.

2.6.1 System Related Criteria

We start with a description of three different criteria that allows to measure the performance of the system. First of all, a very common and popular criterion is the *makespan* ($C_{\max}$) [189, 188] which is defined by Formula 2.15.

$$C_{\max} = \max_{1 \le j \le n} C_j \tag{2.15}$$

Clearly, makespan denotes the completion time of the last executed job. As pointed out by Xhafa and Abraham [188], makespan is an indicator of the general productivity of the Grid. Small values of makespan mean that the scheduler is providing good and efficient planning of jobs to resources [188]. As defined, makespan is suitable only for "off-line", static problems where all n jobs are already known. If the system is running for months or even years, it makes a little sense to optimize makespan, since the makespan cannot be computed[8]. Of course, as we will closely discuss in Section 2.7, otherwise dynamic problem can be considered as static at each given time t, and the "partial makespan" can be computed using n^t (the number of jobs in the system at time t) instead of n. Similar rule applies for many other objectives, i.e., they can be used either in the "ex post" fashion, using all n jobs, or in an online but "partial" manner, using only currently available n^t jobs. Since makespan optimizes only the upper bound of jobs' completion times, additional objectives should be applied [188].

Probably the most popular is the *machine usage* criterion [168, 55, 188]. It expresses how much the resource capacity is utilized over the time. It is a very common criterion used by the resource owners who wants to maximize the utilization of their system. Let $\mathcal{EX}^t$ is the set of jobs being executed at the time t, then MU^t denotes the momentary machine usage[9] at this given time t (see Formulas 2.16 and 2.17). The *machine usage MU* corresponds to Formula 2.18.

$$\mathcal{EX}^t = \{j | j \in \mathcal{J}, r_j \le t < C_j, S_j \le t\} \qquad \forall t \text{ s.t. } t \ge 0 \tag{2.16}$$

$$MU^t = \frac{\sum\limits_{j \in \mathcal{EX}^t} \left(\sum\limits_{i \in \mathcal{R}_j} usage_{i,j} \right)}{\sum\limits_{i=1}^{m^t} cap_i} \qquad \forall t \text{ s.t. } t \ge 0 \tag{2.17}$$

$$MU = \frac{1}{C_{\max}} \sum_{t=0}^{C_{\max}} MU^t \tag{2.18}$$

Since heterogeneous machines are considered, the *weighted machine usage WMU* criterion is also useful. As discussed in [169], when a choice is to be made between two machines, it is better to highly utilize the fast machine rather than the slow machine since

[8] Clearly, we do not know when the last n-th job will arrive or complete.

[9] For the given time t the value of MU^t is computed using *currently available machines* (m^t) only.

the fast machine computes much more operations in the given time than the slow one. It is important to notice that such a scenario is not covered by the classic machine usage criterion where only the proportion of used and available CPUs is measured disregarding their speeds. Here the momentary weighted usage WMU^t is expressed by Formula 2.19, where the speed v_i of the machine i is used as a weight. Therefore, the weighted machine usage WMU defined in Formula 2.20 expresses the average amount of utilized CPU operations.

$$WMU^t = \frac{\sum\limits_{j \in \mathcal{EX}^t} \left(\sum\limits_{i \in \mathcal{R}_j} v_i \cdot usage_{i,j} \right)}{\sum\limits_{i=1}^{m^t} v_i \cdot cap_i} \qquad \forall t \text{ s.t. } t \geq 0 \tag{2.19}$$

$$WMU = \frac{1}{C_{\max}} \sum_{t=0}^{C_{\max}} WMU^t \tag{2.20}$$

2.6.2 Job Related Criteria

Previous criteria are suitable to measure the performance of the system. However, they do not provide any information with respect to overall job performance. Such requirements are often expressed by *job related criteria*. Among those, very popular are the response time, the wait time and the slowdown.

The *response time* (RT) represents the average time a job spends in the system, i.e., the time from its submission to its termination (see Formula 2.23). The *wait time* (WT) is the mean time that a job spends waiting before its execution starts (see Formula 2.24). The *slowdown* (SD) is the mean value of all jobs' slowdowns (see Formula 2.25). Slowdown is the ratio of the actual response time of the job to the response time if executed without any waiting. By definition, $SD \geq 1$. As pointed out by Feitelson et al. [54], the use of response time places more weight on long jobs and basically ignores if a short job waits few minutes, so it may not reflect users' notion of responsiveness. Slowdown reflects this situation, measuring the responsiveness of the system with respect to the job length, i.e., jobs are completed within the time proportional to the job length. In another words, it prefers completion of shorter jobs in a shorter time horizon in comparison with time consuming jobs where a longer waiting is acceptable. Wait time criterion complement the slowdown and response time criteria. Short wait times prevent the users from feeling that the scheduler "forgot about their jobs". If not specified otherwise, all following criteria consider *non-failed* jobs only. Job is considered as non-failed if *it has not been killed* due to a machine failure. This guarantees that the resulting values of, e.g., response time will not be distorted due to misleading, short runtimes of such killed jobs. However, jobs that are cancelled by users or are killed due to exceeding their ep_j limit are considered as non-failed in this case, and are included when computing the values of objective functions, since their premature termination is usually fault of the user and not a "system's fault". From this point of view the *number of killed jobs* (KJ) is an important metric showing the amount of "job mortality" caused solely by the failures in the system. The set of *Non-Failed Jobs* ($\mathcal{NFJ}$) is defined by Formula 2.21 while the number of killed jobs is defined in Formula 2.22.

$$\begin{aligned} \mathcal{NFJ} &= \{j \mid j \in \mathcal{J}, j \text{ was not killed due to a machine failure}\} & (2.21)\\ KJ &= n - |\mathcal{NFJ}| & (2.22) \end{aligned}$$

Clearly, when no machine failures appear, then $|\mathcal{NFJ}| = n$ and $KJ = 0$. Formulas 2.23, 2.24 and 2.25 show the objective functions corresponding to the response time, the wait time and the slowdown criteria, considering non-failed jobs only.

$$\begin{aligned} RT &= \frac{1}{|\mathcal{NFJ}|} \sum_{j \in \mathcal{NFJ}} (C_j - r_j) & (2.23)\\ WT &= \frac{1}{|\mathcal{NFJ}|} \sum_{j \in \mathcal{NFJ}} (S_j - r_j) & (2.24)\\ SD &= \frac{1}{|\mathcal{NFJ}|} \sum_{j \in \mathcal{NFJ}} \left(\frac{C_j - r_j}{C_j - S_j} \right) & (2.25) \end{aligned}$$

All three criteria are to be *minimized.* Closer attention requires the slowdown Formula 2.25. Clearly, the denominator $C_j - S_j$ (runtime) must be greater than 0 to prevent division by zero. Moreover, "extremely short" job having very small runtime (e.g., $0 < C_j - S_j < 1$) is dangerous. It often represents job that ended prematurely right after the start due to some error. However, its slowdown can be huge, which may seriously skew the final mean value. Therefore, so called *bounded slowdown* (*BSD*) is often applied [52, 54], where the minimal job runtime is guaranteed to be greater than some predefined time constant, sometimes called a "treshold of interactivity" [54]. However, there is no general agreement concerning the actual value of this threshold. Sometimes it is equal to 10 seconds [52, 54], while different authors use, e.g., 1 minute [181]. Since the resulting *BSD* value is very sensitive to the applied threshold value [140], we set the threshold equal to 1 second in this thesis. It allows us to eliminate problems related to extremely short jobs while keeping the resulting values close (comparable) to the values of "normal" slowdown (*SD*) in most cases. The bounded slowdown is defined by following Formula 2.26.

$$BSD = \frac{1}{|\mathcal{NFJ}|} \sum_{j \in \mathcal{NFJ}} \left(\frac{C_j - r_j}{\max(1, C_j - S_j)} \right) \qquad (2.26)$$

Quality of Service (QoS) is the ability to provide different priority to different jobs and users, or to guarantee a certain level of performance to a job [60]. If the QoS mechanism is supported it allows the users to specify desired performance for their jobs such as completion before given deadline, dedicated access to resources during some time period (advanced reservation), requested amount of resources (CPUs, RAM, HDD, network bandwidth), etc. Such request should be formally established between the user and the resource manager (scheduler) through a negotiation which produces a formal description of the guaranteed service called *Service Level Agreement (SLA)* [60]. However, even in systems which do not explicitly support SLAs, some QoS related mechanisms are often provided. Typically, the *fairness* is required and supported. Generally speaking, jobs belonging to different users should experience similar performance — if not specified otherwise. Also (specific) job requirements (see Section 2.4) are supported and respected by the scheduler. It means that all requests concerning the number of CPUs, the requested CPU architecture, the type and size of the storage infrastructure, etc., are taken into account. In some systems, users are also allowed to specify the deadline d_j for the job, which represents their expectations concerning job completion time. *Minimization of the late jobs* allows to support such specific needs, increasing the service quality. It introduces necessary step toward advanced reservation where both completion and start time are tightly constrained. Other common scheduling criteria related with due dates, namely lateness or tardiness [10], may be also of interest. Still, more precise placement within specific time interval is an ultimate goal. We consider the *late jobs* criterion, measured with the *unit penalty* (U) function.

$$U = \sum_{j \in \mathcal{NFJ}} U_j \tag{2.27}$$

$$U_j = \begin{cases} 1 & \text{if } C_j > d_j \text{ (job } j \text{ is late)} \\ 0 & \text{if } C_j \leq d_j \text{ (job } j \text{ is not late)} \end{cases} \tag{2.28}$$

Clearly, Formula 2.27 expresses the number of late jobs and the goal is to minimize the value of U.

2.6.3 Fairness Related Criteria

So far, all criteria focused either on the system or the job performance. We now present a criterion that focuses on the fairness of the system. The goal is to *maintain fairness* among the users of the system. As far as we know there is no widely accepted standardized metric to measure fairness and different authors use different metrics [147, 148, 146, 181, 117]. A *fair start time* (FST) metric is used in [147, 117]. It measures the influence of later arriving jobs on the execution start time of currently waiting jobs. FST is calculated for each job, by creating a schedule assuming no later jobs arrive. The resulting "unfairness" is the difference between FST and the actual start time. Another metric measures to what extent each job was able to receive its "share" of the resources while in the system [148, 146]. The basic idea is that each job "deserves" $1/n^t$ of the resources, where n^t is the number of jobs present in the system at the given time t. The "unfairness" is computed by comparing the resources consumed by a job with the resources deserved by the job.

[10] Lateness L_j is the amount of time by which the completion time of job j differs from the due date ($L_j = C_j - d_j$). Tardiness is the positive lateness of the job j. Otherwise it is a zero, i.e., $T_j = \max(0, L_j)$.

The paper [181] summarizes several metrics based on the variance, the standard deviation, the coefficient of variation, etc., where smaller resulting values imply higher fairness.

Fairness is usually understood and represented as a *job related* metric, meaning that every job should be served in a fair fashion with respect to other jobs [147, 148, 146, 181, 117]. Such requirements are already partially covered by the common performance criteria like the slowdown and the wait time shown in the previous text. Moreover, job fairness has been also reflected in the design of common scheduling algorithms and the results concerning selected job fairness indicators are well known [148, 146, 181].

In this work, we aim to guarantee fair performance to *different users* of the system as well. Therefore we have proposed a new criterion, called *fairness* (F) shown in Formula 2.34. Our proposal reuses several existing ideas that are discussed in the following text. Fairness criterion minimizes the differences among the normalized mean wait times of all users, thus F is rather a user related than a job related metric. First, we calculate the *normalized user wait time* ($NUWT_o$) for each user (job owner) o, $o \in owner_1, .., owner_u$.

$$\begin{aligned}
\mathcal{JOBS}_o &= \{j | j \in \mathcal{NFJ}, o_j = o\} \text{ (all non-failed jobs of user } o) && (2.29)\\
TUWT_o &= \sum_{j \in \mathcal{JOBS}_o} (S_j - r_j) && (2.30)\\
TUSA_o &= \sum_{j \in \mathcal{JOBS}_o} (p_j \cdot usage_j) && (2.31)\\
NUWT_o &= \frac{TUWT_o}{TUSA_o} && (2.32)
\end{aligned}$$

Normalized user wait time $NUWT_o$ is the *total user wait time* ($TUWT_o$) divided (normalized) by the so called *total user squashed area* ($TUSA_o$), which can be described as the sum of products of the job runtime and the number of requested processors. This user-oriented metric is based on more general *total squashed area* metric proposed in [51]. The normalization is used to prioritize less active users over those who utilize the system resources very frequently [93]. Here we have been inspired by the actual administration policies of the Czech national Grid infrastructure MetaCentrum [127], that use similar normalization when monitoring and adjusting fairness in the production PBS Pro scheduling system [88]. Similar approach has been also adopted in the ASCI Blue Mountain supercomputer cluster when establishing job priorities in the fair-share mechanism [93].

Using the values of $NUWT_o$ we calculate the mean *user wait time* (UWT) following the Formula 2.33. Then the fairness F is measured by calculating the Formula 2.34.

$$\begin{aligned}
UWT &= \frac{1}{u} \sum_{o=1}^{u} NUWT_o && (2.33)\\
F &= \sum_{o=1}^{u} (UWT - NUWT_o)^2 && (2.34)
\end{aligned}$$

The squares in F definition guarantee that only positive numbers are summed and that higher deviations from the mean value are more penalized than the small ones. It is inspired by the widely used *Least Squares method* [183] where similar formula of squared residuals is minimized when fitting values provided by a model to observed values.

In this thesis, the normalized user wait time ($NUWT_o$) is used in the graphs with the experimental results (e.g., Figure 6.7) to reflect the fairness of the applied scheduling algorithms. The closer the resulting $NUWT_o$ values of all users are to each other, the higher is the fairness. If the $NUWT_o$ value is less than 1.0, it means that the user spent more time by computing than by waiting, which is advantageous for the user. Similarly, values greater than 1.0 indicate that the total user wait time is larger than the computational time of his or hers jobs. On the other hand, the fairness (F) criterion is used during the evaluation of performed scheduling decisions, i.e., "inside" scheduling algorithms. Typically, when two possible solutions are available, then the values of F are computed for both of them. The one having smaller F value is considered as more fair. The squares used during computation of F help to highlight unfair assignments which is very favorable when performing scheduling decisions. Sadly, the squares basically prevent us to reasonably interpret the resulting F values as it was possible in case of the $NUWT_o$ criterion. This is the reason why $NUWT_o$ is used in graphs while F is used in the evaluation phase of the scheduling algorithm.

2.6.4 Algorithm Performance

We conclude this section with the *algorithm runtime per one job* (AR) metric shown in Formula 2.35. It measures the average CPU time necessary to create a scheduling decision for one job. Although it is not de facto an optimization criterion[11] it is an important metric that shows the scalability of applied scheduling algorithms with respect to the size of the problem. Let AR_j is the CPU time the algorithm utilizes to develop a scheduling decision(s)[12] for a given jobs j. Then the algorithm runtime per one job is computed using the equation in Formula 2.35.

$$AR = \frac{1}{n}\sum_{j=1}^{n} AR_j \tag{2.35}$$

2.7 Grid Scheduling Problem

We now may formulate the *Grid Scheduling Problem.* It is a problem of finding a *schedule* — a structure that maps jobs onto resources in time, subject to all job and resource related constraints mentioned in Sections 2.2 and 2.3. Especially, Formulas 2.1, 2.2, 2.3, 2.7 and 2.8 must hold in case that the resource pool is *static.* Formula 2.1 guarantees that job in execution will not be preempted, Formula 2.2 guarantees that each job obtains the requested amount of CPUs during its execution, while Formula 2.3 assures that two

[11]We usually do not use AR in the same sense as other criteria, i.e., it is not used to compare some solution against another solution.

[12]In some cases scheduling decision for given job j must be taken more than once, e.g., when rescheduling waiting jobs after a machine failure or due to the optimization routine.

jobs executing simultaneously will have disjunctive sets of assigned CPUs. Formula 2.7 defines the properties of Space Sharing policy, i.e., the upper bound of jobs that can be simultaneously executed on one cluster, while Formula 2.8 guarantees that a job will not be executed on a cluster that does not have enough CPUs. If specific job requirements are used, then also the Formula 2.9 from Section 2.4 must be used when selecting suitable cluster for the given job.

In case that the resource pool is *dynamic*, thus the number of available machines may change through the time, then the constraints from Section 2.5, i.e., especially Formulas 2.13 and 2.14 must be used instead of "static" Formulas 2.7 and 2.8. The difference is that neither the number of jobs nor the number of resources is stable during the scheduling process due to the system dynamics and the users' activity. Although the set of available machines and the set of jobs is dynamically changing through the time still the considered scheduling problem can be understood to be static at the given time t. At any time t ($0 \leq t \leq C_{\max}$) the scheduler has complete knowledge of n^t jobs *currently* present in the system as discussed in Section 2.5.2. Those are waiting and running jobs. At the same time all jobs having their $r_j > t$ are not known to the scheduler since they will arrive in the future [56]. As we have already mentioned in the Section 2.5, additional dynamic changes such as resource failures or restarts can change the set of available machines, thus m^t, l^t and the $\mathcal{M}_k^t$ values must be used for given t, representing the number of currently operational machines, clusters and the set of operational machines in cluster k, respectively. To conclude, at the given time t the scheduler solves the scheduling problem of assigning n^t jobs onto m^t machines, such that for every $j \in \{j_1, .., j_{n^t}\}$ holds $r_j \leq t$ and $C_j \geq t$. The scheduling process is subject to given constraints that are defined by the Formulas 2.1–2.3, 2.13, 2.14 and optionally 2.9.

If the schedule satisfies such constraints it represents a *feasible solution* of the Grid scheduling problem. However, the *quality* of such solution may be very far from optimum. Therefore, the Grid scheduling problem also involves the selected *optimization criteria* that measure the quality of the problem's solution and are used to guide the scheduling technique toward better solutions.

2.7.1 Complexity of the Grid Scheduling Problem

So far, we have defined the Grid scheduling problem. We now focus on the problem complexity. Efficient scheduling in a heterogeneous computing environment such as Grid is clearly important if good performance and utilization shall be guaranteed. However, finding optimal schedules in such a system has been shown, in general, to be *NP*-hard [144, 190], because the Grid scheduling problem is a generalized reformulation of *Multiprocessor Scheduling [SS8]*[13] problem which has been shown to be *NP*-hard [62].

In fact, since the Grid environment is highly dynamic, it is not possible to even *define the optimality of scheduling* [188], as it is defined in combinatorial optimization. In Grid, the scheduler runs as long as the Grid system exists and thus the performance is measured not only for particular applications but also in the long run [188].

[13]The code in square brackets is a reference to the classification used in Garey and Johnson [62].

Moreover, it is easy to show that even significantly simplified subproblems are intractable [104]. Let us consider one machine with several CPUs which correspond to several identical machines in parallel, denoted as P. A simplified subproblem involves one machine, sequential jobs, and $C_{\max}$ as a single objective. Assume that the problem is to be solved at the given time t, thus the release dates can be omitted as well. Using Graham's $\alpha|\beta|\gamma$ notation [71], this problem is formulated as $P||C_{\max}$. Even this problem is known to be *NP*-hard [174]. Let us consider other similar problem, where sequential jobs are having deadlines and U is a single objective (number of late jobs). Again, the problem is to be solved at the given time t, ignoring the release dates. This problem can be formulated as $P||U$. This problem is also *NP*-hard because there exists a polynomial Turing reduction from a known *NP*-hard problem $P||C_{\max}$. Clearly, makespan can be transformed to unit penalty function using reduction $C_{\max} \rightarrow L_{\max} \rightarrow U$ [174]. Similar problem involving jobs with deadlines has been proven to be *NP*-hard in [77].

2.7.2 Conclusion

Based on the previous definitions and characteristics we may now conclude this chapter by saying that the Grid scheduling problem is a problem of finding a feasible schedule subject to the job and the resource related constraints, the system and user related dynamics and the applied optimization criteria. For the purposes of this thesis the problem is considered as centralized[14]. Since the problem is *NP*-hard, the found solution will most likely be *suboptimal*.

Finally, let us briefly formulate some basic requirements on the actual *Grid scheduler*. First of all, the dynamic character of the problem implies additional constraint which is a *bounded time* available to the scheduler for making scheduling decisions. The problem is *online* but not in the strong, hard real-time, sense such as in the case of a multi-tasking CPU scheduler. If the runtime of the scheduling algorithm is lower than the inter-arrival time between two successive events such as job arrivals or job completions, it is considered to be *acceptable* since a dedicated computer for scheduling can be expected for large Grid computing systems[15]. Even more, a small amount of time spent on scheduling certainly makes sense if it allows to increase the quality of the generated schedule. In addition, optimization procedures may use much more time during less critical hours — typically during the night [51]. The quality of the final schedule has to be monitored by objective functions that represent proper criteria as were described in Section 2.6.

Second, since finding optimal schedule for the Grid scheduling problem is intractable, *non-optimal scheduling* techniques such as heuristics or metaheuristics [186, 138, 137, 68] have to be used [188] by the proposed scheduler. As discussed in Section 2.7.1, it would be quite questionable to search for *optimal solutions*, since various criteria can be followed simultaneously and their suitable combination may not be exactly clear, therefore it would not be possible to even define the optimality [188].

[14]As we have mentioned in Section 2.1, *decentralized scheduling* is not considered in this thesis. Still, most of the definitions we have introduced remain valid also for the decentralized problem.

[15]Of course, sometimes a large set of jobs can arrive at one moment, requiring some nontrivial runtime of the scheduling algorithm.

Finally, the resulting solution should remain scalable and robust, thus able to deal with uncertainty and incomplete information regarding the jobs and the Grid itself.

Chapter 3

State of the Art

In the previous Chapter 2 we have provided detailed description of the Grid scheduling problem. In this chapter we discuss typical techniques commonly applied to solve this problem. We start with a general overview of search and optimization techniques. Next, the principles of heuristic and metaheuristic techniques are described, since they represent de facto standard approaches and are widely applied in production and experimental systems as well as in the literature. Then, a detailed description of queue-based approach is presented. Here we discuss all important features such as common scheduling algorithms, queue(s) management, as well as handling of dynamic real life features such as job arrivals, machine failures or inaccurate processing time estimates. Similarly, detailed description of schedule-based approach follows. Here we focus on the problem of schedule construction and optimization, thus an overview of existing scheduling approaches is provided. Also the reactions on dynamic changes concerning the state of the system are discussed. Since both queue-based and schedule-based systems often use processing time estimates to improve the quality of scheduling, the chapter concludes with an overview of several techniques that can be used to refine inaccurate processing time estimates provided by the user.

3.1 Overview

There are several popular techniques that are used to solve various (complex) optimization problems [23, 68]. These scientific methods are applied to solve problems arising in the direction and management of large systems of, e.g., men, machines, materials and money, in industry, business, government or defense [63]. These techniques differ in the way the solution is being constructed.

The most trivial technique is an *exhaustive search* which tests every possible assignment [22]. Once all assignments have been tested the best one is chosen. This method is very simple but very inefficient, therefore it is useless for any larger problem.

Another popular method is the *Constraint Programming* [128, 68, 76, 57] which uses known properties of the given problem to derive constraints. These constraints are then used to guide the solution technique. Here, so called constraint propagation is used to efficiently decrease the search space of possible solutions, eliminating those that represent a conflict with some constraint. Typical natural constraint is that a CPU can execute only one job at a time. If some solution fails to meet such requirement then it is not accepted. Except for such “hard constraints” also so called “soft constraints” can be used [128], to represent some preferences or expectations regarding the final solution. Typical example is that a job with a higher priority should be completed earlier than a non-priority job. If a solution candidate fails to meet such requirement it still can be accepted [23].

Mathematical Programming is a large family of optimization techniques involving, e.g., linear, integer, or nonlinear programming [49, 137]. It can be used for finding optimal schedules subject to given constraints and objectives. Linear programming often uses simplex method [40] to solve problems where both objectives and constraints can be expressed as linear functions [49]. Integer programming is a special case of linear programming, where the resulting values of variables are required to be integers. Here, branch and bound or cutting plane (branch and cut) methods are often used to find the solution [17, 49]. Similar methods can be used in nonlinear programming for problems having nonlinear objective functions or constraints.

Exact (deterministic) solution techniques [137, 49] such as simplex method, branch and bound or dynamic programming always produce the same result given the same input. On the other hand, there are several *nondeterministic approaches* that may produce different results if executed several times over the same input [23]. The reason is the application of stochastic decision mechanism involving some form of a random function. Typically, some popular heuristic algorithms that are described in the following text are nondeterministic.

Heuristic algorithms [137, 22, 69, 23] are a large group of methods that includes, e.g., dispatching rules and policies [137], local search [23, 137, 57], genetic algorithms [141, 137] or several nature inspired approaches, such as the evolutionary computation [58] or the ant colony optimization [192, 46]. Many real life problems are large and require fast solutions. In such situation a technique to obtain high quality solutions in reasonable time is necessary. Such approach is called a heuristic. Heuristic algorithms do not guarantee that an optimal solution will be found, good solutions in (relatively) short time are preferred instead. However, in some situations we cannot reach good solutions or even decide how far or close the generated solution is from the optimal solution [142].

Due to several reasons like those we have already mentioned in Section 2.7.2, heuristics have become de facto standard techniques for solving the Grid scheduling problem and other approaches are used rather rarely. Therefore, we will closely describe several common types of heuristics in the following text.

3.2 Heuristic Approaches

In this section we will briefly describe the principles of commonly used scheduling techniques applied in the Grids. We will mention *dispatching rules, scheduling policies, list scheduling* and *metaheuristics.*

3.2.1 Dispatching Rules and Scheduling Policies

We start our description with two basic scheduling techniques called dispatching rules and scheduling policies. Dispatching rules and policies are typically simple heuristics that are used to select the ordering of job execution on the resources. Although these two techniques are sometimes interchanged in the literature, we differentiate between them using the definition from [137].

Dispatching rules are used for *deterministic scheduling* where all information regarding jobs and machines are known at the beginning and do not change over the time, while *scheduling policies* are used in case that new information concerning jobs and machines become available continuously during the time. Here the scheduler can decide at any time according to the currently available information [137]. Since the Grid scheduling problem is dynamic, from now on, we will use the term *scheduling policy.* Still, both dispatching rules and scheduling policies usually apply the same principles.

Scheduling policies [68, 137] can be used for various purposes. Often, they are used for *job selection*, i.e., to select the ordering of job execution on the resources. In case that some resource becomes available, the job with the highest priority — given according to the applied policy — is selected for execution. Policies can be divided into two groups: *static* and *dynamic.* Static ones are time independent, while dynamic policies can generate different decisions as the time proceeds[1]. For example, at some point in time, job j_1 may have higher priority than job j_2, while at a later point in time, they may have the same priority [138].

Also, policies are used during the *resource selection* process, i.e., if multiple suitable machines are available in the system, then some policy is used to decide which machine(s) is selected for the job [138].

Basic policies are useful to find a reasonably good schedule with regard to a single objective. However, to address more complicated objective functions, elementary policies may not be sufficient. In these cases, several elementary policies can be combined together to form a composite policy [138, 137], where each elementary policy has its own scaling parameter that weights its importance [31].

So far, we have introduced some basic types of scheduling policies. Examples and descriptions of actual scheduling policies will be presented in Section 3.3, where the typical solutions applied in nowadays scheduling systems will be discussed.

3.2.2 List Scheduling

Although our work does not focus on workflows, we should mention standard technique used to schedule such type of applications. *List scheduling* is a dominant heuristic technique for scheduling parallel task with the precedence constraints. In its simplest form, the first part of the algorithm sorts the nodes of the task graph (e.g., DAG) to be scheduled according to a priority scheme, while respecting the precedence constraints of the nodes. After this step, the node list is topologically ordered [155]. The sorting phase is usually

[1]Please note that the term *dynamic* does not imply dynamic changes of the system (jobs, machines) as described in Section 2.5. Here, it refers to changing time only.

performed by some policy. In the second step, each task from the list is successively scheduled to a processor chosen by some policy — usually the one allowing the earliest start time of the task. Various variants of list scheduling exists that differs in the way the priority scheme for the tasks is defined and the way the processor is chosen [155].

3.2.3 Metaheuristics

Metaheuristic is de facto an iterative generation process that guides "subordinate" heuristic to produce better solutions [69]. It designates a computational method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. Metaheuristics make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions. However, metaheuristics do not guarantee an optimal solution is ever found [68, 142]. Many metaheuristics implement some form of stochastic optimization [68].

Algorithms based on *Local Search (LS)* [138, 137] belong to the family of metaheuristic algorithms [68]. Unlike constructive methods such as branch and bound or backtracking which start with empty schedule, LS based techniques start with existing — so called *initial schedule* and try to improve its quality during the computation. Initial schedule may be generated randomly or it may be constructed by some other technique such as dispatching rule or policy. LS tries to improve the initial schedule by performing *local changes* that modify the schedule. Algorithm 3.1 shows the general structure of the LS algorithm.

By the neighborhood of a schedule we understand a set of new schedules which are created from the previous one with a single local change. By local change we can understand moving one job into new position or a swap of two jobs. The applied definition of neighborhood is very important. It can affect the quality of the solution as well as the speed of the search procedure. According to the type and complexity of the problem the size of the neighborhood may vary since some moves may result in infeasible schedules. Therefore, it may be useful to restrict the possible moves to those which guarantee feasibility of the schedule. Then, the size of the search space can be decreased and more moves can be evaluated in the same time. On the other hand, in some situations better or even optimal solutions can be reached only through steps that temporarily lead to infeasible schedules. Existing LS algorithms differ mainly in the way the new solution is accepted. *Hill climb* algorithm [137, 23] accepts only those moves that improve the value of objective function, i.e., those that improve the quality of the solution. Such approach may cause that only local optimum is reached because global optimum is often reachable only through moves that temporarily result in the worse values of objective function.

Random Walk (RW), *Simulated Annealing (SA)* [137, 11] or *Tabu Search (TS)* [69, 137] sometimes accept moves that leads to worse values of objective function. RW and SA apply probabilistic approach such that improving moves are always accepted and non-improving moves are accepted with some probability. RW uses fixed (low) probability while SA uses *Metropolis criterion* which lowers the acceptance probability in time by decreasing stepwise the temperature parameter which is one of the inputs of the algorithm [75]. *Tabu search* algorithm uses short memory called *tabu list* where few previous moves are stored. If the algorithm is trying to repeat previously performed move this change is not allowed

Algorithm 3.1 *General Local Search Algorithm*

```
1: $schedule_{current}$ := $schedule_{initial}$;
2: $schedule_{best}$ := $schedule_{initial}$;
3: $stopping\ condition$ := **false**;
4: **while** $stopping\ condition$ = **false** **do**
5:    $schedule_{candidate}$ := select schedule $\in$ neighborhood of $schedule_{current}$;
6:    **if** $schedule_{candidate}$ is acceptable **then**
7:       $schedule_{current}$ := $schedule_{candidate}$;
8:       **if** $schedule_{current}$ is better than $schedule_{best}$ **then**
9:          $schedule_{best}$ := $schedule_{current}$;
10:      **end if**
11:   **end if**
12:   refresh $stopping\ condition$;
13: **end while**
14: **return** $schedule_{best}$
```

if this move is present in the tabu list, protecting the algorithm against short cycles. Tabu list has limited size and the oldest item is always removed when the list becomes full.

Existing LS algorithms also differ in the implementation of the *stopping condition*. If a global optimum is reached the algorithm should stop in all cases. However, in many situations it is not obvious when the global optimum is reached. For example when decreasing the number of delayed jobs the objective function U will be equal to the number of late jobs. Then the global optimum is a schedule with $U \geq 0$. The value of global optimum is different for different setups of the problem and unless we found schedule where $U = 0$ we cannot be sure whether the optimum was reached or not. Moreover, several objectives are usually followed in the real world, therefore additional stopping conditions are used, such as stopping when given number of iterations (moves) is reached. The cycle may also stop when the fixed number of non-improving moves is reached [68].

Genetic Algorithm (GA) [141] applies techniques inspired by the evolutionary biology such as inheritance, mutation, selection, and crossover. In comparison with former LS techniques, GA maintains and improves a set of solutions — so called population. Usually, GA starts with randomly generated initial population of chromosomes representing the candidate solutions. In each generation, the fitness of every chromosome in the population is evaluated. Then, based on their fitness value, multiple individuals are selected from the current population and modified to form a new population. New population is used in the next iteration. Commonly, the algorithm terminates when either a maximum number of generations was produced, or a satisfactory fitness level was reached.

Metaheuristics such as Local Search or Genetic Algorithms represent popular optimization techniques commonly used for solving variety of scheduling problems such as manufacture scheduling [138], workforce scheduling [21], timetabling [136, 132] or spacecraft mission planning [33]. As we will show in Section 3.4.2, metaheuristics are frequently applied to solve job scheduling problems. Detailed information about these and various other metaheuristics can be found in [167, 68, 23, 176, 124].

So far, we have introduced general overview of some types of heuristic algorithms that are widely used when scheduling jobs onto computational resources. In the following Sections 3.3 and 3.4 we will describe how the aforementioned approaches are applied in the Grid and cluster environment. We will focus on the main features of the queue-based and schedule-based scheduling systems, while discussing the principles and the application of typical scheduling algorithms.

3.3 Queue-based Approaches

In this section we present the principles of current production Grid and cluster scheduling systems and we provide detailed description of selected scheduling algorithms that are commonly used by these systems. All major production systems such as PBS Pro [88], LSF [191], Sun Grid Engine (SGE) [66], Condor [173], gLite WMS [24], Maui and Moab [7], GridWay [82], Unicore [145], etc., are so called queueing systems. It means that these systems follow the queue-based scheduling approach, using one or more incoming *queues* where jobs are stored until they are scheduled for execution. A scheme of such a system is shown in Figure 3.1. The theoretical background of queueing systems was studied and described in many publications, such as [35, 5]. The scheduling process consists of several steps that define the behavior of the whole scheduler. For the purpose of this thesis we define two major categories: the *queue management* and the *scheduling policy*.

The *queue management* is the ongoing process of assigning jobs onto specific positions

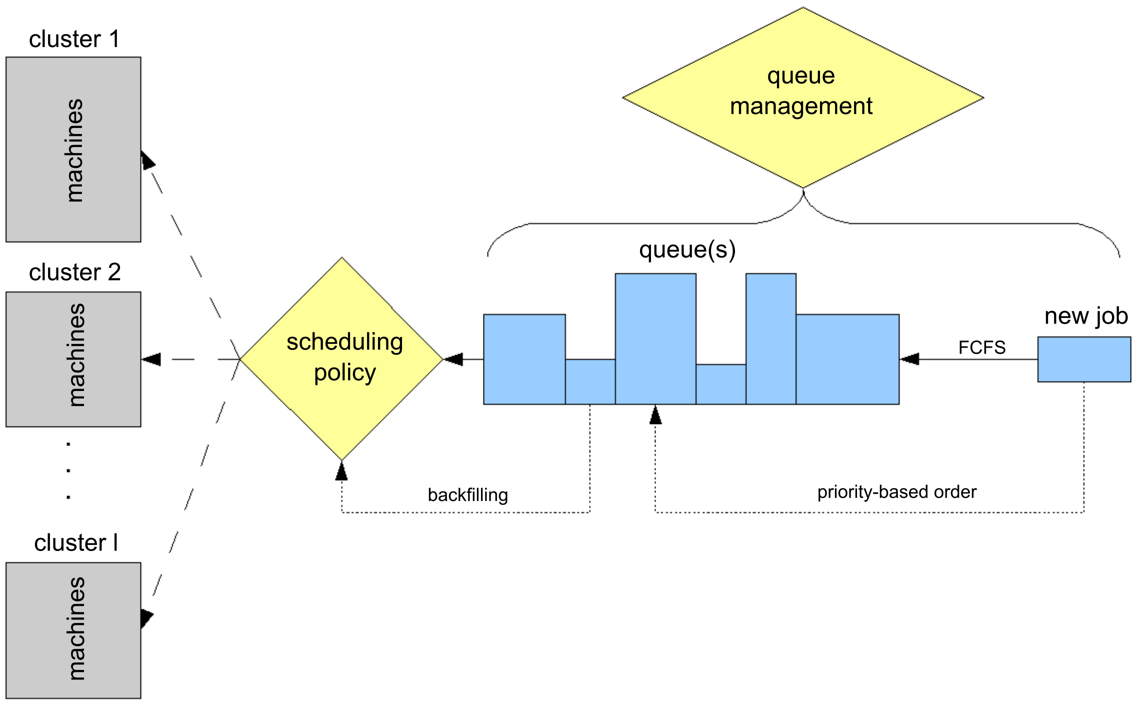


Figure 3.1: Queue-based system.

in the specific queues. Typically, the queue management is responsible for moving newly submitted job into a queue. If there are more queues in the system, the proper one is selected[2]. If the jobs in that queue are ordered according to some priority w_j, then the queue management is responsible for assigning such priority to the newly arriving job. If the priority of a job can change through the time, then the queue management does this update as well. If no priorities are used, then the job is placed at the end (tail) of the target queue. Therefore, all jobs in such queue are ordered according to their arrival time with the oldest job being at the head of the queue. Once the jobs are stored in the queue, the queue management is responsible for keeping the proper ordering of jobs in the queue. If the job priorities are used, then the queue management guarantees that jobs in the queue are ordered according to their priority in the decreasing order[3].

If multiple queues are present in the system, then the queue management applies some policy to define the order in which the queues will be selected by the scheduling policy. Typically, *Round-robin* algorithm [88] is used, selecting the queues in a circular fashion. Once the ordering is established, the scheduling policy either attempts to run one job from each queue or all jobs from the currently selected queue are checked before the next queue is processed. Also the combination of these approaches can be used [139, 88]. Queues are selected according to their priorities which are set up by the system administrator.

The *scheduling policy* is used for selecting jobs from the queue for execution. Job is scheduled if all resources it requires are currently available on some cluster. It means, that all constraints defined in Section 2.7 are met by such cluster. If there are multiple choices (i.e., clusters) to execute the job a process called *node allocation* is used to select the “best resource” from the list of all available resources. Again, this process involves some heuristic that makes this decision. *First Fit (FF), Best Fit (BF), Min Load First (MLF)* or *Fastest Resource First (FRF)* are some of many policies which are used for this purpose in the production systems [7]. First Fit schedules jobs onto first suitable resource, while Best Fit chooses the node that has the fewest available resources but still enough to execute successfully the job while respecting all job constraints. This helps to keep large nodes free for large jobs and lowers resource fragmentation. Min Load First do the exact opposite, i.e., the cluster with the highest number of idle CPUs is chosen [20, 7]. Fastest Resource First selects the fastest available resource. If not specified otherwise, all presented scheduling algorithms in this thesis use the First Fit when selecting the execution site for the job. For this purpose, clusters are ordered according to the total number of CPUs in a descending order, therefore the largest cluster is checked first[4]. If two or more clusters have the same size then the fastest one is tried first.

Various scheduling policies can be used to decide which job will be scheduled. We describe the most common ones in the following text. Moreover, we will also closely discuss several scenarios that are not usually covered by generally available descriptions of

[2]Typically, the target queue is directly specified upon each job submission.

[3]It means that the first job at the head of the queue has the highest priority. Jobs with the same priority are ordered according to their arrival time.

[4]It helps to save necessary algorithm runtime, since the chance that enough CPUs will be available is generally higher when the largest cluster is tried first. Moreover, unlike the BF, FRF or MLF, not all suitable clusters have to be tested.

scheduling algorithms. More precisely, we will describe what actions are taken when, e.g., processing time estimates are inaccurate or machine failures appear. Such description is often missing in the literature. However, these actions represent an important functionality that must be provided when creating realistic simulations or building production solutions.

3.3.1 First Come First Served

All systems support basic *First Come First Served (FCFS)* scheduling policy [88, 156, 109]. FCFS always schedules the first job in the queue, checking the availability of the resources required by such job. If all the resources required by the first job in the queue are available, it is immediately scheduled for the execution, otherwise FCFS waits until all required resources become available. While the first job is waiting for the execution none of the remaining jobs can be scheduled, even if the required resources are available. FCFS is shown in Algorithm 3.2.

Algorithm 3.2 *FCFS*

```
1: stopping condition := false;
2: if queue is empty then
3:    stopping condition := true;
4: end if
5: while stopping condition = false do
6:    j := first job in queue;
7:    if j in queue can be executed then
8:       k := select cluster using FF policy;
9:       remove j from queue and send it on k;
10:      if queue is empty then
11:         stopping condition := true;
12:      end if
13:   else
14:      stopping condition := true;
15:   end if
16: end while
```

Despite its simplicity, FCFS approach presents several advantages. It does not require an estimated processing time of the job and it guarantees that the response time of the first job does not depend on the execution times of remaining jobs. On the other hand, if parallel jobs are scheduled then this fairness property often implies a low utilization of the system resources, that cannot be used by some “less demanding” job(s) from the queue [172]. Many approaches were applied to solve this problem which we describe in the following Section 3.3.2.

3.3.2 Backfilling Algorithms

Algorithms using *backfilling* are an optimization of the FCFS algorithm that try to maximize the resource utilization [121]. Backfilling requires that each job specifies its estimated execution time, so that the scheduler can predict when jobs will be finished.

We start with EASY (Extensible Argonne Scheduler sYstem) Backfilling [156] algorithm that is shown in Algorithm 3.3. It works as FCFS but when the first job in the queue cannot be scheduled because specified amount of resources is not yet available (line 17), it calculates its earliest possible starting time using the processing time estimates of running jobs. Then, it makes a reservation to run the job at this pre-computed time (line 19)[5]. Next, it scans the queue of waiting jobs and schedules immediately every job not interfering with the reservation of the first job [121] (lines 11-16). This helps to increase the resource utilization, since idle resources are *backfilled* with suitable jobs, while decreasing the average job wait time. If a reservation has been created for the first job in the queue, it remains valid (line 5) until this job is finally sent for execution. As soon as this job starts to execute, this reservation is cancelled since it is not needed anymore (line 15). The algorithm finishes when the whole queue has been tested (line 24).

EASY Backfilling takes an aggressive approach that allows short jobs to skip ahead provided they do not delay the job at the head of the queue. The price for improved utilization of EASY Backfilling is that execution guarantees cannot be made because it is hard to predict the size of delays of jobs in the queue. Since only the first job gets a reservation, the delays of other queued jobs may be, in general, unbounded [131][6].

EASY Backfilling is supported by all major production systems such as Condor [185], PBS Pro [88], Maui [86], Moab [7] or LSF [139].

While EASY Backfilling makes reservation for the first job only, Conservative Backfilling [52, 159, 119] makes the reservation for every queued job which cannot be executed at the given moment as shown in Algorithm 3.4. It means that backfilling is performed only when it does not delay any previous job in the queue. Here the scheduling decisions are made upon job submittal, thus we can predict when each job will run, giving the users execution guarantees. Users can then plan ahead based on these guaranteed response times. Obviously, there is no danger of starvation as a reservation is made for every job that cannot be executed immediately. Apparently, such approach places a greater emphasis on predictability [131, 52].

The Algorithm 3.4 shows the pseudo-code of Conservative Backfilling. Unlike the EASY Backfilling, it makes a reservation for every job that cannot start at the given moment (see line 18), by calculating its earliest possible starting time using the processing time estimates of running jobs. The duration of the reservation period is based on the job's estimated processing time (ep_j). These reservations are stored in a set ($reservations$) which is reused every time the algorithm is executed (line 5). No job being scheduled for execution can collide with some reservation from this set (see line 11). When some job having a reservation is finally scheduled for execution its reservation is removed from the $reservations$ set (line 15). Algorithm terminates when either all jobs in the queue were scheduled for execution or the reservations for them were established (line 23).

[5]The reservation is not created if the job already has a reservation, typically from the previous algorithm execution (see the condition at line 18).

[6]If a job is not the first in the queue, new jobs that arrive later may skip it in the queue. While such jobs do not delay the first job in the queue, they may delay all other jobs. Therefore, the system cannot predict when a queued job will eventually run [131].

Algorithm 3.3 *EASY Backfilling*

```
1: if size of queue = 0 then
2:    reservation := null;
3:    stopping condition := true;
4: else
5:    reservation := previously established reservation;
6:    stopping condition := false;
7:    job index := 1;
8: end if
9: while stopping condition = false do
10:   j := job at position job index in the queue;
11:   if job j can be executed not colliding with reservation then
12:      k := select cluster using FF policy;
13:      remove j from queue and send it on k;
14:      if job index = 1 then
15:         reservation = null; (if job had a reservation, cancel it)
16:      end if
17:   else
18:      if job index = 1 and reservation = null then
19:         reservation := make reservation for j using FF policy;
20:      end if
21:      job index := job index + 1;
22:   end if
23:   if job index > size of queue then
24:      stopping condition := true; (the whole queue has been tested)
25:   end if
26: end while
```

3.3.3 Priority-based Algorithms

In all previously described algorithms the incoming queue is ordered according to jobs' arrival times. If one wants to assign different priorities to different jobs or users [170] *priority-based scheduling policies* can be used. There are many such solutions, e.g., *Shortest Processing Time First (SPTF)* [88], *Longest Processing Time First (LPTF)* [120] or *Earliest Deadline First (EDF)* [14, 27] which assign different priorities to jobs according to their estimated execution time (ep_j) or specified deadline (d_j) respectively, and the queue order is defined according to these parameters. *Fair Share* [88] is an example of priority-based policy, where jobs are ordered such that the usage of the system is distributed among Grid users or groups according to some proportion, usually selected by the system administrator.

We will closely describe the *Earliest Deadline First (EDF)* [14, 27] policy, that represents the simplest solution when prioritizing jobs according to their deadline. EDF policy is described in Algorithm 3.5. Clearly, the EDF algorithm is a simple extension of FCFS,

Algorithm 3.4 *Conservative Backfilling*

```
1: if size of $queue = 0$ then
2:   $reservations := \emptyset$;
3:   $stopping\ condition :=$ true;
4: else
5:   $reservations :=$ previously established $reservations$;
6:   $stopping\ condition :=$ false;
7:   $job\ index := 1$;
8: end if
9: while $stopping\ condition =$ false do
10:   $j :=$ job at position $job\ index$ in the $queue$;
11:   if job $j$ can be executed not colliding with existing $reservations$ then
12:     $k :=$ select cluster using FF policy;
13:     remove $j$ from $queue$ and send it on $k$;
14:     if $\exists reservation_j \in reservations$ for job $j$ then
15:       $reservations := reservations \setminus reservation_j$;
16:     end if
17:   else
18:     $reservation_j :=$ make reservation for $j$ using FF policy;
19:     $reservations := reservations \bigcup reservation_j$
20:     $job\ index := job\ index + 1$;
21:   end if
22:   if $job\ index >$ size of $queue$ then
23:     $stopping\ condition :=$ true;
24:   end if
25: end while
```

where jobs with approaching deadlines are shifted toward the head of the queue disregarding their arrival time. The proper ordering can be performed either at each EDF execution (as shown in line 1), or upon each new job arrival, i.e., as a part of the queue management process (see Figure 3.1). Still, EDF faces similar problems as FCFS concerning the poor machine usage. Moreover, when some waiting jobs are already delayed they will remain at their positions at the head of the queue. This may threaten other jobs in the queue whose deadline is approaching and they will be delayed too.

To solve these issues a combination of EASY Backfilling [156] and priority function has been used in the *Flexible Backfilling* [172] algorithm. In the Flexible Backfilling, a priority value w_j is computed for each job j by exploiting a set of heuristics. Each heuristic follows a different strategy to satisfy both users and system administrator requirements. The final

Algorithm 3.5 *Earliest Deadline First*

```
1: sort the $queue$ according to $d_j$ in an increasing order;
2: execute $FCFS$;
```

priority value w_j assigned to each job is the sum of the values computed by each heuristic. Priority values are re-computed at each scheduling event, typically at each job submission or job completion. Three heuristics are used by Flexible Backfilling: *Aging*, *Deadline*, and *WaitMinimization*[7].

Aging aims to avoid job starvation. For this reason, higher scores are assigned to those jobs which have been present in the queue for a longer time. The value of the *Aging* heuristic for given job j is computed using formula 3.1. Here, t is the current time and $t - r_j$ is the current wait time of job j, while $agefactor$ is a multiplicative factor set by the administrator according to the adopted system management policies.

$$Aging(j,t) \quad = \quad agefactor \cdot (t - r_j) \tag{3.1}$$

Deadline aims to maximize the number of jobs that terminate their execution within their deadline. It requires an estimation of the job execution time in order to evaluate its completion time with respect to the current time. The heuristic assigns a minimal value (Min) to any job whose deadline is far from its estimated termination time. When the distance between the completion time and the deadline is smaller than a threshold value (t_j), the score assigned to the job is increased in inverse proportion with respect to such distance. The threshold value may be tuned according to the importance assigned to this heuristic. Without loss of generality, when a job is already over its deadline, its updating priority value is set to Min while jobs with a closer deadline receive higher priority. This strategy improves the number of jobs executed within their deadline, removing previously discussed problems of the EDF policy. Let ep_j to be the estimated execution time of job j.

$$\begin{aligned}
ep_j^{exp} &= ep_j \cdot \frac{v_i^{est}}{v_i^{max}} \\
C_j^{exp} &= t + ep_j^{exp} \\
t_j &= d_j - estfactor \cdot ep_j^{exp} \\
\alpha_j &= \frac{Max - Min}{estfactor \cdot ep_j^{exp}}
\end{aligned}$$

Then, ep_j^{exp} denotes the expected processing time of job j and C_j^{exp} denotes the expected completion time of the job j with respect to the current time t. Here v_i^{max} is the speed of the fastest available machine i (optimistic prediction), and v_i^{est} is the speed of the machine that has been used to estimate the execution time of j[8]. Parameter t_j is the time from which the job's priority must be increased allowing to meet its deadline, while $estfactor$ is a constant which permits to overestimate ep_j^{exp}. The last parameter is α_j,

[7]If necessary, more heuristics can be used as was demonstrated by the authors of Flexible Backfilling in [172], where *Licenses* heuristic is used to optimize utilization of software licenses in the system.

[8]Flexible Backfilling expects that a sample execution of each job has been performed, therefore v_i^{est} is known. If v_i^{est} is not known, ep_j can be used as ep_j^{exp} [172].

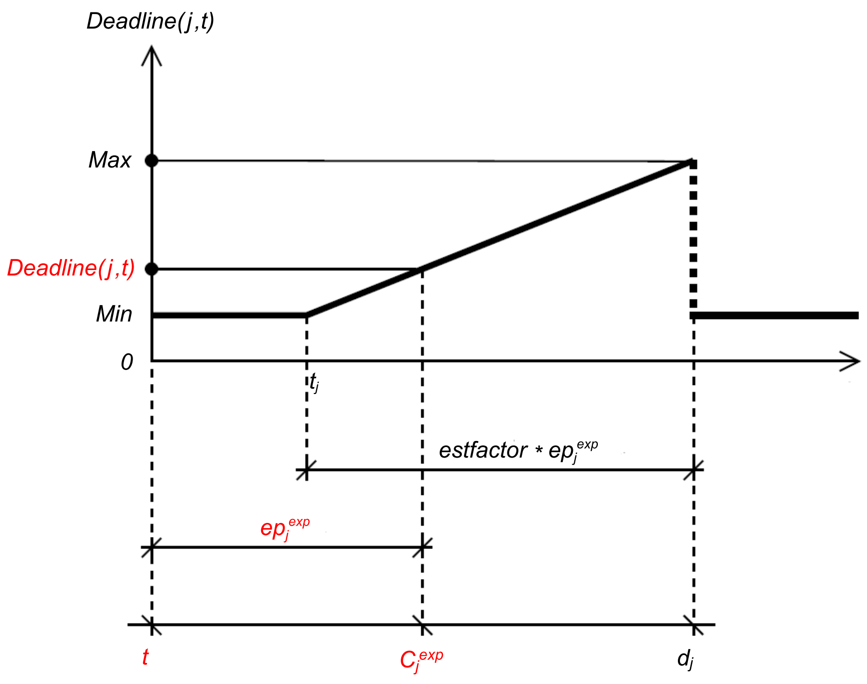


Figure 3.2: Graphical representation of *Deadline*(j, t) heuristic.

which is the angular coefficient of the straight line passing through the points (t_j, Min) and (d_j, Max). Then, *Deadline* heuristic is computed according to the Formula 3.2.

$$Deadline(j,t) = \begin{cases} Min & \text{if } C_j^{exp} \leq t_j \\ \alpha_j(C_j^{exp} - t_j) + Min & \text{if } t_j < C_j^{exp} \leq d_j \\ Min & \text{if } C_j^{exp} > d_j \end{cases} \quad (3.2)$$

A graphical representation of *Deadline* heuristic is shown in Figure 3.2[9]. Red color is used to show how the resulting value is computed at time t using ep_j^{exp} value. The result is greater than Min since $t_j < C_j^{exp} \leq d_j$ holds (see Formula 3.2).

Finally, *WaitMinimization* favors jobs with the shortest estimated execution time. The rationale is that shorter jobs should be executed as soon as possible to improve the average wait time and slowdown of jobs in the queue. Moreover, their execution will not probably cause huge delays (slowdowns) for longer jobs. Let *boostvalue* be the factor set by an administrator[10] and ep_{min} is the minimal ep_j with respect to the currently queued jobs. The value of *WaitMinimization* heuristic is computed using Formula 3.3.

[9]Figure 3.2 is based on the scheme presented in [27].

[10]In their works [172, 105], the authors of Flexible Backfilling have used hand-tuned values of all necessary parameters: $agefactor = 0.01$, $estfactor = 2.0$, $Max = 20.0$, $Min = 0.1$ and $boostvalue = 2.0$. We preserve the values of these parameters to allow straightforward comparison.

$$WaitMinimization(j) = \frac{boostvalue \cdot ep_{min}}{ep_j} \tag{3.3}$$

Together, Flexible Backfilling is implemented as shown in Algorithm 3.6. At each scheduling event such as job arrival or job completion, the values of w_j for all queued jobs are recomputed by these heuristics so that $w_j = Aging(j,t) + Deadline(j,t) + WaitMinimization(j)$. Next, the queue is sorted according to the new w_j values (line 5) and the classic EASY Backfilling procedure starts.

Algorithm 3.6 *Flexible Backfilling*

```
1: $t$ := current time;
2: for all jobs in $queue$ do
3:     $w_j := Aging(j,t) + Deadline(j,t) + WaitMinimization(j)$;
4: end for
5: sort $queue$ according to $w_j$ in an increasing order;
6: execute $EASY\ Backfilling$;
```

Flexible Backfilling is an example of *dynamic* as well as *composite* scheduling policy [138, 137]. For example, for given job j, *Aging* and *Deadline* heuristics deliver different results as the time t proceeds, i.e., the behavior of Flexible Backfilling is *dynamic* as was discussed in Section 3.2.1. Moreover, the job priority w_j is computed using three different heuristics focusing on different criteria, therefore Flexible Backfilling represents a *composite* policy which aims to satisfy several optimization criteria. In this case, the use of dynamic priority w_j helps to improve wait time, slowdown and the number of jobs that meet their deadline, while the application of backfilling approach helps to improve especially the machine usage.

3.3.4 Algorithms Operating Multiple Queues

All previous algorithms operated over a single incoming queue. In the real systems, there are usually multiple queues that are designated for specific jobs or users. For example, there may exist different queues for short, normal or (very) long jobs. Also, some queues are designated only for selected users, usually belonging to some specific group. Scheduling algorithms applied in such an environment must be able to operate over several queues. Typically, the scheduling algorithm chooses the queues in a circular fashion based on the queue priorities. Once the queue is selected, similar algorithms to those presented in Sections 3.3.1, 3.3.2 or 3.3.3 are applied. There is no “general” description of how such multi-queue solution looks like, since there are usually many possible setups available. Each system typically uses specific setup which highly depends on the system purpose and requirements of its owners and users. Therefore, Algorithm 3.7 represents an example of a multi-queue scheduling algorithm, that implements main features of the currently used scheduling algorithm of the PBS Pro resource management system, which is used in the Czech national Grid infrastructure *MetaCentrum* [127]. We designate Algorithm 3.7 as the *PBS-like* algorithm.

Algorithm 3.7 *PBS-like algorithm*

```
1: $queues := \{queue_1, queue_2, .., queue_f\}$;
2: if $queues$ not sorted or their priorities have changed then
3:   sort $queues$ according to their priorities $qw_i$ in an increasing order;
4: end if
5: for $i := 1$ to $f$ do
6:   $queue_i := queues[i]$ ;
7:   for all jobs in $queue_i$ do
8:     $o_j :=$ owner of job $j$;
9:     $w_j :=$ calculate total user squashed area $TUSA_{o_j}$;
10:  end for
11:  sort $queue_i$ according to $w_j$ in an increasing order;
12:  for all jobs in $queue_i$ do
13:    if job $j$ can be executed then
14:      $k :=$ select cluster using FF policy;
15:      remove $j$ from $queue_i$ and send it on $k$;
16:    end if
17:  end for
18: end for
```

There are f *queues* in the system, each having a priority qw_i which is fixed and set by the system administrator. The queues are ordered according to their priority[11] and in each run all queues are subsequently selected (line 6). Once some queue is selected, all jobs within this queue are ordered according to the total user squashed area $TUSA_{o_j}$ (see Section 2.6) in an increasing order. This ordering puts higher priority to less active users who did not utilize the system very much so far. Once the queue is ordered, all jobs are tested whether they can start their execution. If so, such jobs are immediately sent on the available resource. No reservations are being made for jobs that are not able to start at the given moment. Therefore, this algorithm is a form of backfilling approach, where no job gets a reservation if it cannot currently start. This is a favorable issue since this algorithm — in contrast to other backfilling algorithms — requires no estimates of expected job processing time (ep_j).

3.3.5 Job Processing Time Estimates

Let us briefly describe the assumptions that can be made concerning the job processing time estimates. In the real Grid, the job processing time estimates are usually very inaccurate. While for some type of applications the processing time is quite predictable (image processing, video processing), sometimes the estimate cannot be precise since the user has no chance to make it correct. Typically, if the job's task is to perform some type of optimization where a large space of variables is being explored, it is often impossible to predict whether results can be obtained in few seconds, minutes or even days or weeks.

[11]Unless the queue priorities are changing in time, this is done only once as can be seen in lines 2–4.

Moreover, there can be some type of error in the job's source code which will result in just a few seconds of execution while normally such job would last much longer. Also, the user may not specify the estimate at all.

Although the processing time estimates are not strictly required, still most current queue-based systems favor users who are able to provide such information. Their jobs can be used for backfilling which leads to smaller wait time. If no user estimate is provided, the *queue time limit* can be used instead, since nowadays systems often use more queues that specify the maximum processing time of the job. Sadly, this substitution will cause that only few values will be used as estimates. On the other hand, similar situation often applies for users' estimates. It has been shown that users tend to repeatedly use the same values (e.g., five minutes, one hour, and so on), reusing popular estimate values for their jobs, while the most popular estimate is usually the maximal time limit of the given queue [178].

In current systems, the processing time estimate or queue time limit is used as a maximum time limit that a job can execute. In case that the actual job processing time is longer than the estimate, the job is killed [131, 113, 178, 91] since it exceeded its available runtime. This is essential to *ensure that reservations are respected* [178]. Therefore, users usually *overestimate* the job processing time to avoid killing of their jobs [131, 113].

3.3.6 Queue-based Approach and the Dynamic Environment

So far, we have introduced description of several common queue-based scheduling algorithms. Still, we have not discussed how these solutions deal with dynamically changing environment where processing time estimates are inaccurate or dynamic features such as machine failures appear. The situation is trivial when algorithms do not use reservations at all, i.e, when FCFS, EDF or PBS-like algorithm is applied. None of these scheduling policies have to reflect its previous decisions (no existing reservations), performing straightforward decisions according to the current situation. As soon as processing time estimates are used to establish reservations (EASY, Conservative and Flexible Backfilling) problems arise when such estimates are inaccurate and/or machine failures appear.

Since the reservations are established upon jobs submissions using their estimated processing times, inaccuracy causes that jobs can usually start earlier [52, 131] than their reservations allow[12]. For EASY and Flexible Backfilling the solution is quite straightforward since there is always at most one job with a reservation. Therefore, when earlier start time than guaranteed by the reservation is available it is accepted, existing reservation is canceled and a new one (earlier) is created.

The situation is different for Conservative Backfilling, since there are typically several reservations, one for each currently waiting job. The first solution is to leave existing reservations intact disregarding appearing earlier time slots. Sadly, such solution is very inefficient. Another solution is to cancel all existing reservations and re-backfill all jobs according to a new situation and establish new reservations [131]. However, this may

[12]It is a result of the overestimated processing time, which is a typical case in nowadays systems as we have discussed in Section 3.3.5.

violate the system's execution guarantees that were provided by the previous — now canceled — reservations [131]. Backfilling allows later jobs to skip over jobs that arrived earlier, provided a suitable gap is found in the reservations' schedule. This can only happen when the available gap is smaller than the size of some earlier job, but large enough for the later job. However, if a new round of backfilling is done later, thanks to an earlier termination, these gaps may become large enough for the earlier job so it will get reservation there. Later jobs now may not be backfilled as previously and will therefore run much later than was the guaranteed time of previous reservation [131]. To avoid such situations, another approach is taken where existing reservations' schedule is compressed [131]. Existing reservations are checked one by one starting with the earliest (nearest) reservation and they are inserted at the earliest possible start time [131]. Still, gaps may remain in such compressed schedule. However, these provide new backfilling opportunities, that can be exploited in the future by newly arrived jobs [52]. An example of such compression is shown in Figure 3.3.

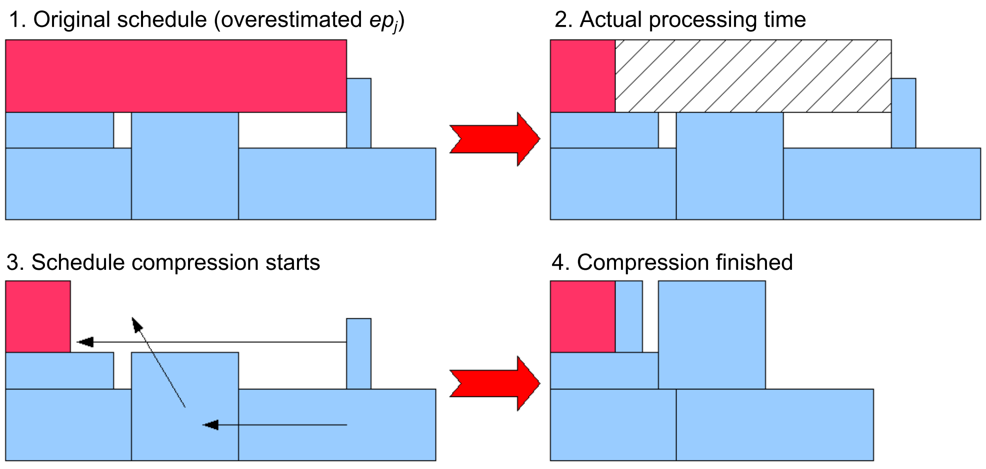

1. Original schedule (overestimated ep_j)
2. Actual processing time
3. Schedule compression starts
4. Compression finished

Figure 3.3: Compression of reservations' schedule after an early job completion.

Once machine failures appear, some jobs reservations may become invalid since the target machine(s) are down. EASY and Flexible Backfilling simply find a new valid reservation, while Conservative Backfilling cancels all invalid jobs reservations and then creates new ones for all such jobs. Valid reservations are left intact, and a partial re-backfilling has to be performed.

3.3.7 Related Approaches

So far we have closely described several queue-based scheduling algorithms. FCFS, EASY Backfilling, Conservative Backfilling or EDF represent typical solutions that can be found in production systems. Flexible Backfilling is an interesting approach focusing on the problem involving machine usage and job deadlines while PBS-like algorithm represents real life-based solution involving multiple queues.

Of course, there are dozens other scheduling algorithms that either combine or build on the top of previously mentioned techniques or even propose brand new approaches. There exist several modifications of the general backfilling approach. The Maui scheduler allows the system administrator to set the maximal number of reservations [86]. In [160] a compromise strategy called *Selective Backfilling* was proposed where jobs do not get a reservation until their expected slowdown exceeds some threshold. If the threshold is chosen properly, only the most needy jobs get a reservation. Additional variants of backfilling allow more flexibility. *Slack-based Backfilling* [168] supports priorities that are used to assign each waiting job a slack, which determines how long it may wait before the start of execution. Important jobs will have little slacks in comparison with the others. More examples of various backfilling implementations can be found in [154, 172]. Another example of combined approach is so called *Convergent scheduling* that has been introduced in [114] and then adopted in [27]. Convergent Scheduling exploits a set of heuristics that guide the scheduler in making decisions. Each heuristic manages a specific problem constraint, and contributes to compute a value that measures the degree of matching between a job and a machine. In the given example the scheduler optimizes problem involving job deadlines, resource utilization and efficient usage of software licenses. In order to take decisions, the scheduler assigns priorities to all the jobs in the system, which are then used to determine a final job-machine matching.

New approaches can be represented, e.g., by so called *pilot-based* scheduling. This new interesting approach has been applied within the Atlas project. Here a pilot based *glideinWMS Workload Management System* was used [152]. The main idea behind this system is that instead of real user's job a set of "pilot jobs" is broadcasted into underlying queueing systems [134]. These pilot jobs behave like probes that are used to find suitable machines for real jobs. Once the pilot job is scheduled onto some machine, it checks whether specified job requirements are satisfied by the machine. If it is true then the pilot job sends a signal to the actual job which replaces it and starts its execution. Moreover pilot job can discover and publish machine characteristics such as OS version, CPU model, available RAM and disk space or the availability of certain software in the real time [152]. This approach allows to minimize job failures, because the real job is executed only if the target machine satisfies all requirements. Once the actual job starts its execution at the machine, remaining pilot jobs can be used for remaining user's jobs. If there is no such job then all pilot jobs can be removed from the queues or they exit immediately since no actual job is waiting [134]. Last but not least, users can flexibly plan their jobs according to the up to date information delivered by the pilot jobs.

Similar idea has been applied in the *DIANE (DIstributed ANalysis Environment)* tool developed in CERN [129, 130]. DIANE uses so called *master-worker architecture* where the "pilot job" is called a *worker*, while the *master* is responsible for mapping the tasks onto workers. DIANE has been used in several projects and systems including Geant4, EGEE (EGI-InSPIRE) [130] or MetaCentrum [127].

3.4 Schedule-based Approaches

In this section we present the concept of schedule-based approach and show how it differentiates from the queue-based approaches. While many production scheduling systems use queue(s) to store waiting jobs, *schedule-based systems* use schedule as a more complex data structure that *maps jobs onto available machines in time* [2, 79]. The *schedule* represents de facto a plan of future job execution. In the same fashion as EASY or Conservative Backfilling require information about expected job runtime (ep_j), also the schedule-based solutions need such information to construct the schedule. A scheme of the general schedule-based system is shown in Figure 3.4.

Clearly, there are no incoming queues that would store the jobs [79]. Instead of that a two dimensional data structure is being build that assigns each job its own space and time slot. The x-axis represents system time and the y-axis represents particular machine on a particular cluster. There are other aspects that differentiate the design of schedule-based systems from the queue-based systems. The major difference is that nontrivial scheduling decision must be taken every time some new job arrives. Such immediate decision is necessary to find a suitable place for the job in the schedule. Queue-based methods usually perform such decision when the job is selected for execution, i.e., at the "last possible moment" (when machine(s) become available). An exception represent those queue-based methods that use reservations. Especially Conservative Backfilling creates a plan of reservations that stores informations about reserved time slots for every waiting job. Such plan is de facto identical to the data structure build by the schedule-based methods. When machine(s) become available, scheduling decisions are trivial for schedule-based methods. At that point in time, scheduler simply sends on such machine(s) those jobs that are stored in the schedule on given coordinates (i.e., time and machine

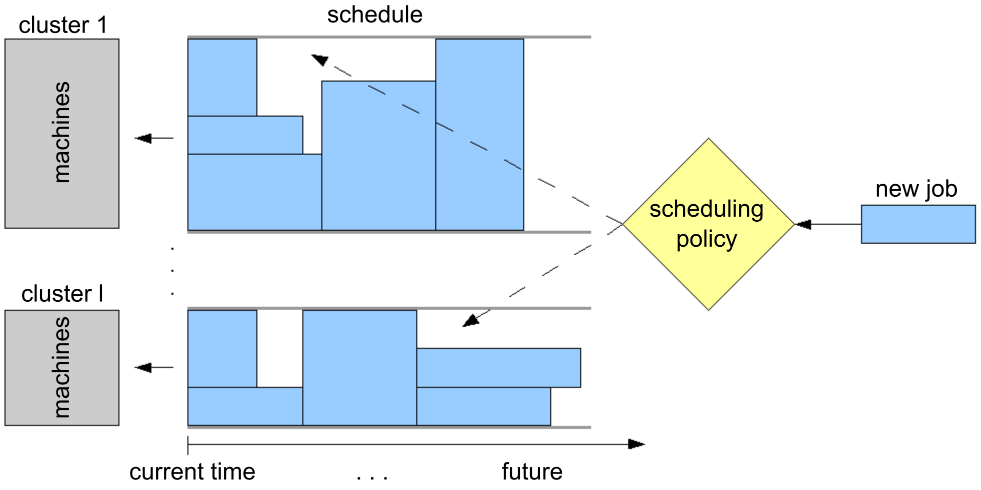


Figure 3.4: Schedule-based system.

coordinates)[13].

The use of schedule offers three major advantages. First, the schedule allows to *make predictions*, i.e., to guarantee for each job its start time, expected completion time and the target machine(s). Such information can be very valuable to the users since they can use it to better organize and plan their work. Second, since the schedule holds detailed information about expected execution of each job, it is possible to *evaluate the quality of the solution* which is represented by such schedule. Here several objective criteria described in Section 2.6 can be used. The evaluation allows to identify possible problems concerning job performance, efficiency of resource utilization or, e.g., fairness issues. Next, an *optimization procedure* can be launched that tries to improve the quality of the schedule with respect to the applied objective criteria.

In the following text we focus on the known works that use schedule-based approach to schedule jobs and we discuss three main issues related to the application of such an approach. Those are the *construction of the schedule* , the *optimization of existing schedule* and the application of schedule-based approach in the *dynamic environment.* As far as we know, schedule-based solutions are not applied in any major production system, therefore following text summarizes known techniques and approaches that have been used either in the literature [1, 187, 2, 143, 125] or in the experimental systems [92, 79, 164].

3.4.1 Construction of the Schedule

If the schedule-based approach is used we must be able to construct the schedule. For this purpose, several techniques are discussed in the literature or are applied in the experimental systems. Let us start with the experimental *Computing Center Software (CCS)* scheduling system [92]. CCS relies on a backfill-like approach when creating the schedule [79]. For every new job the *earliest suitable gap* in the schedule is found and the job is placed to it. A suitable gap is a period of unused CPU time which is large enough to admit the placement of the new job. Pseudo-code of such an algorithm is shown in Algorithm 3.8. Clearly, for each new job j the earliest suitable gap is being found. The gap can be either in front or after already planned jobs [79]. The first case represent a backfill-like situation. Once the gap is found, the job is placed into it and the schedule is updated with respect to this new situation. If no suitable position is found, then the job is canceled since it cannot be executed[14].

CCS allows to use priorities that are based on the expected processing times of the jobs (ep_j). More precisely SPTF or LPTF (see Section 3.3.3) policies can be used to order the jobs *prior* the schedule is constructed via the backfill-like algorithm [92].

Global Optimising Resource Broker (GORBA) [164] is another experimental schedule-based system designed for scheduling sequential and parallel jobs as well as workflows. In this system, similar approach to CCS is used involving simple rules such as *Cheapest Resource First, Fastest Resource First*, or *Cheapest or Fastest Resource First*, that are

[13]Situation is more complicated when processing time estimates are inaccurate or machine failures appear. We will discuss these issues later in Section 3.4.3.

[14]Such situation can happen for several reasons. For example, specific job requirements are not supported on any cluster or the number of requested CPUs is higher than it is currently available in the system.

Algorithm 3.8 *CCS-like algorithm*

```
1: $j$ := newly arrived job;
2: $position$ := **null**;
3: $schedule$ := existing schedule;
4: $position$ := find earliest suitable gap in $schedule$;
5: **if** $position$ = **null then**
6:     cancel $j$ since no suitable $position$ has been found;
7: **else**
8:     $schedule$ := move $j$ into $schedule$ on $position$;
9: **end if**
```

selected according to the specific requirements of the job. The resource prices per hour and performance factors given by the administrator are used to calculate comparable resource prices [161].

In [1], the objectives are to minimize the makespan and flowtime (the sum of the completion times [2]), therefore schedules are generated either by *Shortest Job on the Fastest Resource (SJFR)* which minimizes flowtime or by *Longest Job on the Fastest Resource (LJFR)* which optimizes the makespan.

Several heuristics such as *Min-Min, Max-Min, Opportunistic Load Balancing (OLB), Minimum Execution Time (MET)* or *Minimum Completion Time (MCT)* were applied in [143] to create the initial schedule. These heuristics create the initial schedule using known job or/and machine parameters. Detailed description can be found in [125] and [143]. Also, some solutions start with a randomly generated initial schedule, that is then immediately optimized [2, 143, 125].

While many previously mentioned works apply similar approaches for schedule construction, differences appear concerning techniques used to optimize such schedules. Also, these approaches employ different techniques to handle the dynamic behavior of the system. We closely discuss these issues in Sections 3.4.2 and 3.4.3 respectively.

3.4.2 Optimization of the Schedule

Once the schedule-based approach is adopted the application of optimization procedure to improve the quality of initial schedule is natural. Several metaheuristics have been frequently applied to perform such optimization and we present them here.

Concerning the experimental systems, CCS [92] applies no optimization procedure. On the other hand, GORBA [164] uses so called *Hybrid General Learning and Evolutionary Algorithm and Method (HyGLEAM)* optimization procedure, combining local search with the GLEAM algorithm [161, 87] which is based on the principles of evolutionary [126] and genetic algorithms. The detailed description of the complex HyGLEAM solution can be found in [87].

In [13], a graph-based mapping algorithm, called *Heterogeneous Multi-phase Mapping (HMM)* has been proposed to optimize the mapping of both parallel jobs and DAG-like workflows onto the heterogeneous computing nodes of the Grid. The algorithm is structured according to two phases. In the first phase, an initial mapping solution is carried out. In the second phase, Tabu Search [69] algorithm is applied to the atomic nodes in the critical path of the DAG, to improve the quality of the initial solution. Provided experiments showed that HMM performs better than or in the same way as other six mapping algorithms, and that it obtains good results when mapping two large real applications on the heterogeneous resources composed of many processors. Comparisons with an exhaustive mapping algorithm showed that HMM always carried out mappings which were very close to the optimal solution.

Two metaheuristics designed for Grid load balancing are used in [162]. Here *Genetic Algorithm (GA)* [141, 137] and *Tabu Search (TS)* are used to solve the load balancing problem. The effectiveness of each algorithm is shown for a number of test problems, especially when prediction information concerning background machine load is not fully accurate. Performance comparisons showed that both GA and TS outperform known approaches such as Best Fit, Random, Min-Min, Max-Min, and Sufferage [162].

Application of different local search-based algorithms have been proposed in [1]. *Simulated Annealing (SA)* [137, 11], *GA* and *TS* including their hybrid versions namely hybrid *Genetic-Simulated Annealing (GA-SA)* and hybrid *Genetic-Tabu Search (GA-TS)* are proposed. In GA-SA, simulated annealing is used to accept or reject new mutations of previous solutions while in the hybrid GA-TS approach, reproduction, crossover and mutation in GA is replaced by reproduction, crossover and Tabu search. Instead of random mutation changes, each member of the population undergoes a separate optimization process performed by a Tabu search algorithm [1]. Unfortunately, this paper only presents the proposal of aforementioned algorithms but no experimental evaluation is presented to support the given ideas.

Other approaches can be found in [187]. Nice survey of recent applications of metaheuristics for Grid scheduling problems can be found in [188, 186].

3.4.3 Schedule-based Approach and the Dynamic Environment

So far we have shown how the schedule is build or optimized but we did not provide any information on how the existing implementations and proposals deal with the dynamic changes such as new job arrivals, the inaccuracy of processing time estimates or the machine failures. Surprisingly, many works ignore these real life features, thus representing very simplified static problems which are far from reality [2, 13, 143, 10, 4].

In [10, 13], authors assume that all the jobs and resources are known in advance and their number is not changing, which allows to create the schedule for all jobs at once. Similar situation applies for [4]. Authors of [2] and [143] also consider only static problem. Moreover, only unary resources (1 CPU) and sequential (non-parallel) jobs are used in the problem.

The authors of [28] solve similar optimization problem but allow dynamic job arrivals and resource changes. The dynamics is handled by periodical rescheduling procedure that involves genetic algorithm. Regrettably, runtime issues of the rescheduling procedure with respect to the problem size are not discussed in the paper.

In [1] and [162] local search based methods not requiring total re-computation upon dynamic events are proposed to solve Grid scheduling problems. Dynamic job arrivals are supported in [1], however the efficiency of the proposed solution involving SA, GA, TS, GA-SA and GA-TS cannot be evaluated, since the paper does not include experimental comparison as we have already mentioned in Section 3.4.2. Similarly, authors of [162] allow to simulate dynamic changes in the background load of the computing nodes but the number of jobs is always static in their experiments. The inaccuracy of information concerning changing background machine load is maintained via periodical optimization that can improve the quality of the schedule. As the optimization periods decreases, which results in more scheduling runs, the performance gains from the application of GA and TS increase [162].

As far as we know, only the CCS and GORBA cover all dynamic features that are typical for Grid-like environments. Both experimental systems recompute the schedule from scratch when a dynamic change such as job arrival or machine failure appears. In CCS, each time a new job is submitted or a running job ends before it was estimated, a new schedule has to be computed. CCS allows its users to use so called *advanced reservation*. These reservations are specified manually by users. When an advanced reservation is requested and confirmed, it means that a check was made to see if the reservation would conflict with currently running jobs or other confirmed reservations. A reservation request that fails this check is denied by the scheduler [88]. In CCS, advanced reservations are kept during the rescheduling phase. Therefore, when rescheduling is performed in CCS, all non-reservations are deleted from the schedule and sorted according to the applied rule. Then they are reinserted in the schedule at the earliest possible start time. Therefore, the rescheduling process is based on total replacement of the previous schedule excluding previously accepted reservations [79].

In GORBA, the rescheduling is triggered either when new jobs arrive, some waiting job is canceled or changes in the resource pool occur. When the actual processing time differs from the planned time above a given threshold, the rescheduling is triggered as well [161]. Although it helps to keep the schedule up to date, this approach may be quite time consuming for large number of jobs as was observed and discussed in [161].

3.4.4 Discussion

In this section we have described the main characteristics of schedule-based methods. We have shown the main differences of this approach with respect to the popular queue-based schema. Since queue-based methods are widely used in production systems it is well documented what techniques should be used to deal with common dynamic features of the Grid environment (see Section 3.3.6). Moreover, many applied queue-based scheduling algorithms use no or very limited planning functionality (reservations). Therefore, dynamic changes concerning jobs or machines can be often handled very easily, preserving good speed and scalability that is necessary as discussed in Section 2.7.2.

The situation is different for schedule-based methods since planning is always used and the schedule is maintained over the time. Therefore, dynamic changes must be reasonably reflected in the existing schedule to keep it up to date and consistent. However, as far as we know the existing works usually applied two different solutions. They either ignore these features and use a simplified problem or they perform total re-computing of existing solution to handle the dynamic features. None of these two approaches is very good. Simplified problem does not allow modeling of realistic scenarios while (frequent) re-computations represent potential problem concerning speed and scalability as was shown in case of GORBA [161] as well as in our own research [104].

3.5 Job Processing Time Prediction

Many current scheduling techniques such as LPTF, SPTF, backfilling or the schedule-based approach use the estimates of the job processing time. The more precise these estimates are, the better results can be expected from these techniques [32, 67, 157, 179]. Moreover, these estimates can be used to predict queue wait times, which is useful for guiding resource selection when several systems are available, to co-allocate resources from multiple systems, to schedule other activities, and so forth.

In current systems, the estimates are usually specified either by the user or by the predefined time limit associated with the queue (see Section 2.4). Sadly, such estimates are typically very inaccurate and overestimated [131, 113, 160, 32, 34]. Authors of [113] have shown that even when users are motivated to improve the accuracy of their estimates — with the assurance that their jobs will not be killed after that "smaller" amount of time has elapsed — the accuracy of their new estimates was, on average, only slightly better than their original estimates [113].

In such situation, it is not surprising that several automated techniques have been proposed to establish more precise estimates without an user interaction. These techniques can be divided into several categories according to the applied estimation technique. The technique proposed in [44] uses repeated executions of the job to establish the estimate while other solutions work on the basis of compile-time analysis [150, 12] or using a historical information together with the statistical analysis [158] of previously executed jobs. Such approach has been applied in [157] where the authors use a template-based approach to categorize and then predict job execution times. This approach is based on the observation that similar applications are more likely to have similar run times than applications that have nothing in common. The similarity is based on several parameters such as the type of the job, the owner of the job owner, the requested number of CPUs, etc. According to this information the jobs are divided into categories and the runtime estimation is computed using historical data [157]. Using these predictions, the mean queue delay predictions are derived by simulating the future behavior of the scheduler [158]. Although the proposed model fits the execution-time data well [135], the mean error ranges from 34% to 77% [158].

In [47, 48] a similar set of assumptions for estimating queue wait times is used. In this works, a uniform-log distribution is used to model the remaining lifetimes of jobs currently executing to predict when required machine(s) will become available and thus when the job waiting at the head of the queue will start. The estimator calculates the wait time for the first job only, it does not address the question of how long it will take to get there [47].

However, these approaches [47, 48, 158] make the underlying assumption that the scheduler is employing a fairly straightforward scheduling algorithm such as FCFS [47, 48, 158] or Conservative Backfilling [158] as well as that the system is static for the duration of their experiments, i.e., no failures, restarts or administrator interference appear.

Queue Bounds Estimation from Time Series (QBETS) is an example of fully automatic method for predicting bounds (with specific levels of certainty) on the amount of queue delay each individual job will experience [135]. It uses more realistic approach than previous techniques [157, 158, 47, 48] since it does not need to create a job execution model to predict the amount of time those jobs will wait. Instead of that, QBETS make job wait time inference from the actual job wait time data itself. This is desirable since modeling job execution time may be difficult for large-scale production computing centers [73, 135]. Further, no underlying assumptions about scheduler's algorithms or machine stability must be made which is more realistic approach since actual scheduling algorithms involving all specific setups are rarely published and are not typically simple enough to be modeled with a straightforward procedure [135]. Unlike the previous approaches [157, 158, 47, 48] where expected (mean) wait time for a particular job is returned, QBETS uses bounds on the time an individual job will wait rather than a specific, single-valued prediction of its waiting time. Authors argue that such quantified confidence bounds are more "user friendly", since users can know the probability that their job will fall outside the range. The plea here is that the user prefers to know that, for example, there is a 75% chance that the job will execute within 15 minutes, rather than knowing that the expected wait time for a particular job is 3 hours [135]. In the experimental evaluation, QBETS was shown to be more accurate than any other tested technique and prediction method [135].

Similar prediction techniques can be integrated into the production systems such as *Network Weather Service (NWS)* [184] or *MonALISA* [171]. NWS is a real life operational system responsible for monitoring of the Grid resources such as network links and machines and it is also able to dynamically characterize and provide short-term performance forecasts based on historical performance measurements. The goal of NWS is to forecast the performance deliverable to the application from a set of network and computational resources. NWS currently integrates QBETS method for forecasting queue delays[15].

MonALISA is a high level monitoring service developed for the STAR Unified Meta-Scheduler (SUMS). MonALISA is designed to easily integrate existing monitoring tools and procedures and to provide this information in a dynamic, self describing way to any other services or clients. It supports queue monitoring for most popular systems such as PBS Pro, LSF, Condor and SGE. It collects information containing queue length, worst response time, estimated response time, number of free CPUs and the number of running jobs. Therefore, systems like NWS or MonALISA may be used to provide necessary input data required by the aforementioned prediction techniques.

3.5.1 Summary

To summarize this section, there exist several techniques that can be used to estimate job wait/processing time before actual execution. As far as we know, all techniques are

[15]NWS with QBETS runs at: `http://nws.cs.ucsb.edu/ewiki/nws.php?id=Batch+Queue+Prediction`

primarily designed for queue-based systems and their application in schedule-based systems is not always easy. For example, QBETS represents flexible real life system that is useful for users of queue-based systems, calculating them the probability that their job will execute within specified time limit. However, when constructing the schedule, a single value representing the estimated processing time is more useful. From this point of view, the application of QBETS would not be straightforward. In this case, techniques such as [157, 158] make more sense since they are designed to provide such an estimate, using historic data of similar jobs.

Still, the actual precision highly depends on the type of jobs that are executed in the given system. For example, an experimental prediction system recently developed by one of the students of Faculty of Informatics [90] is using analysis of historical data similar to those presented in [135] and [157, 158]. Still, the resulting precision is often very poor. For example, for 50% of jobs the difference between the calculated estimate and the real processing time is at least 61%. Moreover, for 20% of jobs the difference is higher than 2,377%! Another problem is that these techniques may highly *underestimate* the actual precessing time of jobs [90] which may have bad effects for both schedule and queue-based algorithms. From this point of view, it is better if processing time is overestimated since guaranteed start times remain valid. When the processing time of the job is overestimated, the job finishes earlier and a gap appears but all start times remain guaranteed. On the other hand, when the processing time is underestimated, the job has to be killed, otherwise its execution may easily collide with existing reservations of different jobs.

Chapter 4

Proposed Approach

In this chapter, we describe the proposed application of the schedule-based approach, developed for solving the studied job scheduling problem defined in Chapter 2. Currently used production Grid schedulers are all based on queueing systems [88, 86, 7, 139]. As was already shown in Section 3.3 such systems usually utilize scheduling policies that all work in an ad-hoc fashion. It means that scheduling decisions are typically made for one job at a time according to the current state of the system. The order in which jobs are selected is usually fixed according to the job arrival time or job priority. Typically, when the given job is being scheduled, remaining jobs in the queue and their requirements are not considered by the scheduling policy. Such a scenario may result in inefficient schedules that cause bad performance for some jobs [131, 52, 154, 102].

Some solutions such as Conservative Backfilling focus on these issues through job reservations that allow better predictability [131, 52]. Still, reservations are established in given job order and are not modified once created[1]. In another words, although the reservations allow to plan ahead, without *evaluation* and *optimization* they still represent an ad-hoc solution where decisions are fixed once created and do not change even when it is clear that such schedule is inefficient. Therefore, Conservative Backfilling represents only the first step toward schedule-based approach, allowing to make predictions concerning future system's behavior as was discussed in Section 3.4. It uses neither evaluation nor optimization, therefore the benefits of reservations are not fully utilized.

On the other hand, widely used queue-based systems are quite *tolerable to the dynamic Grid environment* where various changes such as job arrivals or machine failures appear through time. When no planning is used (no job reservations), queue-based approach is very tolerable to such dynamic features since the scheduling decisions are always performed according to the *current state* of the Grid and no or very limited consideration of previous decisions is used during the scheduling process [95]. Even when some form of planning is applied (reservations), techniques like those presented in Section 3.3.6 can be used to keep the solution up to date.

Our goal is to apply schedule-based approach in an efficient manner. Unlike some existing works such as [10, 13, 2, 143, 10, 4], our goal is to propose *realistic solution*, i.e.,

[1]Detailed discussion has been given in Section 3.3.2 and 3.3.6

solution that would solve the complete problem defined in Chapter 2. It means that it should be able to deal with the *dynamic behavior of the system* and *uncertainty*, that includes dynamic job arrivals, specific job requirements concerning machine parameters, machine failures and restarts as well as inaccurate processing time estimates. Moreover, its runtime requirements should be low and it should scale well with respect to the problem size.

Moreover, we want to avoid problems related to the “ad-hoc” application of scheduling policies. The main idea is to accompany the application of schedule with the *evaluation* and the *optimization*. Using the evaluation we aim to analyze the quality of generated schedule using proper objective functions that represent both users’ and administrators’ requirements as we have defined in Section 2.6. Once the evaluation is used, it can guide the solution techniques toward construction of feasible and efficient schedules. For example, the decisions delivered by scheduling policies can be analyzed or an optimization procedure as a method to improve the quality of generated schedule can be used [104, 102]. We propose the use of simple and fast scheduling policies to build the initial schedule over the time and a local search based optimization procedure that would help to improve the quality of the initial solution [104, 105].

This chapter describes the proposed solution in detail, introducing all techniques proposed to achieve discussed goals. The chapter is organized as follows. We start with a description of the so called *event-based approach* that is used to correctly maintain dynamic changes appearing in the system through the time. Next, an *incremental approach* is described, whose main principle is to reuse existing (previous) solution, thus keeping the necessary runtime of scheduling algorithms in an acceptable level. We also present so called *gap filling approach* that aims to maximize schedule’s efficiency while following the incremental approach as well. It is used both during schedule construction and optimization. Finally, the principle of *anytime approach* is described that allows flexible application of optimization algorithm. Next, we present several use cases that show how the proposed approaches are used in various situations. Also, techniques proposed to solve the problem of *inaccurate processing time estimates* are presented here.

The chapter continues with a description of applied data representation of the schedule itself. Using it, implementations of the proposed scheduling techniques can be properly described. Those are *policies* used for construction of initial schedules and the *optimization algorithms* that are used to further optimize the quality of initial schedules delivered by the policies. Beside the scheduling algorithms, also the implementations of several useful methods that maintain or operate over the schedule are discussed here. Moreover, the analysis of the *computational complexity* with respect to the applied data structures is presented for each such method or scheduling algorithm.

4.1 Principles

The proposed solution uses several ideas and principles which we present in this section. It is based on the application of two scheduling techniques to deliver the solution [104]. It uses a *scheduling policy* [68, 137] to generate initial schedule and an *optimization algorithm* that can be used to improve the quality of such initial solution. The scheduling policy is typically used upon each new job arrival and it uses similar principles as techniques described in Sections 3.2.1 and 3.3.2. It finds a suitable position for the new job in the existing schedule. When the policy finishes, the job is moved into the schedule on a position found by the policy. Such schedule is called *initial solution.* It can be further improved by the optimization algorithm. We use *metaheuristics* (see Section 3.2.3), namely *local search*-based optimization algorithms [104, 103, 138, 137], since similar techniques have been successfully applied in several areas including the Grids [1, 162, 2, 143, 187]. It involves a *search procedure* that modifies the initial solution by changing the positions of jobs in the schedule. These so called *moves* are either accepted or rejected subject to the result of the *evaluation procedure* that covers chosen objective functions.

Before we proceed to the description of algorithmic details of these two techniques, let us describe four important features of the proposed solution. Those are *event-based, incremental, gap filling* and *anytime approach.* Various issues arise when starting to apply schedule-based approach in Grids. First of all, the schedule has been usually used for problems with static or predictable behavior. This means that various parameters such as number of machines and their performance, number of jobs and especially job processing time were all known in advance, allowing to create the schedule en bloc. Unfortunately in the Grids none of the above parameters remains stable or is precisely known in advance. The Grid is *fully dynamic environment* as we characterized in Section 2. Therefore, the most important principle is that scheduling process must properly react on the dynamic changes of the system.

4.1.1 Event-based Approach

Appropriate reactions of the scheduling process on the dynamically changing state of the system are implemented by the application of so called *event-based approach.* It helps us to deal with the dynamic environment [104, 105, 95], and it works in the following fashion. At any time, the scheduler maintains a schedule which is created according to the known state of the Grid, covering both resources and jobs. Such state is only temporal and can change at any time. The change occurs in the form of an *event.* Each such event is analyzed since it signals new state of the Grid and proper reactions are immediately taken. For simplicity, we assume that the events are delivered to the scheduler through some reliable service, i.e., all events are considered as correct and up to date.

For the purposes of this work we define typical events that allow us to model the problem presented in Chapter 2. First of all *job arrival* event signals that new job has arrived from some user. *Job completion* is used to inform the scheduler that some job finished its execution. Job completion may be either *early* or *on time.* Other events are related to the current state of the computational resources. Events such as *machine failure* and *machine restart* signal the change in the pool of available resources. If a machine failure appears, all jobs being executed there are immediately killed. This situation is represented with a *failed job* event.

All preceding events are *external events* which are delivered to the scheduler "from the outside". However, we also use *internal events* that are generated by the scheduler itself to guide the scheduling process. *Perform optimization* signals that optimization of current solution should start. *Periodic optimization* event is used to cyclically optimize the schedule. Finally, *schedule jobs* event is used to inform the scheduler that it is time to test whether some jobs can be moved from the schedule onto available (free) resources.

Event-based approach represents general framework that is used to control the scheduling process, i.e., using events we are able to take proper actions and keep the solution efficient and up to date. All important events and related scheduler's reactions will be closely discussed in Section 4.2, using a scheme shown in Figure 4.1.

4.1.2 Incremental Approach

In our previous works [104, 97] as well in works of other authors [161], we have encountered that schedule-based approach is often very time consuming when large problems are solved statically, i.e., the whole schedule is computed at one moment. That is why we apply *incremental approach* [33] which reuses existing solution that is built *incrementally* over the time. This principle has been applied for both the scheduling policy and the optimization algorithm. First we describe the application of this principle in case of the scheduling policy. Every time a new job arrives an existing schedule is used as initial solution and the scheduling policy is used to place the new job within existing schedule using some applied strategy[2]. If multiple assignments are available, the evaluation procedure is used to determine the best candidate. This application of scheduling policy is reasonably fast since it reuses previously computed schedule and keeps the number of changes low. The reuse of existing solution saves a lot of computational time and allows to come with a new solution in a shorter time [104].

Moreover, scheduling policy allows not only to compute the new solution faster but we have designed it to create a valid, executable schedule. This means that jobs can be executed according to the schedule created with such policy because all constraints related to the job-machine mapping are met. For example if job j requires $usage_j$ CPUs for its execution then the policy guarantees that the new schedule will respect such constraint while not exceeding the available capacity of selected resource(s), i.e., the number of free CPUs. However, this initial solution may not respect some additional soft constraints, e.g., job deadline or the quality of this solution may not be very good.

Such situations can be fully or at least partially solved by further schedule optimization. It starts with the existing schedule and introduces *local changes* to optimize the solution with respect to the optimization criteria. Again, the optimization algorithm follows the idea of incremental approach by using existing solution as the initial solution. Here the difference between our incremental approach and existing schedule-based solutions such as CCS or GORBA can be seen. CCS as well as GORBA recomputes the schedule from scratch every time a new job is submitted or a running job ends before it was estimated [79, 161].

[2]The applied strategy may change for different policies. We will discuss these strategies in Section 4.6.

4.1.3 Gap Filling Approach

Both the scheduling policy and the optimization procedure use so called *gap filling approach* [104]. It represents an efficient method — inspired by the backfilling approach commonly used in the queue-based systems [154, 52, 159] — which detects and fills existing gaps in the schedule with suitable jobs. Not only that it improves response time, wait time, slowdown or e.g., machine usage[3], it also perfectly follows the proposed incremental approach, thus reducing computational time [104].

4.1.4 Anytime Approach

In the Grid environment the situation is changing dynamically due to uncertainty, errors, or simply due to the new job arrivals. Some of these events — typically arrival of the large set of jobs, high priority job arrival or machine failure — require prompt reactions to satisfy users and to adjust the schedule accordingly. Fortunately, we can achieve *anytime approach* delivering solutions at allowed time with the help of scheduling policy and optimization algorithm.

As we have explained, scheduling policy is sufficient to create executable schedule in a short time following the incremental approach. This allows to handle job arrivals very quickly, so that the optimization algorithm can be used to improve the quality of such initial solution. Similarly as in the case of scheduling policy, each accepted move of the optimization algorithm results in an executable schedule, i.e., all constraints related to the job-machine mapping are again satisfied. It allows to stop the optimization algorithm at *any time* [98] while still obtaining executable schedule. This is a very useful feature in the dynamic Grid environment because various events may force us to make prompt decisions [15].

4.2 Event-based Scheduling System

In the following text we demonstrate the application of aforementioned principles. We show how the proposed schedule-based approach can address the dynamic behavior of the Grid using the scheduling policy and the optimization algorithm. The proposed solution represents a scheduling system which is driven by the incoming events. It is designed as an endless loop. In each iteration, an incoming event is analyzed and proper actions are taken based on the event type. A number of reactions which are all driven by the previously described events will be presented and explained to show that the scheduling policy and optimization algorithm can be flexibly used to overcome the Grid dynamics while keeping the benefits of schedule-based approach in most cases.

Figure 4.1 describes the behavior of the proposed event-based scheduling system. There are three different event queues that store incoming events before these are properly handled. The *high priority* queue stores events with the highest priority. Those are *schedule*

[3]The improvement is based on the fact that such approach limits the fragmentation of the processor time [179, 154].

jobs, *machine failure* and *machine restart* events. These events must be handled immediately. Otherwise, the performance of the system may degrade or existing schedule may become invalid, causing incorrect scheduling decisions. Next, there is a *normal priority* queue that is used for normal events such as *job arrival*, *on time job completion*, *early job completion* or *failed job*. The *low priority* queue stores *perform optimization* and *periodical optimization* events.

Every time a high priority event is detected, following actions are taken. First of all, both normal and low priority queues are stopped, i.e., it is not allowed to start the processing of events waiting in these queues. Next, if the *low priority event handler* is currently handling some event it is terminated immediately. If some event, e.g., *job arrival* or *job completion*, is currently handled by the *normal priority event handler*, the system waits until it is completed[4]. As soon as both normal and low priority event handlers are stopped, the *high priority event handler* starts to take proper actions to handle incoming high priority event. Once it finishes, it checks whether some other high priority event is available. If so, it handles it. Otherwise, it unblocks the normal and low priority queues.

The handling of normal priority events is similar. First, the low priority queue is stopped and — if currently active — the low priority event handler is terminated immediately. Once the normal priority queue is empty and all related events have been handled, the low priority queue is unblocked and low priority events can be handled.

Except for the *schedule jobs* event, all successfully handled events launch a new internal event. The type of the new event depends on the type of the original event. In the following text we closely describe the actions taken by the event handlers according to the event type.

4.2.1 Job Arrival

Naturally, we start with the job arrival. A job arrival is always handled by a scheduling policy which is used to find the initial solution, i.e., to put the incoming job into a suitable place in the existing schedule. It uses the incremental approach so the existing schedule is reused in the new solution as much as possible. It helps to save a lot of computational time. In case that no cluster is suitable to execute the job, it is canceled immediately. Once the policy adds the new job into the schedule an internal high priority *schedule jobs* event is launched immediately (see Figure 4.1). It guarantees that if some resources are available all corresponding jobs from the schedule will be sent for execution immediately.

4.2.2 Job Execution

An attempt to start jobs' execution is performed when *schedule jobs* event is detected. In that case, all free machines are identified. Then, corresponding parts of the schedule are checked for jobs that are planned to use such machines. If a match is found, the job is removed from the schedule and it is immediately sent on that machine(s) where its execution starts. This action is repeated for all jobs that can start their execution at that time.

[4]Otherwise some jobs could get lost easily.

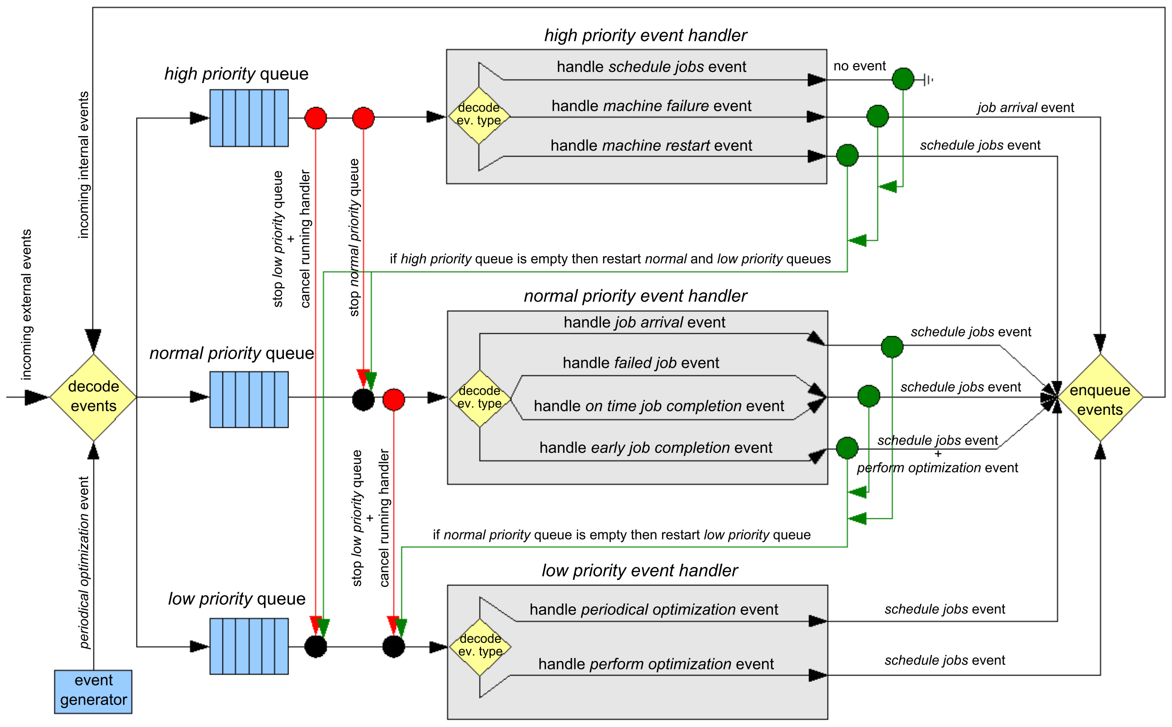

high priority event handler
high priority queue
handle schedule jobs event
handle machine failure event
handle machine restart event
decode ev. type
no event
job arrival event
schedule jobs event
incoming internal events
incoming external events
stop low priority queue + cancel running handler
stop normal priority queue
if high priority queue is empty then restart normal and low priority queues
normal priority event handler
normal priority queue
decode events
handle job arrival event
handle failed job event
handle on time job completion event
handle early job completion event
schedule jobs event
schedule jobs event
schedule jobs event + perform optimization event
enqueue events
stop low priority queue + cancel running handler
if normal priority queue is empty then restart low priority queue
low priority event handler
low priority queue
periodical optimization event
handle periodical optimization event
handle perform optimization event
schedule jobs event
schedule jobs event
event generator

Figure 4.1: The scheme of event handling.

In fact, *schedule jobs* event is launched every time the *state of the system changes*, to guarantee that jobs do not wait longer than necessary. More specifically, it is launched when new job has been added into the schedule, some job has finished its execution, some machine has been restarted or the schedule optimization routine has been performed. All these situations can potentially create opportunities to execute some waiting job. As we can see in Figure 4.1, no new event is created once this procedure ends, since there is no chance to schedule more jobs with respect to the current state of the system.

4.2.3 Machine Failure

In case that *machine failure* event appears, corresponding parts of the schedule are marked as currently unusable[5]. The exiting schedule becomes inconsistent if there are some jobs planned on that — now failed — machine. To remove the inconsistency, the schedule for such cluster is canceled and all corresponding jobs are resubmitted using the *job arrival* events. Next, all resubmitted jobs are placed into the schedule using same approach as in case of a normal job arrival. If some jobs were being executed on that failed machine, those are killed immediately and *failed job* events are delivered. Failed jobs are not resubmitted[6].

[5]Using the *machine failure* event we can also cover the situation when an old machine is permanently removed from the cluster.

[6]Automatically "restartable" jobs are not considered.

As soon as resubmitted jobs are properly handled, *schedule jobs* event is generated to start a new scheduling round using the new, up to date, schedule.

4.2.4 Machine Restart

In case of *machine restart*, corresponding part of the schedule is marked as usable[7]. By default, no further action such as job resubmission is taken to save computational time. Free machine will be soon used by newly arriving jobs. Alternatively, an optimization algorithm (*perform optimization* event) can be used to balance the load of the system. Restarted machine with empty schedule will be quickly identified by the optimization algorithm as a good candidate for jobs being moved from their former positions. As usual, *schedule jobs* event is generated in the end.

4.2.5 On Time Job Completion

When a job completes its execution as expected (on time), corresponding CPUs become available and a *on time job completion* event is delivered to the scheduler. Once received, an internal *schedule jobs* event is immediately launched so that next jobs from the schedule can start their execution.

4.2.6 Early Job Completion

As in the previous case, early job completion means that corresponding CPUs become available and an *early job completion* event is delivered to the scheduler. The difference is that now the scheduler knows that at least one gap has appeared in the current schedule. Therefore, the schedule is immediately compressed similarly as in the case of Conservative Backfilling (see Section 3.3.6). Next, *schedule jobs* event is launched to immediately schedule jobs on the resources. After that, an internal *perform optimization* event is created to start the optimization algorithm. It is used to fix the schedule in case that the schedule compression left some gaps — which happens frequently.

4.2.7 Schedule Optimization

Schedule optimization is launched either with *periodical optimization* or *perform optimization* events. In the first case, optimization is run periodically to improve the quality of the initial solution delivered by a scheduling policy. The optimization is carried out by an iterative local search-based algorithm that evaluates each move using the applied optimization criteria. It is launched with a *periodical optimization* event that is repeatedly generated with a given frequency, independently from other events. For this purpose an event generator is used as can be seen in Figure 4.1 (bottom left).

On the other hand, *perform optimization* event is typically used to optimize the schedule after an early job completion has been detected. The goal is to move jobs into gaps that remained in the schedule after the compression. To guarantee fast response of the system,

[7] *Machine restart* event can be used to simulate the addition of a new machine into the cluster.

optimization phase represents a *low priority* operation. It means that it is always interrupted when some new event such as job arrival, job completion, machine failure or restart is delivered to the system. Thanks to the applied *anytime approach* (see Section 4.1.4) the schedule remains consistent after the interruption.

This section has described how the proposed schedule-based solution is designed to react on common situations that are typical for the dynamic Grid environment. Still, the briefly mentioned problem of *inaccurate processing time estimates* represents a major issue when schedule-based approach is applied. Therefore, it will be closely analyzed in the following section.

4.3 Handling of Inaccurate Job Processing Time Estimates

The origin and importance of inaccuracy of job processing time estimates has already been discussed both for the queue-based (see Sections 3.3.5 and 3.3.6) and the schedule-based systems (Section 3.4.3). Also, an overview of techniques designed to refine such estimates has been presented in Section 3.5. This section describes the approaches we have proposed to handle this problem.

Standard applications of schedule-based methods such as manufacture scheduling [138], workforce scheduling [21] or, e.g., timetabling [136, 132] expect that the processing time is known precisely in advance. For several reasons, this assumption is very unrealistic in real Grids, as we have explained in Section 3.3.5. Therefore, realistic application of schedule-based solution must be able to deal with (very) inaccurate estimates. These estimates are either specified by the user or the queue time limit can be used instead, if no user estimate is provided. In case that neither users' estimates nor queue time limits are available, some automated techniques such as those based on historical and statistical information can be used [157, 135]. Also, synthetic estimates can be generated using a workload model such as [178, 177]. As we already explained, processing time estimates are usually *overestimated*, to prevent killing the jobs once they reach their predefined time limits. Still, even when the job is killed, from the scheduler's point of view such job's termination represents an *on time job completion* and does not require any "repair action".

As the expected processing time is generally overestimated — in most cases — the job finishes earlier than it is specified in the schedule, since the schedule is built using the estimates. Similarly as in the case of Conservative Backfilling (see Section 3.3.6), an early job completion creates a gap in the schedule [160]. In the following text we will describe how the proposed schedule-based solution deals with such situation.

Since the inaccuracy of processing time estimates is natural, our major effort was to propose an efficient solution able to deal with this situation. Similarly as our whole concept of schedule application, also this solution uses the event-based approach. The whole recovery process starts with an *early job completion* event which is detected by the scheduler once some job j finishes earlier than expected. It signals, that a gap has appeared in the current schedule. The size of the gap is equal to $usage_j$ and its duration is $ep_j - p_{i,j}$, i.e., the duration is the difference between estimated processing time and the actual processing time ($p_{i,j}$) on the machine i. In the first step, the schedule is immediately compressed, similarly as in the case of Conservative Backfilling. Implementation details of the applied compression algorithm will be given in Section 4.5.2, discussing the differences with respect to the compression algorithm applied in Conservative Backfilling. Schedule compression is relatively fast operation with respect to the total rescheduling. Once the schedule is compressed, *schedule jobs* event is launched to immediately schedule jobs on the free resources.

Although the schedule has been compressed it does not guarantee that all gaps have been successfully removed. These gaps represent unused CPU time that could be utilized by some “later” jobs from the schedule. Therefore, an internal *perform optimization* event is created to start the optimization algorithm. The goal of the optimization algorithm is to identify these gaps and find suitable jobs that can utilize them [102]. This represents the application of the general gap filling approach. Of course, the whole optimization process is subject to evaluation. Only those moves that improve the quality of the schedule are accepted [104]. The moves performed by the optimization algorithm represent local changes of the schedule which follow the proposed incremental approach. Also, the applied *anytime approach* guarantees that the optimization algorithm will leave the schedule in a consistent state if a higher priority event will stop the optimization procedure.

As we described above, the information provided by the user may be of very different quality. Using the principles of event-based approach that involves gap filling and anytime approach we have proposed incremental solution that fixes the schedule in case that processing time estimates are inaccurate. The rest of this chapter will focus on the implementation details of the proposed solution.

4.4 Applied Data Structures

This section describes the data structures that are used to represent jobs and the schedule. Using them, we can better describe the implementation details of proposed scheduling algorithms (see Sections 4.6 and 4.7). Moreover, it also allow us to discuss the computational complexity of the proposed solution.

4.4.1 Job

For each job unique object is created that has several parameters (internal variables) that fully characterize given job. These parameters cover all important properties of the job as has been defined in Sections 2.2 and 2.4, i.e., job id, owner id, arrival time, estimated processing time, specific job requirements, etc. Beside that, the object can also store additional information related to its *position in the schedule*. For example, it stores a list of assigned CPUs as defined by the schedule. Also the expected start time and the expected finish time are stored within the object. These schedule-related values are updated when the job is either added to the schedule or its position in the schedule has changed due to, e.g., the compression of the schedule or the optimization, so that the values remain consistent with the current schedule.

4.4.2 Schedule

The schedule defines *when and where* jobs will be executed introducing a two dimensional rectangular representation. Such schedule is represented as an array of particular cluster's schedules, i.e., $schedule := [cl\_sched_1, .., cl\_sched_l]$. Using this notation, $schedule[k]$ denotes the schedule of cluster k (i.e., cl_sched_k) and it is stored as a linear list of jobs ordered according to the expected start times of the jobs. If two or more jobs in the cluster's schedule have the same start time, then the one being assigned to the CPU with the smallest *id* becomes the predecessor of the remaining jobs in the list and so on. Once some job is added into the schedule it contains information about its expected start time, completion time, and a list of assigned CPUs, i.e., these parameters are properly instantiated in the object that represents this job. An example of the $schedule[k]$ and corresponding $2D\,schedule[k]$ is shown in Figure 4.2(a)[8].

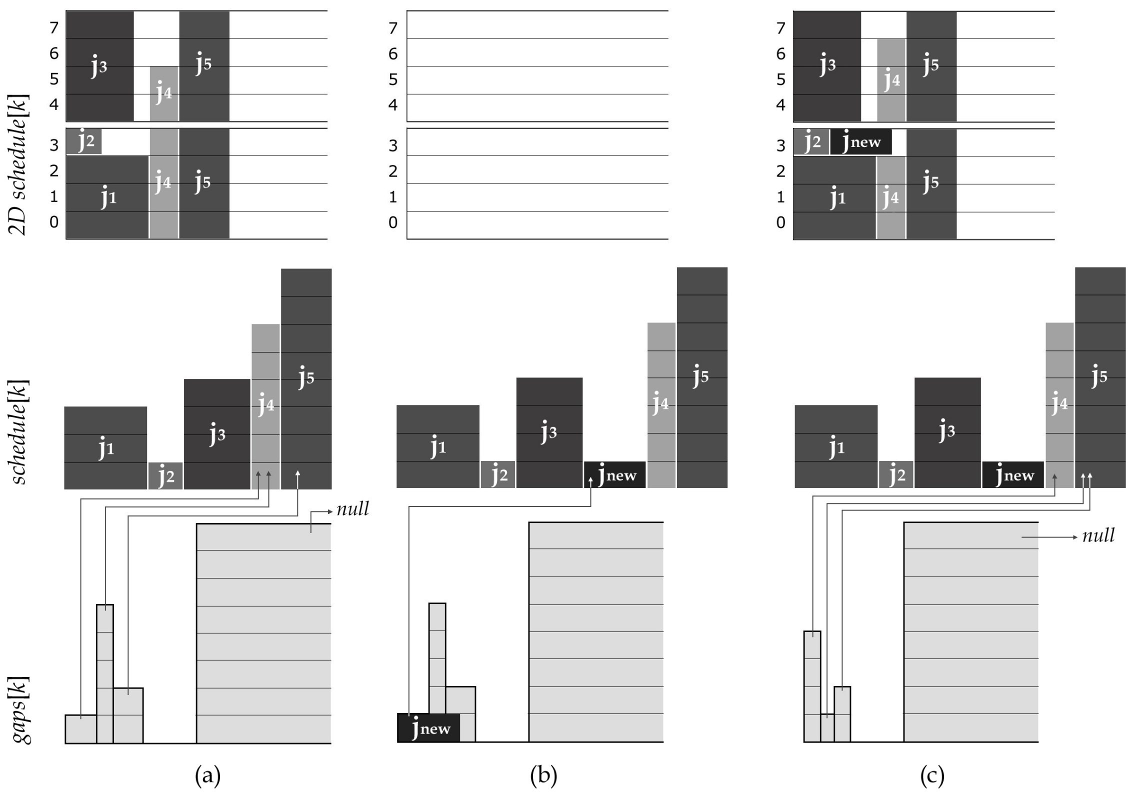


Figure 4.2: Schedule and gap representation.

The $2D\,schedule[k]$ often contains time periods when not all CPUs are used by the jobs. These idle periods are called *gaps*. A gap is considered to be the period of idle CPU time (see Section 3.3.6). It appears every time the number of available CPUs of

[8] $2D\,schedule[k]$ is used only to demonstrate translation of $schedule[k]$ into "human readable" format.

the cluster in the existing schedule is greater than the number of CPUs requested by the job(s) in the given time period. An infinite gap also appears at the end of each cluster's schedule (after the last job) or when there are no jobs in some cluster's schedule at all. Gaps for given cluster k are stored in a similar data structure as jobs — in the linear list $gaps[k]$. Just like jobs, gaps hold information about their start time, duration and usage. Here usage expresses the number of available (idle) CPUs in this gap. Moreover, each gap has a pointer to the nearest following job. If there is no following job (gap at the end of the $schedule[k]$) this pointer is set to *null*. For given time t, there is always at most one gap in the $gaps[k]$. Two successive gaps in the $gaps[k]$ have either different usage or the completion time of earlier gap is smaller than the start time of the later gap. Otherwise, such gaps are merged into a single, longer gap. An example of $gaps[k]$ corresponding to the $2D\,schedule[k]$ is shown in the bottom of Figure 4.2(a).

4.5 Operations over Schedule

Once we have defined the data structure representing the schedule we can describe basic operations frequently performed over the schedule. We will demonstrate how suitable gap(s) for given job can be found and used. Next, an update procedure will be shown, that is used to keep the applied data structures up to date. Finally, we demonstrate how the schedule is evaluated. We will discuss the computational complexity of each such solution using the "big Oh" asymptotic notation [64, 106, 81] depicting the upper bound of algorithm runtime. Since some of the proposed solutions are quite complex algorithms, the resulting formulas may as well be rather complicated. Therefore, standard transformations [64] toward simpler formulas will be applied when advisable. Surprisingly, the literature often discusses these transformation rules only for functions with a single parameter ($f(n)$) [81, 37, 70, 94, 118]. However, in our solution, many algorithms have more than one natural parameter influencing their performance. Intuitively, the complexity of schedule-related algorithm will often depend both on the number of jobs (n) and the number of available CPUs ($cpus$), i.e., functions like $f(n, cpus)$ can be expected. Fortunately, it has been shown that asymptotic notation as well as standard transformations can be applied for functions with multiple variables, provided the worst case (or expected case) runtimes of algorithms are strictly nondecreasing with the size of the problem [81]. Since this holds for our solution, standard transformations of asymptotic notations can be applied.

As discussed in Section 2.5, as soon as machine failures or user activity is considered we should use n^t, l^t, $cpus^t$ or $cpus^t_k$ when discussing the number of *currently* available jobs, clusters or the number of CPUs (of cluster k), respectively. However, with this notation the resulting formulas describing computational complexity become rather complicated. To keep the formulas as simple as possible, we prefer simplified notation using n, l, $cpus$ or $cpus_k$ instead. Not only that the resulting formulas are much simpler, they also remain correct. As can be seen in Formula 4.1, n, l, $cpus$ and $cpus_k$ represent upper bounds.

$$n^t \leq n, \quad l^t \leq l, \quad cpus^t \leq cpus, \quad cpus^t_k \leq cpus_k \qquad \forall t \text{ s.t. } t \geq 0 \tag{4.1}$$

Therefore, Formulas 4.2 and 4.3 hold, thus the applied transformation is correct with respect to the asymptotic computational complexity.

$$n^t = \mathcal{O}\left(n\right), \quad l^t = \mathcal{O}\left(l\right) \qquad \forall t \text{ s.t. } t \geq 0 \tag{4.2}$$

$$cpus^t = \mathcal{O}\left(cpus\right), \quad cpus_k^t = \mathcal{O}\left(cpus_k\right) \qquad \forall t \text{ s.t. } t \geq 0 \tag{4.3}$$

4.5.1 Gap Filling

One of the most important operation is related to the applied gap filling approach. As defined in Section 4.1.3 both the scheduling policy and the optimization procedure often use gap filling approach to detect and fill existing gaps in the schedule with suitable jobs. To allow this, specific operation is necessary that for given job finds suitable gap(s) at cluster k. This is done by $FindFirstGap(k, job)$ procedure shown in Algorithm 4.1. It works as follows. First, it test whether cluster k is suitable to execute the job, i.e., whether all job's requirements can be satisfied by this cluster (line 2). If so, the $gaps[k]$ list is traversed (lines 5–27) to find the first gap that is large enough for the job (line 9). If a single gap is not large enough, two or more adjacent gaps can be used for the job (see inner cycle at lines 12–25) if following constraints for all such gaps are met. All gaps must at least have the same usage as the job itself. Also the completion time of the preceding gap must be equal to the start time of the next gap (line 16). Finally, the duration of the job must be less than or equal to the total duration of the considered gaps (line 22). In case that the inner cycle fails to find suitable gaps, the counter of outer loop is increased to skip gaps that have already been tested in this inner loop (line 19). Once suitable gap(s) is found (line 9 or 22) the $FindFirstGap(k, job)$ procedure returns the found gap (the first gap if adjacent gaps are used). Gap's pointer can now be used to place the job in the $schedule[k]$. If the pointer is $null$, it means that there are no suitable "inside gaps" and the infinite (last) gap at the end of $schedule[k]$ was selected by the algorithm. It means that the job will be placed at the end of $schedule[k]$ list (see Section 4.4.2).

Figure 4.2(a) shows some existing schedule of cluster k, i.e., $schedule[k]$, $gaps[k]$ and the corresponding "human readable" $2D\, schedule[k]$. Figure 4.2(b) demonstrates how the $FindFirstGap(k, job)$ procedure works, i.e., how the suitable gaps are found for a new job j_{new}. Once these gaps are found the pointer of the first gap is used to place j_{new} into the $schedule[k]$, between j_3 and j_4. Figure 4.2(c) shows the new schedule, including the updated $gaps[k]$ list.

To conclude, let us discuss the time complexity of the $FindFirstGap(k, job)$ procedure. As discussed, two or more adjacent gaps can be used for placement of a new job if necessary constraints are met. Therefore, the process of finding of the first suitable gap(s) for a job has at worst linear complexity with respect to the number of gaps in the $gaps[k]$ list. Let g_k is the number of gaps in $gaps[k]$ list, then the number of necessary operations is bounded by Formula 4.4.

$$FindFirstGap(k, job) \quad = \quad \mathcal{O}(g_k) \tag{4.4}$$

Algorithm 4.1 *FindFirstGap*(k, job)

```
1: $total\_gaps$ = size of $gaps[k]$;
2: if cluster $k$ is not suitable for $job$ then
3:   return not found;
4: end if
5: for $i := 1$ to $total\_gaps$ do
6:   $current\_gap := gaps[k][i]$; (selects $i$-th gap in the $gaps[k]$ list)
7:   $duration$ := duration of $current\_gap$;
8:   $usage$ := usage of $current\_gap$;
9:   if $duration \geq ep_{job}$ and $usage \geq usage_{job}$ then
10:    return $current\_gap$;
11:  else if $usage \geq usage_{job}$ then
12:    for $j := i + 1$ to $total\_gaps$ do
13:      $usage_{next}$ := usage of $gap[k][j]$;
14:      $start_{next}$ := start time of $gap[k][j]$;
15:      $end_{prev}$ := end time of $gap[k][j-1]$;
16:      if $usage_{next} \geq usage_{job}$ and $start_{next} = end_{prev}$ then
17:        $duration$ += duration of $gap[k][j]$;
18:      else
19:        $i := j - 1$; (counter of outer loop is increased)
20:        break;
21:      end if
22:      if $duration \geq ep_{job}$ then
23:        return $current\_gap$;
24:      end if
25:    end for
26:  end if
27: end for
```

The value of g_k is related to the number of jobs in the *schedule*$[k]$. Let n_k is the number of jobs in *schedule*$[k]$. Then Formula 4.5 holds as explained in the following text.

$$g_k \quad \leq \quad n_k + 1 \tag{4.5}$$

Clearly, for empty *schedule*$[k]$ there is a single infinite gap. When new job arrives, it uses part of this gap. If the job's usage is equal to the number of CPUs in *schedule*$[k]$ then no new gap will appear. Otherwise, one new gap will appear. Let us assume that there are x gaps in *gaps*$[k]$ list and n_k jobs in the *schedule*$[k]$ and a new job is being added to the schedule. Then, three different cases may happen. If the job's duration and usage is exactly the same as of the selected gap, then this gap will disappear and $g_k = x-1$. In case that job's usage is equal to the gap's usage but job's duration is smaller, the duration of the gap will be shortened but no new gap will appear, thus the number of gaps will remain the same ($g_k = x$). Finally, if both the duration and the usage of the job is smaller than the duration of the gap, this gap will be divided into two gaps, thus the $g_k = x+1$. Similar situation applies when two or more gaps are used. Let us assume that y successive gaps have been used to place the job. Then, each of the first $y-1$ gaps will either disappear or its usage will be reduced by the usage of the job. No new gaps will appear for the first $y-1$ gaps. For the last y-th gap applies the same rules as we have shown for the single gap case. Such gap will either disappear, it will be reduced, or it will be divided into two different gaps. Therefore, at most one new gap will appear due to a new job arrival.

Let g is the total number of gaps in the whole *schedule* that contains n jobs. Then Formula 4.7 holds as a trivial result of Formulas 4.5 and 4.6.

$$g \leq \sum_{k=1}^{l}(n_k+1) \tag{4.6}$$

$$g \leq n+l \tag{4.7}$$

4.5.2 Schedule Update

Schedule update is an important operation that is used every time the *schedule* data structure has been modified. It updates internal parameters of jobs stored within the *schedule* list. Moreover, the corresponding *gaps* list is updated as well. When the *schedule* is modified, e.g., due to a job arrival, machine failure or compression, it means that both jobs' parameters as well as gaps may change. For example, upon job arrival the *FindFirstGap*(k, job) procedure is used to find suitable gap(s) for new job. Once such gap(s) is found, the job is placed into it[9]. Therefore, existing *gaps*$[k]$ list is no longer valid since some gap(s) have been utilized with the new job. Also the internal parameters of the new job such as expected start time, completion time, etc., have to be correctly instantiated in the corresponding *schedule*$[k]$ list. Such update is performed by the *UpdateSchedule*(k, t) procedure shown in Algorithm 4.2. First, the existing *gaps*$[k]$ list is erased (line 1). Then an auxiliary list $first\_free\_slots$ is created that is lately used to store information about the first free time slot on given CPU_i[10]. At the beginning, it is initialized (lines 3–13) either to the current time t (CPU_i is currently free, line 8), to the estimated completion time of job that currently uses CPU_i (CPU_i is busy, line 11), or it is set equal to infinity, provided that the given CPU_i is currently failed (line 6).

Then, internal "schedule-related" variables of all waiting jobs are updated with the *UpdateJob*$(job, first\_free\_slots)$ procedure (line 16)[11]. This procedure is shown in Algorithm 4.3. First, the schedule-related parameters of the given job are reset (lines 1 and 2 in Algorithm 4.3). In the first iteration ($j = usage_{job}$) the earliest possible start time and the expected completion time of the job are found using the auxiliary $first\_free\_slots$ list

[9]The job is placed into *schedule*$[k]$ using the pointer stored in the gap.

[10]CPU_i denotes the i-th CPU on cluster k, such that $1 \leq i \leq cpus_k$.

[11]As has been discussed in Section 4.2.1, no special check must be done to ensure that the number of currently running CPUs is greater than or equal to $usage_{job}$ since all jobs requesting unavailable resources are always killed and removed from the *schedule*$[k]$ before the update procedure starts.

Algorithm 4.2 *UpdateSchedule*(k, t)

```
1: $first\_free\_slots$ := new list with size $cpus_k$; $max\_compl\_time$ := 0; $gaps[k]$ := null;
2: $n_k$ := size of $schedule[k]$; $running\_CPUs$ := $cpus_k$;
3: for $i$ := 1 to $cpus_k$ do
4:    if $CPU_i$ is failed then
5:       $running\_CPUs$ := $running\_CPUs$ - 1;
6:       $first\_free\_slots[i]$ := $\infty$;
7:    else if $CPU_i$ is free then
8:       $first\_free\_slots[i]$ := $t$;
9:    else
10:      $job$ := job currently running on $CPU_i$;
11:      $first\_free\_slots[i]$ := $C_{job}$; ($CPU_i$ becomes free at time $C_{job}$ as $job$ completes)
12:   end if
13: end for
14: for $i$ := 1 to $n_k$ do
15:    $job$ := $schedule[k][i]$; (selects $i$-th job in the $schedule[k]$ list)
16:    $gaps_{job}$ := $UpdateJob(job, first\_free\_slots)$;
17:    $gaps[k]$ := append $gaps_{job}$ to $gaps[k]$ list;
18:    $max\_compl\_time$ := max($max\_compl\_time, C_{job}$);
19: end for
20: $last\_gap$ := new $Gap(max\_compl\_time, \infty, \infty, running\_CPUs, null)$;
21: $gaps[k]$ := append $last\_gap$ to $gaps[k]$ list;
22: $updated[k]$ := true;
```

(lines 4–8 in Algorithm 4.3). Remaining iterations are used to construct the list of gaps that may appear "in front" of this job, i.e., prior the job's start time. Once the gap is detected (line 10), its start time, end time, duration and usage are computed (lines 11–13 in Algorithm 4.3). Then, an object representing this gap is created and the gap is stored in the $gaps_{job}$ list as shown by lines 14–15 in Algorithm 4.3. Finally, the field corresponding to the first free time slot on the currently considered CPU ($first\_free\_slots[CPU_{index}]$) is set equal to the expected job completion time (line 18 in Algorithm 4.3). It guarantees that all CPUs used by the *job* will be considered as busy during the whole execution of the *job*, thus no other job can be planned on these CPUs during this time interval. The constructed $gaps_{job}$ list is returned to the Algorithm 4.2 as a result of the *UpdateJob* procedure (see line 16 in Algorithm 4.2).

This list is appended to the newly constructed $gaps[k]$ list[12]. Finally, the last gap with the infinite duration (see Section 4.4.2) is created as is shown in line 20. Its usage is equal to the number of running CPUs (*running_CPUs*) and it is added as the last gap into the $gaps[k]$ list (line 21).

UpdateSchedule procedure updates all schedule-related internal parameters of jobs

[12]If the $gaps_{job}$ list is empty, then the $gaps[k]$ remains unchanged.

Algorithm 4.3 *UpdateJob*$(job, first\_free\_slots)$

```
1: $S_{job}$ := -1; $C_{job}$ := -1; $gap\_end$ := -1;
2: $\mathcal{CPU}_{job} := \emptyset$; $gaps_{job}$ := new empty list;
3: for $j := usage_{job}$ down to 1 do
4:    $CPU\_index$ := index of the $j$th smallest value in $first\_free\_slots$;
5:    $\mathcal{CPU}_{job} := \mathcal{CPU}_{job} \cup CPU\_index$;
6:    if $j = usage_{job}$ then
7:       $S_{job} := first\_free\_slots[CPU\_index]$;
8:       $C_{job} := S_{job} + ep_{job}$;
9:    end if
10:   if $first\_free\_slots[CPU\_index] < gap\_end$ then
11:      $gap\_start := first\_free\_slots[CPU\_index]$;
12:      $gap\_duration := gap\_end - gap\_start$;
13:      $gap\_usage := j$;
14:      $gap$ := new $Gap(gap\_start, gap\_end, gap\_duration, gap\_usage, job)$;
15:      $gaps_{job}$ := add $gap$ at the first position in $gaps_{job}$;
16:   end if
17:   $gap\_end := first\_free\_slots[CPU\_index]$;
18:   $first\_free\_slots[CPU\_index] := C_{job}$;
19: end for
20: return $gaps_{job}$;
```

stored in *schedule*$[k]$ list while also updating the corresponding *gaps*$[k]$ list. Once it terminates, all necessary information related to the schedule of cluster k are up to date. Since the *UpdateSchedule* procedure is potentially a time consuming operation, it should be executed only when it is desirable. Redundant executions must be avoided. For example, when the job completes early, it is necessary to update (compress) the schedule. On the other hand, on time job completion or schedule evaluation do not usually require the update. To avoid such redundant executions an auxiliary boolean list $updated := [updated_1, .., updated_l]$ is created, where *updated*$[k]$ denotes whether *schedule*$[k]$ is up to date or not. Every time the update of *schedule*$[k]$ is performed, *updated*$[k]$ is set to true as can be seen on line 22 in Algorithm 4.2. As soon as some change in the *schedule*$[k]$ appears, the *updated*$[k]$ is invalidated (set to false), implying that an update is now desirable. This list is a persistent data structure, therefore it is not destroyed when the update procedure finishes. An example of *UpdateSchedule*(k, t) application can be shown in Figure 4.2(b). It represents initial situation where new job (j_{new}) has been placed into *schedule*$[k]$ using some available gap. At this moment, the schedule is not up to date, since *gaps*$[k]$ list and internal job parameters have not been updated. This is expressed by the empty $2D$ *schedule*$[k]$. Once the *UpdateSchedule*(k, t) is executed both *gaps*$[k]$ and internal job parameters are updated (see Figure 4.2(c)), therefore $2D$ *schedule*$[k]$ can be correctly created, showing the new, up to date, schedule.

Let us discuss the time complexity of the *UpdateSchedule* procedure. The procedure computes new up to date values for all jobs and gaps in the *schedule*[*k*] and *gaps*[*k*] lists. Let us assume that there are $cpus_k$ CPUs in the cluster k and n_k jobs in the *schedule*[*k*]. First, the procedure initializes the $first\_free\_slots$ data structure which requires $cpus_k$ steps. Then, for all n_k jobs the *UpdateJob* procedure is executed, updating internal values of each job and computing the list of gaps. It can be shown, that for n_k jobs the complexity related to these updates is in $\mathcal{O}(n_k \cdot cpus_k)$. Since such analysis is not straightforward we present it in Appendix A. Together, the complexity of *UpdateSchedule* is bounded by Formula 4.8.

$$\begin{aligned} UpdateSchedule(k,t) &= \mathcal{O}\left(cpus_k\right) + \mathcal{O}\left(n_k \cdot cpus_k\right) \\ &= \mathcal{O}(cpus_k \cdot n_k) \end{aligned} \tag{4.8}$$

It is important to realize that the *UpdateSchedule* procedure is able to compress the schedule of cluster k if some job completes earlier than expected. This is the result of the *UpdateJob* procedure that finds the earliest start time for each job (see Algorithm 4.3). Moreover, *UpdateJob* procedure simultaneously constructs the *gap*[*k*] list and the job's CPU list (line 15 and 5 in Algorithm 4.3). Therefore, no additional computation is necessary to update the *gaps*[*k*], $\mathcal{CPU}_{job}$ or to compress the schedule, and the complexity related to such operations is amortized by the $\mathcal{O}(cpus_k \cdot n_k)$ complexity of the *schedule*[*k*] update procedure. Unlike the Conservative Backfilling, our implementation of the compression algorithms keeps the original ordering of expected job start times. Although this implementation is less efficient, it is much faster since the schedule is traversed only once. In Conservative Backfilling, the complexity of compression is quadratic because the schedule is scanned again for each job [131]. The difference between Conservative Backfilling and our implementation is shown in Figure 4.3.

4.5.3 Schedule Evaluation

Schedule evaluation is one of the most important features of the proposed approach. In the following text we describe how this evaluation is implemented in our solution. Schedule evaluation is used to measure the quality of the solution (schedule). Therefore, one or more objective functions like those presented in Section 2.6 can be used. All of them use information related to the job or machine performance. Typically, job wait time is used to compute functions like wait time, response time or slowdown while the number of utilized CPUs over given period is used to measure (weighted) machine usage. Except for few criteria like makespan, the schedule must be analyzed job by job to compute the value of desired objective functions.

The following Algorithm 4.4 shows an example of how such objective functions can be computed. Objective functions are specified using the $criteria$ parameter. The *schedule* is analyzed cluster by cluster and job by job using two loops. To obtain correct values, it is necessary to use an up to date *schedule*. Therefore, each *schedule*[*k*] is firstly updated if the corresponding *updated*[*k*] = **false** (line 6). For each job, the *ComputeSubresults(job,*

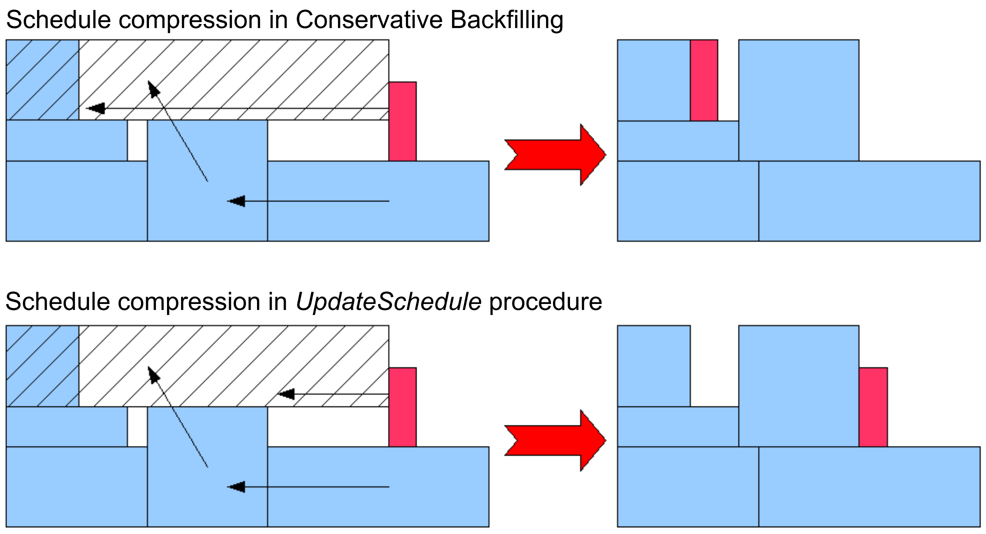

Schedule compression in Conservative Backfilling
Schedule compression in *UpdateSchedule* procedure

Figure 4.3: The difference in the implementation of the schedule compression algorithms in Conservative Backfilling (top) and in *UpdateSchedule* procedure (bottom).

Algorithm 4.4 *ScheduleEvaluation*(*schedule*, *criteria*, *t*)

```
1: $n$ := number of jobs in $schedule$;
2: $index$ := 1;
3: $subresults$ := new empty list with size $n$;
4: for $k$ := 1 to $l$ do
5:     if $updated[k]$ = false then
6:         $UpdateSchedule(k, t)$;
7:     end if
8:     $n_k$ = size of $schedule[k]$;
9:     for $j$ := 1 to $n_k$ do
10:        $job$ := $schedule[k][j]$;
11:        $subresults[index]$ := $ComputeSubresults(job, criteria)$;
12:        $index$++; (increase the index for $subresults$ list)
13:    end for
14: end for
15: $results$ := $ComputeFinalResults(subresults, criteria)$;
16: return $results$;
```

criteria) procedure is used to calculate intermediate values, which are lately used to compute the values of objective functions. These values are stored in the *subresults* list (line 11). Its size is equal to the number of jobs in the schedule, i.e., it stores necessary intermediate values independently for each job. Once all jobs are browsed, the resulting objective function value can be computed using the *ComputeFinalResults*(*subresults*, *criteria*) procedure and returned (line 16). For example, if *criteria* = "wait time", then *ComputeSubresults(job, criteria)* computes jobs' wait times and stores them in the *subresults* list. Next, *ComputeFinalResults(subresults, criteria)* uses this list to compute the requested average value.

As discussed, jobs' internal values must be up to date to obtain correct results. This is guaranteed by the update procedure that must be performed if the *schedule*[*k*] has been changed since the last update. Similarly as the update procedure, also the evaluation procedure browses each job in the particular *schedule*[*k*] list. Therefore, it is a good idea to merge the update and evaluation together to limit the computational requirements of the evaluation procedure. If such merging is applied the *ComputeSubresults*(*job*, *criteria*) can be incorporated into the *UpdateJob*(*job*, *first_free_slots*) (Algorithm 4.3). The only required change is to make the *subresults* list persistent and available to both procedures. If so, the whole *subresults* list does not have to be recomputed every time the *ScheduleEvaluation* is executed, delivering the result much faster. This optimized evaluation procedure is shown in Algorithm 4.5.

Algorithm 4.5 *OptimizedScheduleEvaluation*(*schedule*, *criteria*, *t*)

```
1: $n$ := number of jobs in $schedule$;
2: for $k := 1$ to $l$ do
3:    if $updated[k]$ = false then
4:       $UpdateSchedule(k, t)$; (now with modified $UpdateJob$ subroutine)
5:    end if
6: end for
7: $results := ComputeFinalResults(subresults, criteria)$;
8: return $results$;
```

Clearly, in the *OptimizedScheduleEvaluation*(*schedule*, *criteria*, *t*) we were able to remove the inner for cycle by moving its logic into the slightly modified *UpdateJob* procedure, which is called from the *UpdateSchedule* procedure. Also, *updated*, *subresults* and *criteria* lists are now persistent, i.e., they do not have to be specified as an input parameters of these procedures. Using the *criteria* list, *UpdateSchedule* now can compute intermediate values of requested objectives and store them into the persistent *subresults* list. Next, the final values of all requested objective functions are computed (line 7) and returned in a *results* list (line 8). Necessary modifications of *UpdateJob* procedure are not shown here, since they are all trivial. This optimized evaluation procedure represents the implementation applied in our solution.

To conclude, the complexity of *OptimizedScheduleEvaluation* will be discussed. Let us start with the unlikely worst case scenario when the *UpdateSchedule* must be performed for all currently available clusters, i.e., l times. The *subresults* list is computed job by job in the *UpdateJob* procedure. Let *size* is the number of applied objective functions (size of *criteria* list). Clearly, *size* will always be some relatively small constant. Let a be a constant that bounds the number of computational steps needed to compute the intermediate value of objective function for given job[13]. In this case, the time needed to compute the intermediate values of objective functions is $\mathcal{O}(size \cdot a) = \mathcal{O}(1)$. Therefore, the complexity of the *UpdateJob* procedure is still bounded by $\mathcal{O}(cpus_k)$, since $\mathcal{O}(1 + cpus_k) = \mathcal{O}(cpus_k)$ and the complexity of *UpdateSchedule* remains $\mathcal{O}(cpus_k \cdot n_k)$. The final results are created using the *ComputeFinalResults* function that uses the *subresults* list of intermediate values. Since for many objective functions a "job by job" approach is required we can assume that the size of the *subresults* list is proportional to the number of jobs (n) and objective functions ($size$). Therefore, the complexity of *ComputeFinalResults* is bounded by Formula 4.9 and the complexity of the *OptimizedScheduleEvaluation* procedure is bounded by Formula 4.10.

$$\begin{aligned} ComputeFinalResults(...) &= \mathcal{O}(size \cdot n) = \mathcal{O}(n) && (4.9)\\ OptimizedScheduleEvaluation(...) &= \mathcal{O}(l) + \mathcal{O}\left(\sum_{k=1}^{l} cpus_k \cdot n_k\right) + \mathcal{O}(n) && (4.10)\end{aligned}$$

Since all equations in Formula 4.11 hold[14], and using the fact that $l \leq cpus$, it can be further simplified to Formula 4.12.

$$\sum_{k=1}^{l} cpus_k = cpus, \qquad \sum_{k=1}^{l} n_k = n, \qquad \sum_{k=1}^{l} cpus_k \cdot n_k \leq cpus \cdot n \qquad (4.11)$$

$$\begin{aligned} OptimizedScheduleEvaluation(...) &= \mathcal{O}(l + cpus \cdot n + n)\\ &= \mathcal{O}(cpus \cdot n) && (4.12)\end{aligned}$$

This represents the worst case, when all l clusters' schedules have to be updated. However, in most situations this is not necessary. Since we use incremental approach, schedule modifications appear locally in given *schedule*$[k]$ only, while remaining clusters' schedules remain up to date. In this case the complexity of *OptimizedScheduleEvaluation* is bounded by Formula 4.13 as only *schedule*$[k]$ has to be updated.

$$OptimizedScheduleEvaluation(...) = \mathcal{O}(l + cpus_k \cdot n_k + n) \qquad (4.13)$$

4.5.4 Acceptance of Schedule

Schedule evaluation measures the quality of given *schedule* using one or more objective functions. Therefore, the result of such evaluation is some value, e.g., the slowdown.

[13]We can assume that this number is some rather small constant since the operations needed here are typically very simple, e.g., addition or subtraction of two numbers.

[14]Rightmost equation in Formula 4.11 is correct, since $cpus = (cpus_1 + ... + cpus_l), n = (n_1 + ... + n_l)$, therefore $cpus \cdot n = (cpus_1 + ... + cpus_l) \cdot (n_1 + ... + n_l)$. Then $cpus \cdot n \geq \sum_{k=1}^{l} cpus_k \cdot n_k$ always holds, because both $cpus_k$ and n_k are always positive (≥ 0) numbers.

When two different solutions are available, the evaluation is not capable to decide which one is more suitable for us. It only provides us with values that somehow characterize the solution. The final decision is implemented in a separate procedure, which is shown in Algorithm 4.6. This procedure uses three inputs. The first two inputs are the two solutions that will be compared. The $schedule_{curr}$ represents existing (previously accepted) solution while $schedule_{new}$ represents the newly created *candidate solution*. The $schedule_{new}$ is typically a product of optimization or it represents new possible solution when assigning job into the existing schedule. The third parameter specifies the criteria that will be used when comparing these two candidates.

Algorithm 4.6 $AcceptCandidate(schedule_{curr}, schedule_{new}, criteria)$

```
1: $t$ := current wall clock time; $weight := 0$; $size$ := size of $criteria$ list;
2: $results_{curr}$ := new empty list; $results_{new}$ := new empty list;
3: $results_{curr} := OptimizedScheduleEvaluation(schedule_{curr}, criteria, t)$;
4: $results_{new} := OptimizedScheduleEvaluation(schedule_{new}, criteria, t)$;
5: for $i := 1$ to $size$ do
6:     $weight[i] := \frac{results_{curr}[i] - results_{new}[i]}{results_{curr}[i]}$;
7:     $weight \mathrel{+}= c_i \cdot weight[i]$;
8: end for
9: if $weight > 0$ then
10:     return true;
11: else
12:     return false;
13: end if
```

The $AcceptCandidate(schedule_{curr}, schedule_{new}, criteria, subresults)$ returns true if the $schedule_{new}$ is better than current $schedule_{curr}$, otherwise it returns false. It works in following way. First, the values of used objective functions (*criteria* parameter) are computed for both schedules (lines 3–4). Using them, decision variables $weight[1], \dots, weight[size]$ are computed in the main loop (see lines 5–8). Their meaning is following: when the $weight[i]$ is positive it means that the $schedule_{new}$ is better than the $schedule_{curr}$ with respect to the applied criterion[15]. Strictly speaking, $weight[i]$ defines percentual improvement or deterioration in the value of objective function of $schedule_{new}$ with respect to the $schedule_{curr}$. To be precise, this only holds for objective functions that should be minimized, e.g., slowdown, response time, wait time. For objectives that shall be maximized (e.g., machine usage) the order of values in the numerator must be swapped, i.e., line 6 must be rewritten as shown in Formula 4.14. In order to keep the code clear, we kindly ask the reader to accept this simplification.

[15]It is the i-th criterion from the *criteria* list.

$$weight[i] := \frac{results_{new}[i] - results_{curr}[i]}{results_{curr}[i]} \tag{4.14}$$

It can easily happen, that for given $schedule_{new}$ some weights are positive while others are negative. In our implementation the final decision is taken upon the value of the *weight* (line 7), which is computed in the main loop as the sum of $c_1 \cdot weight[1]$,..., $c_{size} \cdot weight[size]$. If some of the used objectives is considered to be more or less important, then corresponding c_i constant can be used to increase or decrease the effect of each partial objective. However, proper selection of these values is quite difficult [188]. In our work, each of these constants is always equal either to 1 or to 0, according to the studied problem. Clearly, 0 constant causes that the value of corresponding criterion is ignored. Some trivial correction is needed when the $results_{curr}[i]$ is equal to zero, to prevent division by zero error. Again, to keep the code clear we do not present it here.

The complexity of *AcceptCandidate* is based on the $\mathcal{O}(cpus \cdot n)$ complexity of the evaluation procedure which is computed twice here. Then, $weight[1], \ldots, weight[size]$ are computed. Again the number of operations required to compute $weight[i], i \in (1, ..., size)$, can be bounded by sufficiently large constant a. Therefore, to compute the final *weight* value, at most $(a \cdot size)$ steps are needed. Therefore, the complexity of the *AcceptCandidate* can be bounded as follows.

$$\begin{aligned} AcceptCandidate(...) &= \mathcal{O}(cpus \cdot n) + \mathcal{O}(cpus \cdot n) + \mathcal{O}(a \cdot size) \\ &= \mathcal{O}(cpus \cdot n) + \mathcal{O}(1) \\ &= \mathcal{O}(cpus \cdot n) \end{aligned} \tag{4.15}$$

Again, this represents the worst case when all clusters' schedules in both $schedule_{curr}$ and $schedule_{new}$ must be updated. On the other hand, in many cases the existing $schedule_{curr}$ is known and updated *before* the *AcceptCandidate* is executed, thus the evaluation of the $schedule_{curr}$ is bounded by $\mathcal{O}(l + n)$, since l tests whether $updated[k] =$ **false** must be performed before the values of requested objective functions are computed ($\mathcal{O}(n)$ steps) using the *subresults* list[16]. Moreover, in the new schedule only the given $schedule_{new}[k]$ usually requires the update (see Formula 4.13 in Section 4.5.3). Therefore, the complexity can be bounded as shown by Formula 4.16.

$$\begin{aligned} AcceptCandidate(...) &= \mathcal{O}(l + n) + \mathcal{O}(l + cpus_k \cdot n_k + n) + \mathcal{O}(a \cdot size) \\ &= \mathcal{O}(l + cpus_k \cdot n_k + n) + \mathcal{O}(1) \\ &= \mathcal{O}(l + cpus_k \cdot n_k + n) \end{aligned} \tag{4.16}$$

[16]When the whole schedule is up to date, i.e., $updated[k] =$ **true** for every cluster, then objective functions do not have to be recomputed and the complexity can be further reduced to $\mathcal{O}(l)$. This would require some trivial modifications of the schedule evaluation procedure. For simplicity, we do not present them here.

4.6 Scheduling Policies

In this section we present implementations of proposed scheduling policies. As discussed, the main purpose of scheduling policy is to generate initial schedule as fast as possible, because the scheduling policy is applied at each job arrival. It is also used when some jobs must be rescheduled due to a machine failure.

We present two different scheduling policies in the following text. They both have similar base which is the gap filling approach. The differences among these policies are related to the problems they are applied to and will be closely discussed in the text.

4.6.1 Best Gap

The *Best Gap (BG)* is the basic policy applied in our solution. Its main idea is similar to Conservative Backfilling. Upon each job arrival at each suitable cluster the first suitable gap is identified. Conservative Backfilling would choose the earliest gap if two or more are available on different clusters. Instead of that, BG uses evaluation to decide which gap is the best one. The decision is taken using the *AcceptCandidate* procedure with respect to the objective function(s) specified in the *criteria* list. The pseudo code is shown in Algorithm 4.7.

Algorithm 4.7 *BestGap*(*job*, *criteria*)

```
1: $schedule_{initial} := [cl\_sched_1, .., cl\_sched_l]$; $schedule_{new} := \emptyset$; $schedule_{best} := \emptyset$; $k := 0$;
   $pointer := null$;
2: for $k := 1$ to $l$ do
3:   if $cluster_k$ is suitable to perform $job$ then
4:     $schedule_{new} := schedule_{initial}$;
5:     $pointer := FindFirstGap(k, job)$;
6:     $schedule_{new}[k]$ := place $job$ in the gap in the $schedule_{new}[k]$ using the $pointer$;
7:     if $schedule_{best} = \emptyset$ then
8:       $schedule_{best} := schedule_{new}$; ($schedule_{best}$ is correctly initialized)
9:     else if $AcceptCandidate(schedule_{best}, schedule_{new}, criteria)$ then
10:      $schedule_{best} := schedule_{new}$;
11:    end if
12:  end if
13: end for
14: return $schedule_{best}$
```

BestGap(*job*, *criteria*) adds newly arriving *job* into the existing schedule, denoted as the $schedule_{initial}$ following the incremental — time saving — approach. It finds a suitable cluster and a proper time slot for the *job*. When the policy finishes its execution, it places the new *job* into the selected cluster's schedule. The details of the implementation follows. The procedure selects all suitable clusters' schedules as candidates for the new *job*. The cluster k is considered to be *suitable* if it has enough CPUs to execute the *job*, and if it satisfy all specific job requirements as discussed in Sections 2.3 and 2.4 respectively. *BestGap* is always executed after initial test that guarantees that *job* is executable at least on one cluster. Therefore, it is sure that such suitable cluster will be found. Once the suitable cluster is found (see line 3) the *earliest suitable gap* for the new *job* is found in its schedule using $FindFirstGap(k, job)$ procedure[17]. If two or more suitable gaps are available in the given cluster's schedule the search stops as soon as the first (earliest) gap is found. If some later gap would be used instead, it would only deteriorate the performance for the new job since its wait time, start time, response time, etc., would increase. More importantly, it may not be possible to place future jobs into these earlier gaps as the time is running and the cluster may spend its time being idle not having a suitable job for the gap at the right moment which would decrease the machine usage [104, 105].

Once the suitable gap is found the job is placed into it and the new resulting schedule $schedule_{new}$ is evaluated according to the *AcceptCandidate* function with respect to the best so far found $schedule_{best}$. If this assignment is accepted, then the $schedule_{new}$ becomes the new $schedule_{best}$ (line 10). Otherwise this assignment is rejected. In case that $schedule_{best}$ is empty (lines 7–8), the $schedule_{new}$ is automatically accepted. This guarantees that the $schedule_{best}$ is correctly initialized at the beginning of *BestGap* execution. Finally, a new iteration begins and the $schedule_{new}$ is reset to the initial state (line 4). This cycle continues until all suitable clusters are tested. Then the policy returns the $schedule_{best}$ as the newly found solution.

Let us briefly mention some interesting consequences with respect to the Conservative Backfilling. In case that all suitable clusters have the same speed, then the best gap is always the one with the earliest start time. In such case the *BestGap* behaves exactly like Conservative Backfilling which always selects the earliest possible time slot. Similar rule applies when the whole system consists of a single cluster. On the other hand, if the clusters have different speed, then the earliest gap may not be the best one, depending on the applied objective function. Typically, job's response time may be smaller if the job is executed later on a faster cluster than if it is executed immediately on a slow one.

The complexity of *BestGap* procedure is based on the main cycle which has l steps. In each step the gap is being found (see Formula 4.4) and the *AcceptCandidate* procedure is computed. In the worst case, the complexity of *AcceptCandidate* is bounded by $\mathcal{O}\left(cpus \cdot n\right)$ (Formula 4.15). However, as soon as first iteration finishes, the $updated[k] = \mathbf{true}$ for every cluster k. From now on, the schedule evaluation will execute the most demanding *UpdateSchedule* procedure only for those $schedule[k]$ that have been changed. In the worst case (all clusters are suitable for job) this will happen $l-1$ times. Each time, the complexity of the *AcceptCandidate* will be bounded with Formula 4.16. Therefore the complexity of the whole *BestGap* policy is bounded by Formula 4.17. Here, g is the number of all gaps in the $schedule_{initial}$.

[17]If the cluster k is suitable for the *job* then at least one suitable gap is always found. It is the last, infinite gap situated at the end of cluster's schedule. Its usage is always greater or equal to the $usage_{job}$.

$$\begin{aligned} BG(...) &= \mathcal{O}\left(\sum_{k=1}^{l} g_k\right) + \mathcal{O}\left(cpus \cdot n\right) + \mathcal{O}\left(\sum_{k=2}^{l}(l + cpus_k \cdot n_k + n)\right) \\ &= \mathcal{O}\left(g\right) + \mathcal{O}\left(cpus \cdot n\right) + \mathcal{O}\left(l^2 + cpus \cdot n + l \cdot n\right) \end{aligned} \tag{4.17}$$

Using the Formula 4.7 ($g \leq n + l$), the Formula 4.17 can be further modified into Formula 4.18.

$$BG(...) = \mathcal{O}\left(n + l\right) + \mathcal{O}\left(cpus \cdot n\right) + \mathcal{O}\left(l^2 + l \cdot n\right) + \mathcal{O}\left(cpus \cdot n\right) \tag{4.18}$$

Moreover, since the number of clusters (l) is always bounded by the number of all CPUs ($l \leq cpus$), it can be further simplified as shown in Formula 4.19.

$$\begin{aligned} BG(...) &= \mathcal{O}\left(cpus \cdot n\right) + \mathcal{O}\left(n + l + l^2 + l \cdot n\right) \\ &= \mathcal{O}\left(cpus \cdot n + l^2 + l \cdot n\right) \\ &= \mathcal{O}\left(cpus \cdot n + l^2\right) \end{aligned} \tag{4.19}$$

4.6.2 Best Gap — Earlier Deadline First

The previous *BG* policy is very general, strictly focusing on selecting the best gap with respect to the applied criteria. The *Best Gap — Earlier Deadline First (BG-EDF)* policy represents a modification of the *BG* policy. The purpose of the modification was to increase the probability that the job's deadline will be met. It means that this policy has been specially designed with respect to the late jobs (U) criterion.

The first part of the *BestGap—EarlierDeadlineFirst*(*job*, *criteria*) policy works exactly like *BG* (see lines 1–11 in Algorithm 4.8). *BG* represents general approach suitable for several criteria including the late jobs. When the first (earliest) suitable gap in the given cluster's schedule is used the probability that the job's deadline will be met is higher than if some later gap is used instead.

The difference is that beside the *BG* policy, the second *Earlier Deadline First (EDF)* policy is used in the same iteration. The goal is to decide which strategy will lead to a better solution. The *EDF* policy subsequently goes through the list of jobs in the $schedule_{new}[k]$ (schedule of cluster k) and finds the first job j such that $d_j > d_{job}$ holds. Incoming job is placed right before the job j (job becomes direct predecessor of j in the list $schedule[l]$), shifting job j and all later jobs in the list $schedule[l]$. Note that not all the jobs in the $schedule_{new}[k]$ have to be ordered by their deadline — some job(s) having arrived earlier could have been assigned to this cluster's schedule using the "gap filling" *BG* policy, which does not consider deadline order at all[18]. Clearly, the goal of *EDF* is to assign earlier start times to jobs with approaching deadlines regardless existing

[18] *Earlier Deadline First (EDF)* shall not be confused with the "classical" Earliest Deadline First policy, which keeps *all* jobs ordered with respect to their deadlines.

Algorithm 4.8 *BestGap—EarlierDeadlineFirst*(*job*, *criteria*)

```
1: $schedule_{initial} := [cl\_sched_1, .., cl\_sched_l]$; $schedule_{new} := \emptyset$; $schedule_{best} := \emptyset$; $k := 0$;
   $pointer := null$;
2: for $k := 1$ to $l$ do
3:   if $cluster_k$ is suitable to perform $job$ then
4:     $schedule_{new} := schedule_{initial}$;
5:     $pointer := FindFirstGap(k, job)$;
6:     $schedule_{new}[k]$ := place $job$ in the gap in the $schedule_{new}[k]$ using the $pointer$;
7:     if $schedule_{best} = \emptyset$ then
8:       $schedule_{best} := schedule_{new}$;
9:     else if $AcceptCandidate(schedule_{best}, schedule_{new}, criteria)$ then
10:      $schedule_{best} := schedule_{new}$;
11:    end if
12:    $schedule_{new} := schedule_{initial}$;
13:    $j$ := index of the first job in the $schedule_{new}[k]$ such that $d_j > d_{job}$;
14:    $schedule_{new}[k]$ := insert $job$ into $schedule_{new}[k]$ right before the job $j$;
15:    if $AcceptCandidate(schedule_{best}, schedule_{new}, criteria)$ then
16:      $schedule_{best} := schedule_{new}$;
17:    end if
18:  end if
19: end for
20: return $schedule_{best}$
```

gaps. Such aggressive approach often changes (increases) the start times of previously added jobs that were outrun by the new job. Also, several new gaps may appear in the schedule. New gaps are not a serious problem, they can be utilized in the future by newly arriving jobs. However, it is troublesome if the start times of previously added jobs have been increased too much. Therefore, the newly constructed $schedule_{new}$ must be analyzed by the *AcceptCandidate* to decide whether this solution will be accepted as the new $schedule_{best}$ (line 15). Once all suitable clusters have been tested both by *BG* and *EDF* policy the $schedule_{best}$ is returned as the newly found solution (line 20).

Similarly to *BG*, the complexity of *BG-EDF* is bounded by $\mathcal{O}\left(cpus \cdot n + l^2\right)$. On the other hand, in each iteration, *BG-EDF* performs some additional operations. It passes the $schedule_{new}[k]$ list (at most n_k steps) and then performs additional *AcceptCandidate* procedure. Therefore, the complexity is bounded by Formula 4.20.

$$\begin{aligned} BG\text{-}EDF(...) &= \mathcal{O}\left(\sum_{k=1}^{l}(g_k + n_k)\right) + \mathcal{O}\left(cpus \cdot n\right) \\ &+ \mathcal{O}\left(2 \cdot \sum_{k=2}^{l}(l + cpus_k \cdot n_k + n)\right) \end{aligned} \tag{4.20}$$

Still, after some basic transformations (see Formula 4.21) the upper bound complexity is the same as in the case of *BestGap* algorithm.

$$\begin{aligned} \textit{BG-EDF}(...) &= \mathcal{O}(g+n) + \mathcal{O}(cpus \cdot n) + \mathcal{O}\left(l^2 + cpus \cdot n + l \cdot n\right) \\ &= \mathcal{O}(n+l) + \mathcal{O}(cpus \cdot n) + \mathcal{O}\left(l^2 + l \cdot n\right) + \mathcal{O}(cpus \cdot n) \\ &= \mathcal{O}(cpus \cdot n) + \mathcal{O}\left(n + l + l^2 + l \cdot n\right) \\ &= \mathcal{O}\left(cpus \cdot n + l^2 + l \cdot n\right) \\ &= \mathcal{O}\left(cpus \cdot n + l^2\right) \end{aligned} \tag{4.21}$$

4.7 Optimization Algorithms

The scheduling policies described in the previous section are typically used upon each job arrival to place a new job in an existing schedule. Although these policy use evaluation to produce good schedules, they only focus on the newly arriving job. Previously scheduled jobs are not primarily considered neither by *BG* nor by the *BG-EDF* when building a new solution. Therefore, several optimization algorithms have been proposed to further optimize these *initial solutions*. Similarly, when the existing schedule is compressed due to an early job arrival, several gaps could remain in the schedule. Here the optimization algorithm can be used to identify suitable jobs that would utilize these gaps while increasing the quality of the solution [102].

In this section we describe three different optimization algorithms. All of them are *local search*-based metaheuristics [104, 103, 138, 137] (see Section 3.2.3). As discussed in Section 4.1.2 these algorithms were designed to follow the proposed approaches, i.e., they use event-based, incremental, gap filling and anytime approach. The first algorithm is called *Random Search* and represents general optimization procedure which has been designed to remain universal regardless applied optimization criteria. Second algorithm named *Gap Search* is a fast optimization procedure applied to fix the schedule when early job completions are detected. Finally, the *Tabu Search* has been proposed specifically for problems involving job deadlines. It is a complementary technique to the *BG-EDF* policy that has been developed with the same specific goal.

4.7.1 Random Search

Random Search (RS) optimization algorithm [103, 102] is a local search-based method working in a Hill Climb-like fashion. It means that only improving solutions are accepted [137]. It has been designed to improve the quality of the initial solution delivered by the scheduling policy. Algorithm 4.9 describes the proposed optimization procedure in detail.

The *RandomSearch*(*iterations*, *time_limit*, *stop_event*, *criteria*) procedure is based on the main optimization loop (lines 3–19). In each iteration a random nonempty *schedule*[*k*] is selected (line 5) and a random job from this schedule is removed (line 10). Next,

Algorithm 4.9 *RandomSearch*$(iterations, time\_limit, stop\_event, criteria)$

```
1: $schedule_{best} := [cl\_sched_1, .., cl\_sched_l]$; $schedule_{new} := schedule_{best}$;
2: $i := 0$; $run :=$ **true**;
3: while ($i < iterations$ and $run$) do
4:   $i := i + 1$;
5:   $source :=$ select random $schedule_{new}[k]$ $(1 \leq k \leq l)$ that contains at least one $job$;
6:   if $source = null$ then
7:     return $schedule_{best}$; (schedule is empty so quit)
8:   end if
9:   $job :=$ select random job from $source$;
10:  remove $job$ from $source$;
11:  if $MoveJobRandomly(job, schedule_{best}, schedule_{new}, criteria)$ then
12:    $schedule_{best} := schedule_{new}$; (move is accepted)
13:  else
14:    $schedule_{new} := schedule_{best}$; (move is rejected, $job$ is returned back);
15:  end if
16:  if $time\_limit$ exceeded or $stop\_event$ detected then
17:    $run :=$ false;
18:  end if
19: end while
20: return $schedule_{best}$;
```

the $MoveJobRandomly(job, schedule_{best}, schedule_{new}, criteria)$ procedure described in Algorithm 4.10 is executed.

First, Algorithm 4.10 permutes the list of clusters' schedules so that they will be selected in random order in the main loop (line 1 in Algorithm 4.10). If the cluster k is suitable for the job, the job is placed in the $schedule_{new}[k]$ on a randomly selected position. Such new $schedule_{new}$ is evaluated using the *AcceptCandidate* procedure. If the move is accepted, the $MoveJobRandomly$ procedure returns true and the new schedule is accepted as the new $schedule_{best}$ (line 12 in Algorithm 4.9). We accept the first improving move, since scanning of all clusters was shown to be time consuming during the tests. Better performance is possible when more local changes are processed. If the move is rejected, the $schedule_{new}$ is reset (line 8 in Algorithm 4.10) and the next cluster is selected in the following iteration. The loop continues until some move is accepted or all suitable clusters have been unsuccessfully tried, meaning that the job is returned to its former position (line 14 in Algorithm 4.9).

The main loop of *RandomSearch* stops when either the predefined number of iterations has been reached, the time limit has been exceeded or a stopping event has been detected. The stopping event signals that some higher priority event arrived to the scheduler, thus the optimization is immediately stopped following the anytime approach.

RandomSearch is used for periodical optimization. It does not restrict itself to gap-filling since the moves are performed randomly, disregarding existing gaps. It allows us

Algorithm 4.10 *MoveJobRandomly*$(job, schedule_{best}, schedule_{new}, criteria)$

```
1: Permute the clusters' schedules to test them in a random order;
2: for $k := 1$ to $l$ do
3:   if cluster $k$ is suitable for the $job$ then
4:     $schedule_{new}[k]$ := place $job$ in the $schedule_{new}[k]$ on random position;
5:     if $AcceptCandidate(schedule_{best}, schedule_{new}, criteria)$ then
6:       return true;
7:     else
8:       $schedule_{new}[k] := schedule_{best}[k]$;
9:     end if
10:  end if
11: end for
12: return false;
```

to make more aggressive changes in the schedule than if we were limited to use only the existing gaps. Moreover, such random approach is very robust since we take no a priori assumptions concerning the objective functions. In another words, the fully random approach is not tailored to some specific criteria, thus remains very flexible. The two algorithms that follow were proposed for specific purposes. Here, the fully random approach has been replaced with simple strategies that helps to guide the solution technique toward expected behavior.

The complexity of *RandomSearch* algorithm is discussed in the following text. The algorithm works in a loop, which in the worst case is bounded by the number of *iterations*. In each pass of the loop a random job is found ($\mathcal{O}(n)$) and — in the worst case — it is subsequently moved onto a random position in each clusters' schedule ($\mathcal{O}(l)$). All of these l moves must be evaluated. The first evaluation can require at most $\mathcal{O}(n \cdot cpus)$ steps, while all remaining $l-1$ evaluations require only $l + cpus_k \cdot n_k + n$ operations, since only the changed $schedule_{new}[k]$ is evaluated[19]. Then, with the aid of Formula 4.11, the total evaluation time can be bounded by $\mathcal{O}\left(l^2 + cpus \cdot n + l \cdot n\right)$ steps. Together, the complexity of *RandomSearch* is bounded by Formula 4.22.

$$RS(...) = \mathcal{O}(iterations \cdot (n + l)) + \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n)\right) \quad (4.22)$$

Using the standard transformations [64], Formula 4.22 can be further simplified as is shown in Formula 4.23.

$$\begin{aligned} RS(...) &= \mathcal{O}\left(iterations \cdot (n + l + l^2 + cpus \cdot n + l \cdot n)\right) \\ &= \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n)\right) \\ &= \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n)\right) \\ &= \mathcal{O}\left(iterations \cdot l^2 + iterations \cdot cpus \cdot n\right) \end{aligned} \quad (4.23)$$

[19]See Formula 4.16 and related discussion for explanation.

4.7.2 Gap Search

Similarly as RS, *Gap Search (GS)* is a local search-based method working in a Hill Climb-like fashion. It is used to optimize the compressed schedule when early job completion appears, as discussed in Section 4.3. The main body of the procedure is identical to Algorithm 4.9. The difference lies in the strategy applied in the "job moving" procedure *MoveJobIntoGap* shown in Algorithm 4.11.

Algorithm 4.11 $MoveJobIntoGap(job, schedule_{best}, schedule_{new}, criteria)$

```
1: Permute the clusters' schedules to test them in a random order;
2: for $k := 1$ to $l$ do
3:   if cluster $k$ is suitable for the $job$ then
4:     $pointer := FindFirstGap(k, job)$;
5:     $schedule_{new}[k]$ := place $job$ in the gap in the $schedule_{new}[k]$ using the $pointer$;
6:     if $AcceptCandidate(schedule_{best}, schedule_{new}, criteria)$ then
7:       return true;
8:     else
9:       $schedule_{new}[k] := schedule_{best}[k]$;
10:    end if
11:  end if
12: end for
13: return false;
```

Since the main purpose of Gap Search is to repair schedule after early job completion, instead of moving jobs to random positions, the procedure moves the jobs into the earliest gaps (see lines 4 and 5). Early job completions may appear very frequently, thus Gap Search must be a quick and straightforward procedure. The success of this technique relies on the same principle as the *BestGap* policy. The gap filling strategy helps us to fill early gaps with suitable jobs. It guarantees that corresponding machines will not remain idle, thus available machine power will be utilized. In addition, once the selected job is removed from its position and moved into the gap, the job itself as well as remaining jobs in the schedule may often start their execution earlier, as they are shifted to an earlier start times using the applied schedule compression.

The discussion on the complexity of the Gap Search algorithm follows. In each loop a random job is found ($\mathcal{O}(n)$) and — in the worst case — it is subsequently moved to the earliest suitable gap on each of l clusters using the *MoveJobIntoGap* procedure. Together, these operations are bounded by $\mathcal{O}(n+g)$. Since $g \leq n+l$ (see Formula 4.7) these l steps can be bounded by $\mathcal{O}(n+l)$. Therefore, when all such l iterations are finished the total evaluation time is the same as it is for the RS algorithm, i.e., it is bounded by $\mathcal{O}\left(l^2 + cpus \cdot n + l \cdot n\right)$. Together, the complexity of Gap Search is bounded by Formula 4.24.

$$GS(...) = \mathcal{O}\left(iterations \cdot (n+l)\right) + \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n)\right) \quad (4.24)$$

Similarly as for the previous RS algorithm, the initial Formula 4.24 depicting the complexity of *Gap Search* can be further simplified as shown by Formula 4.25.

$$\begin{aligned} GS(...) &= \mathcal{O}\left(iterations \cdot (n + l + l^2 + cpus \cdot n + l \cdot n)\right) \\ &= \mathcal{O}\left(iterations \cdot l^2 + iterations \cdot cpus \cdot n\right) \end{aligned} \tag{4.25}$$

4.7.3 Tabu Search

Last optimization algorithm named *Tabu Search (TS)* has been proposed focusing on the specific problem involving job deadlines. It is used as an optimization procedure that accompanies the *BG-EDF* policy, presented in Section 4.6.2.

The algorithm is based on the Gap Search algorithm, which was found to be suitable for problems involving job deadlines [104, 105]. Thanks to the applied schedule compression, selected job as well as remaining jobs in the schedule may often start their execution earlier once the job is removed from its position and moved into a gap. It *increases the probability* that their deadlines will be met.

Unlike the Gap Search, the algorithm includes important feature that is typical for the *Tabu Search*-based algorithms [69, 137]. It is a short term memory called *tabu list* where few previously manipulated jobs are stored. If the algorithm is trying to move some job then this change is not allowed in case that this job is present in the tabu list. It has limited size and the oldest item is always removed when the list becomes full. Tabu list helps to protect the algorithm against short cycles where the same few jobs are repeatedly selected as the move candidates. Since this functionality requires some additional computational steps, it has not been applied in the original Gap Search. As discussed, Gap Search must be very fast to deal with frequent early job completions efficiently, thus its design was kept as simple as possible.

$TabuSearch(iterations, time\_limit, stop\_event, criteria)$ optimization algorithm is described in Algorithm 4.12. In each iteration, one *source* cluster's schedule is chosen randomly from the set of all clusters having at least one non-tabu job in their schedule (line 5). Then, random *job* is chosen in the *source* schedule. Selected job must not be in tabu list, i.e., $job \notin Tabu$. Once the job is selected, it is removed from its current position and an attempt to find a better $schedule_{new}$ — improving the values of the objective functions — is made (line 12) by the *MoveJobIntoGap* function (see Algorithm 4.11). If this attempt is successful the *MoveJobIntoGap* returns true and the $schedule_{best}$ is updated with the $schedule_{new}$ (line 13). Otherwise the move is rejected (line 15). Finally, the job is placed into the *Tabu* (line 17) — so that it cannot be chosen in the next few iterations — and a new iteration of the Tabu search starts. If in some iteration all schedules contain only tabu jobs it means that all clusters' schedules were explored, therefore the tabu list is emptied and another iteration starts (line 7)[20]. Again, the cycle continues until the predefined number of iterations or the given time limit is reached or some higher

[20]In the current implementation the tabu list has maximum size of 10 jobs. If all jobs in the *schedule* are in *Tabu*, it means that there are at most 10 jobs in the whole *schedule*.

priority event is detected (lines 3 and 18 respectively). Then, the $schedule_{best}$ is returned as the newly found solution (line 22).

Algorithm 4.12 *TabuSearch*$(iterations, time\_limit, stop\_event, criteria)$

```
1: $schedule_{best} := [cl\_sched_1, .., cl\_sched_l]$; $schedule_{new} := schedule_{best}$; $Tabu := \emptyset$;
2: $i := 0$; $run :=$ **true**;
3: **while** $(i < iterations$ **and** $run)$ **do**
4:    $i := i + 1$;
5:    $source :=$ select random $schedule_{new}[k]$ $(1 \leq k \leq l)$ with at least one $job \notin Tabu$;
6:    **if** $source = null$ **then**
7:       $Tabu := \emptyset$; (all jobs were tested, reset the tabu list)
8:       **continue**;
9:    **end if**
10:   $job :=$ select random $job$ from $source$ such that $job \notin Tabu$;
11:   remove $job$ from $source$;
12:   **if** $MoveJobIntoGap(job, schedule_{best}, schedule_{new}, criteria)$ **then**
13:      $schedule_{best} := schedule_{new}$; (updates the best so far found solution)
14:   **else**
15:      $schedule_{new} := schedule_{best}$; (move is rejected, $job$ is returned back);
16:   **end if**
17:   $Tabu := Tabu \cup job$; (removes the oldest item if $Tabu$ is full)
18:   **if** $time\_limit$ exceeded **or** $stop\_event$ detected **then**
19:      $run :=$ **false**;
20:   **end if**
21: **end while**
22: **return** $schedule_{best}$
```

TabuSearch algorithm has the same complexity as Gap Search algorithm since it is based on it. The difference is that in each loop a random *non tabu* job is being found. Luckily, the tabu list has fixed (constant) size of c jobs[21], therefore this operation is bounded by $\mathcal{O}(n \cdot c)$ which is in $\mathcal{O}(n)$. The complexity of *MoveJobIntoGap* remains the same as it is for Gap Search. After each iteration, the job being moved is added to the tabu list ($\mathcal{O}(1)$). Together, the complexity of *TabuSearch* is bounded by Formula 4.26.

$$\begin{aligned} TS(...) &= \mathcal{O}(iterations \cdot n \cdot c) + \mathcal{O}(iterations \cdot (n + l)) \\ &+ \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n)\right) + \mathcal{O}(iterations \cdot 1) \end{aligned} \tag{4.26}$$

As usually, Formula 4.26 can be further transformed to simplified Formula 4.27.

[21]Current implementation uses a tabu list of size 10, i.e., $c = 10$.

$$\begin{aligned} TS(...) &= \mathcal{O}\left(iterations \cdot (n+l)\right) + \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n + 1)\right) \\ &= \mathcal{O}\left(iterations \cdot (n+l)\right) + \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n)\right) \\ &= \mathcal{O}\left(iterations \cdot (l^2 + cpus \cdot n + l \cdot n)\right) \\ &= \mathcal{O}\left(iterations \cdot l^2 + iterations \cdot cpus \cdot n\right) \end{aligned} \tag{4.27}$$

4.8 Conclusion

In this chapter, the main principles of the proposed approach have been presented. We have applied *event-based approach* that manages the whole scheduling process. Beside that, also the application of *incremental, gap filling* and *anytime approach* has been emphasized, representing promising methods when seeking for fast, efficient and scalable solution. Two different techniques — policies and local search-based optimization — have been proposed to construct and optimize the schedule under various circumstances.

Both policies and optimization algorithms have been successfully used and published in our studies. In [104, 99, 105] the importance of incremental approach has been demonstrated using *BG-EDF* and *TS*. Here we were extending our initial results published in [98, 96]. In [103, 100] the performance of *BG* and *RS* was shown, while in [102] we have used *BG*, *RS* as well as *GS* when dealing with inaccurate processing time estimates.

Beside the general description of the proposed approach we have also presented implementation details covering all important aspects of the solution such as applied data structures, basic routines operating over the schedule as well as proposed scheduling algorithms. Moreover, the expected computational complexity has been discussed as well.

Chapter 5

Job Scheduling Simulator

This chapter presents the *Alea* job scheduling simulator [97, 101] which we have been developing since 2007. It discusses existing toolkits and simulators and describes the main design and performance characteristics of our simulator. Also, an experimental comparison with the selected existing toolkits and simulators is presented.

5.1 Motivation and Related Work

New proposed algorithms must be heavily tested and evaluated before they are applied in the real systems. Due to many reasons, such as the cost of resources, the reliability, the varying background load or the dynamic behavior of components, experimental evaluation cannot be mostly performed in the real systems. To obtain reliable results, many simulations with various setups must be performed using the same and controllable conditions that simulate different real life scenarios. This is often unreachable in the real Grid.

For this purpose many simulators have been developed. If properly designed, such simulators are very useful since different setups and different data sets can be used to evaluate existing or proposed solutions. While for some purposes an ad-hoc simulator is sufficient, there are also general Grid and cluster simulators allowing to simulate various scenarios and problems. Most of them are available as toolkits that have to be carefully modified and extended before they are suitable for the researcher's goals. The amount of such necessary work is usually quite large if the simulation outputs should be complex and reliable.

There are numerous simulators and toolkits that provide various functionality for simulations of the clusters, network and Grid environments. In the following text we mention the most popular and widely used toolkits and simulators designed to simulate Grid-like environments.

We start with the *MicroGrid* [123] which is rather an emulator than a simulator. It can be used for systematic study of the dynamic behavior of applications, middleware, resources, and networks. The MicroGrid uses the Globus Toolkit 2.2 API for its execution which allows to precisely emulate GTK 2.2 based systems. However such systems are completely outdated since the current version 5.0.3 uses different model based on the concept of web services.

The *Bricks* [165] simulates various scheduling schemes on a typical high-performance global computing systems. The Bricks can simulate various behaviors of global computing systems, especially the behavior of networks and resource scheduling algorithms. Moreover, its modular design allows to incorporate different scheduling algorithms, and it also allows incorporation of existing global computing components via its foreign interface.

The *BeoSim* [89] has been implemented for the purpose of studying multi-site parallel job scheduling algorithms in the context of a multi-cluster computational Grid. The Beosim can be driven either by synthetic workload or real workload. It also provides Java based visualization tool.

The *SimGrid* [115] is a C based toolkit used for the simulation and development of distributed applications in heterogeneous and distributed environment. The SimGrid solves different problems using different programming environments that constitutes different paradigms. The SimGrid's MSG tool is widely used for the basic evaluation and simulation of scheduling algorithms while other tools such as the GRAS and the SMPI are used for development and study of real applications. The SimDag provides functionalities to simulate parallel task scheduling with DAG (Direct Acyclic Graphs) workflow models.

The *Simbatch* [26], based on the SimGrid's MSG, allows to evaluate scheduling algorithms for batch schedulers. Neither the SimGrid nor the Simbatch offer integrated visualization output. However, both simulators may generate a trace that can be lately visualized through the Pajé [41] or the ViTE [39] visualization tools.

The *SimBOINC* [107] is another SimGrid based simulator. It is designed to simulate heterogeneous and volatile desktop Grids and volunteer computing systems. The SimBOINC simulates a client-server platform where multiple clients request work from a central server. The goal of this simulator is a testing of new scheduling strategies for BOINC (Berkeley Open Infrastructure for Network Computing) [9], and other desktop and volunteer systems. The characteristics of the client such as the speed, the availability of the workload or the network availability can all be specified in the simulation inputs.

While the SimGrid, the Simbatch and the SimBOINC are all based on the C programming language, all following simulators are based on the popular Java language [133]. While Java is usually less efficient than C, it allows easy development and effortless portability.

The *Monarc 2* [45] is a simulation framework whose aim is to provide a design and optimization tool for large scale distributed computing systems, with a focus on the forthcoming LHC (Large Hadron Collider) experiments at CERN. Although it incorporates simple scheduling module, the main goal of the Monarc 2 is to provide a realistic simulation of distributed computing systems, customized for specific physics data processing, and to offer a flexible and dynamic environment for the performance evaluation of a range of possible data processing architectures. The Monarc 2 extends the obsolete Monarc simulator [116], by improving its flexibility and performance.

The *GridSim* [163, 25] is a flexible, modular and universal Grid simulation toolkit with a very good documentation. It is written in Java on top of an event simulation library called *SimJava* [80]. Thanks to the Java language, the GridSim is a platform independent toolkit. The GridSim provides functionality to simulate the basic Grid environment and its behavior by providing simple implementations of common entities such as the computational resources or the users. It also allows to simulate simple jobs, the network topology, the data storage and other useful features. Provided implementations represent the base functionality and it is necessary to extend them when performing simulations with more complex requirements. This can be done by the implementation of new Java classes inheriting from the existing GridSim classes.

The GridSim is used by various researchers in their simulations. There exists decentralized scheduler [4] implemented in an obsolete version of the GridSim, but it does not support dynamic behavior of the system and it is not capable of simulating network topology or other recent features. The reason is the incompatibility between older and recent GridSim versions.

The *Grid Scheduling Simulator (GSSIM)* [110] based on the GridSim toolkit has been developed for several years and became publicly available in 2009. It should provide an easy to use Grid scheduling framework for enabling simulations of a wide range of scheduling algorithms in multi-level, heterogeneous Grid infrastructures. However, the GSSIM encounters some problems such as slow execution, weak scalability and poor visualization outputs. Moreover, the GSSIM is not compatible with the standard GridSim releases (including the latest GridSim 5) since it uses its own branch based on a modified version 4. Still, if GSSIM is optimized and made compatible with the GridSim, it will provide an interesting alternative, supporting more features than our simulator.

The *Alea* [97] has been developed since 2007. It is an extension to the widely used GridSim simulation toolkit [163]. It represents “ready to use” centralized scheduling system allowing to apply and compare various scheduling algorithms similar to those used in the production scheduling systems such as PBS Pro [88], LSF [191] or CCS [79]. The solution consists of the scheduler entity and other supporting classes which extend the original basic functionality of the GridSim. The main benefit of our solution is that the Alea allows immediate testing by inclusion of several popular and widely used scheduling algorithms such as FCFS, EDF [122], EASY Backfilling [156], Conservative Backfilling [52, 159], Flexible Backfilling [172], etc. Beside the queue-based algorithms, it also enables the use of algorithms that construct the schedule (scheduling plan) [79]. Over the time, many core improvements have been done concerning the design, the scalability and the functionality. Also, several new scheduling algorithms and objective functions were included as well as the support of additional job and machine characteristics [101]. The Alea now also provides complex visualization tool which supports an export of simulation results into several bitmap formats. The simulation speed and the simulator’s scalability have been significantly improved through the newly developed or redesigned classes. This covers a new memory-efficient job loader and a redesigned job allocation policy, which speeds up the whole simulation (see Section 5.4.2). Moreover, the support of standardized workloads formats has been included as well as the simulation of machine failures using the real life failure traces. All these features are closely discussed in Section 5.3.

To conclude this section, there are several different simulation toolkits that cover various aspects of Grid and cluster simulations. Sadly — based on our experience — several of them are not now usable[1] either due to the incompatibility (MicroGrid), major reconstruction (SimBOINC), or due to the authors' decisions (Bricks, BeoSim). From this point of view, the best choice for researchers is to use the SimGrid or its extension the Simbatch, the GridSim or an ad hoc simulator.

Before we proceed to the description of the main features of our simulator, let us briefly mention the main characteristics of the GridSim toolkit that is used by the Alea simulator.

5.2 GridSim Toolkit

The GridSim toolkit is a Java based toolkit that allows modeling and simulation of entities in parallel and distributed computing environments. It allows to simulate users, their applications, resources as well as schedulers or brokers. It provides a comprehensive facility for creating different classes of heterogeneous resources including computational machines and clusters, data storages or network topology. Both single processor and multi-processor machines managed by time or space shared schedulers can be simulated. The processing nodes within a resource (cluster) can be heterogeneous in terms of processing capability, configuration, and availability [25].

GridSim builds on top of SimJava [80] which provides basic infrastructure for event-based simulations. The most important parts of GridSim are those that simulate users' applications and active entities such as Grid resources, schedulers or system users. An active entity is an object having its own thread and the ability to send and receive events. Using events, active entities may communicate together and react accordingly. By default, any entity can directly communicate with all other entities. If more realistic approach is required, the GridSim allows to establish a network topology so the communication is routed through the simulated network.

Each job is represented by an instance of `Gridlet` class. Gridlets can be parsed (loaded) from a workload file (data set) using a `Workload` class. Gridlets are executed on the Grid resources. Grid resource is represented by the instance of `GridResource` class. It contains one or more machines that constitute, e.g., a computer cluster. Such resource is managed by the local scheduling policy. Current GridSim 5 supports several policies based on space sharing and time sharing paradigm. The allocation policy is implemented in a separate class, e.g., using `SpaceShared` class.

Simulated entities communicate through event-based message passing protocol. For each simulated event — such as the gridlet arrival or the gridlet completion — one message defining this event is created. It contains the identifier of the message recipient, the type of the event, the time when the event will occur, and the message data[2].

Once the gridlets are created they are submitted onto the resources. The submission can be performed either directly from the original `Workload` instance or through some scheduler. GridSim does not provide any implementation of such a scheduler. However, the `GridSim` class, representing a general communicating entity can be used as parent

[1]In April 2011.

[2]In case of, e.g., gridlet arrival such message looks like this: `(receiver_ID; GRIDLET_SUBMIT; simulation_time; gridlet)`.

class for the implementation of the scheduler.

5.3 Alea Overview

The Alea extends existing GridSim classes and also offers brand new classes to provide a "ready to use" job scheduling simulator. It has been designed with respect to our needs concerning simulation capability. This involves the ability to perform complex simulations that involves all major features of the studied problem as we have described in Chapter 2. To be more precise, Alea is able to simulate sequential and parallel jobs, computational clusters, specific job requirements as well as system dynamics such as machine failures or users' activity. It also supports various optimization criteria mentioned in Chapter 2.

Same as the GridSim, the Alea is an event-based modular simulator, composed of independent entities which implements the desired simulation functionality (see Figure 5.1). It consists of new centralized scheduler, the Grid resource(s) with the local job allocation

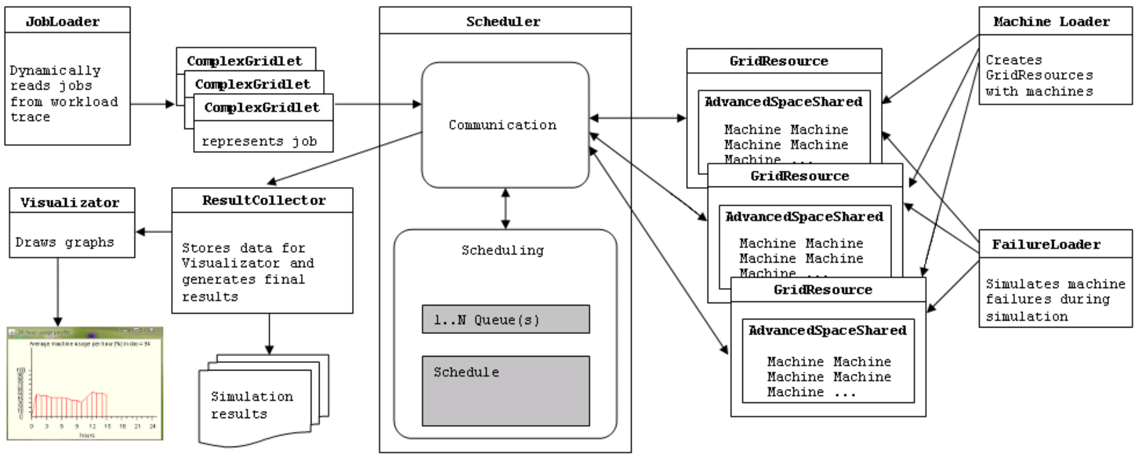


Figure 5.1: Main parts of the Alea simulator.

policy, the job loader, the machine and failure loader and additional classes responsible for the simulation setup, the visualization and the generation of simulation output. By now, the Grid users are not directly simulated, but the job loader entity can be used as a parent class for the future implementation of the Grid user. As in GridSim, simulator's behavior is driven by the event-based message passing protocol. The simulator is fully compatible with the latest GridSim 5 release since no changes were made in the GridSim package itself. All extensions were made by implementing child classes which extend the standard GridSim (parent) classes. Similarly, easy extension of current functionality is possible thanks to the object oriented paradigm used by the GridSim and the Alea. In the following text all important extensions on top of the GridSim package are mentioned and explained.

The simulation is initialized by the `ExperimentSetup` class which creates instances of the scheduler, the job and machine loader, the failure loader and other entities as required by the standard GridSim. The `MachineLoader` entity performs the initialization of the simulated computing environment. It reads the data describing the machines from a file and creates Grid resources accordingly. GridSim itself does not provide such functionality and machine parameters must be "hardcoded" in the source java file. Our solution allows to change machines' parameters without the re-compilation of the whole program.

The `JobLoader` reads the file containing the job descriptions and creates jobs' instances dynamically over the time. The `JobLoader` supports several trace formats including the Grid Workloads Format (GWF) of the Grid Workloads Archive[3] and the Standard Workloads Format (SWF) of the Parallel Workloads Archive[4]. When the simulation time is equal to the job submission time the `JobLoader` sends the job to the scheduler. The `JobLoader` reads only one job at a time to limit the required memory space, and to allow the use of very large workload traces. This approach is necessary since the original GridSim's `Workload` entity reads all data at once which — for large workloads — results in simulation fails due to the Java's Out-Of-Memory error. The job itself is represented by the instance of the `ComplexGridlet` class. The GridSim provides only trivial implementation of a job in its `Gridlet` class. The `ComplexGridlet` extends this class, allowing to simulate more realistic scenarios where each job may require additional properties such as the deadline, the estimated runtime, the specific machine parameters, and other real life based constraints as was discussed in Chapter 2.

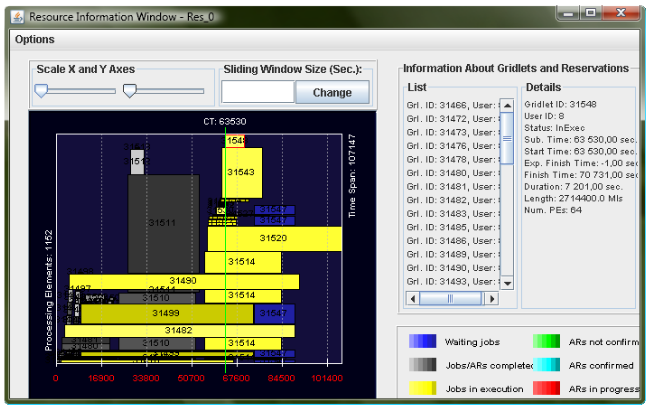
The `FailureLoader` reads the file containing descriptions of machine failures. Once the simulation time reaches the failure start time, the appropriate machine is set to be failed, killing all jobs being currently executed on that machine. When the failure period passes the machine is restarted. As discussed in Sections 4.2.3 and 4.2.4, machine failures can be used to simulate addition of a new machine or permanent machine removal.

As in the GridSim, the resource is represented by the `GridResource` instance and is managed by the local scheduling policy. Unfortunately none of the currently provided policies support the execution of parallel jobs and the simulation of machine failures at the same time. Also the co-allocation of several machines for job execution is not available. Therefore, new allocation policy called `AdvancedSpaceShared` was developed for the Alea based on the GridSim's `SpaceShared` policy. It allows to execute both sequential and parallel jobs on the specified number of CPUs using the space sharing processor allocation policy (see Section 2.3). It enables to simulate more realistic scenarios involving the parallel jobs as well as the simulations of machine failures. In addition, it includes more efficient implementation of the message passing which allows significant improvements in the simulation speed (see Section 5.4).

The newly developed `Visualizator` class generates the simulation's graphical output. The Gridsim itself enables visualizations displaying the process of allocating jobs onto the machines over time as can be seen in Figure 5.2. The `Visualizator` extends this functionality by displaying additional information useful for tuning and debugging of scheduling algorithms. So far, several outputs covering different objectives are supported and displayed. Those are the overall utilization of resources, the cluster utilization, the number

[3]`http://gwa.ewi.tudelft.nl`

[4]`http://www.cs.huji.ac.il/labs/parallel/workload/`



Grl. ID: 31466, User:
Grl. ID: 31472, User:
Grl. ID: 31473, User:
Grl. ID: 31476, User:
Grl. ID: 31478, User:
Grl. ID: 31480, User:
Grl. ID: 31481, User:
Grl. ID: 31482, User:
Grl. ID: 31483, User:
Grl. ID: 31485, User:
Grl. ID: 31486, User:
Grl. ID: 31489, User:
Grl. ID: 31490, User:
Grl. ID: 31493, User:

Gridlet ID: 31548
User ID: 8
Status: InExec
Sub. Time: 63 530,00 sec.
Start Time: 63 530,00 sec.
Exp. Finish Time: -1,00 sec
Finish Time: 70 731,00 sec
Duration: 7 201,00 sec.
Length: 2714400.0 MIs
Num. PEs: 64

Figure 5.2: Visualization of the job allocation procedure as provided by the GridSim.

of waiting and running jobs and the number of requested, utilized and available CPUs. Beside that, also the percentage of failed and running CPUs per cluster can be displayed. The `Visualizator` may work in two different fashions. In the first case, the visualization is generated continuously as the simulation proceeds (see Figure 5.3). In the second case, the graphs are generated when the simulation is finished, using the simulation results as an input. Results are continuously collected by the `ResultCollector`. When the simulation completes, the `ResultCollector` stores them into *csv* files that can be easily used as an input for other tools (Calc, Excel, Spreadsheet, etc.) and it also saves generated graphs into a preferred bitmap file (png, jpg, bmp, gif).

The key part of the Alea is the newly developed `Scheduler` entity, which is described in the following section.

5.3.1 Scheduler Entity

The `Scheduler` is the main part of the Alea. Its behavior is driven by events and corresponding messages, following the approaches proposed in Sections 4.1 and 4.2. Using the events, the `Scheduler` communicates with the `JobLoader` (job arrivals), with the `GridResources` (job submission/completion and failure detection) and with the `Result-Collector` (periodical result collection). Also the internal events are used to manage the scheduling process (see Section 4.2). The `Scheduler` is responsible for performing scheduling decisions according to the selected scheduling policy. It was designed as a modular, extensible entity composed of three main parts (see Figure 5.4) which are discussed in the following text.

The first part stores dynamic information concerning the Grid resources (see Figure 5.4 bottom right). For each `GridResource`, one `ResourceInfo` object is created that holds up-to-date information regarding the current resource status. It stores information about jobs currently in execution, about jobs that are planned for execution (if the schedule is being constructed) and it implements various functions that help to compute or predict various values, e.g., the next free slot available for specific job, etc.

The second part is responsible for the communication with the remaining simulation

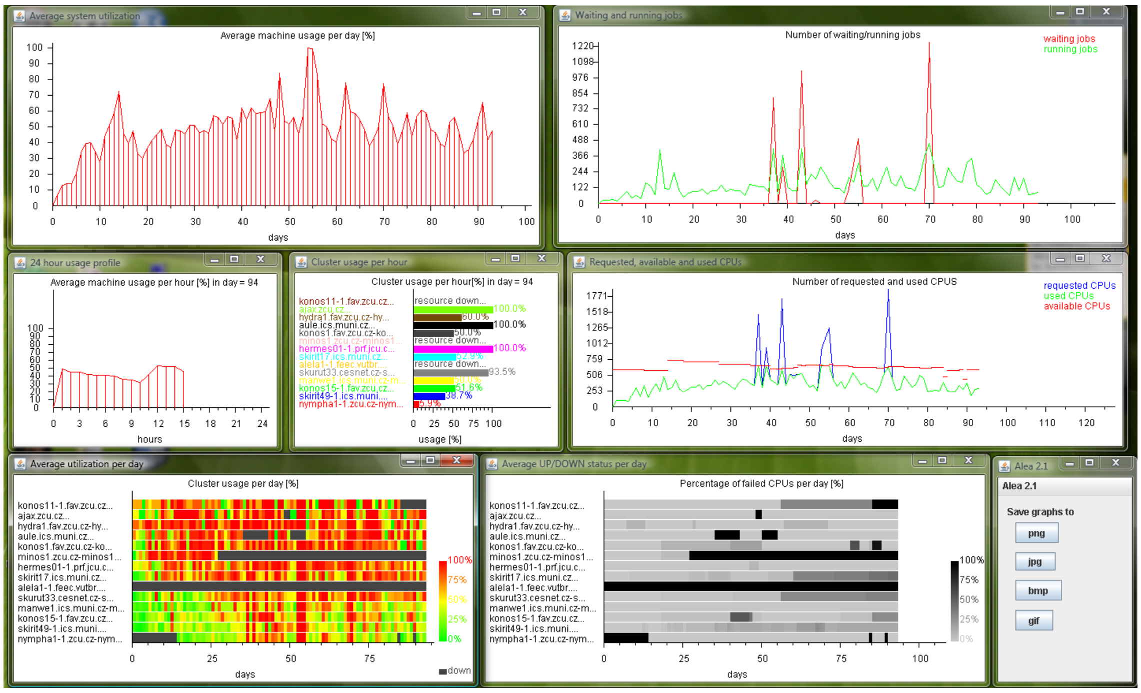

Average system utilization
Average machine usage per day [%]
days
Waiting and running jobs
Number of waiting/running jobs
waiting jobs
running jobs
24 hour usage profile
Average machine usage per hour [%] in day = 94
hours
Cluster usage per hour
Cluster usage per hour[%] in day = 94
usage [%]
Requested, available and used CPUs
Number of requested and used CPUS
requested CPUs
used CPUs
available CPUs
Average utilization per day
Cluster usage per day [%]
Average UP/DOWN status per day
Percentage of failed CPUs per day [%]
Alea 2.1
Save graphs to
png
jpg
bmp
gif

Figure 5.3: Visualization interface of the Alea during the simulation.

entities (see Figure 5.4 top). It accepts incoming messages (events) and reacts accordingly (see Section 4.2). Typically, the `Scheduler` receives newly incoming job from the `JobLoader`. It takes the incoming job and places it into the queue or schedule according to the applied scheduling algorithm. Next, new scheduling round is performed and an attempt to submit jobs present in the queue or schedule is performed. If some resource is available and a suitable job is selected, it is submitted to the resource where it will be executed. Moreover, appropriate scheduler's `ResourceInfo` object is updated according to the new situation. Once some job is completed, it is returned to the `Scheduler` and the `ResourceInfo` object is updated as a result of the new state. Similar update is performed when some machine fails or restarts. Next, a new scheduling round is started. The cycle finishes when no new job arrivals appear and all submitted jobs have been completed. Then the simulation ends and the results are stored into the output files.

The last part of the `Scheduler` contains implementations of several popular and widely used scheduling algorithms (see Chapter 3). Since the recent research in the area of Grid and cluster scheduling focuses on both queue and schedule-based techniques we support both of them. Concerning the queue-based techniques following algorithms are implemented in the `Scheduler` entity: First Come First Served (FCFS), Earliest Deadline First (EDF) [122], EASY Backfilling (EASY) [156], Conservative Backfilling (CONS) [52, 159], Flexible Backfilling (Flex-BF) [172], and a PBS-like (PBS) multi-queue and priority-based scheduling algorithm (see Algorithm 3.7 in Section 3.3.4). Schedule-based techniques use a schedule — instead of a queue(s) — to store the jobs. In this case, each job is placed into

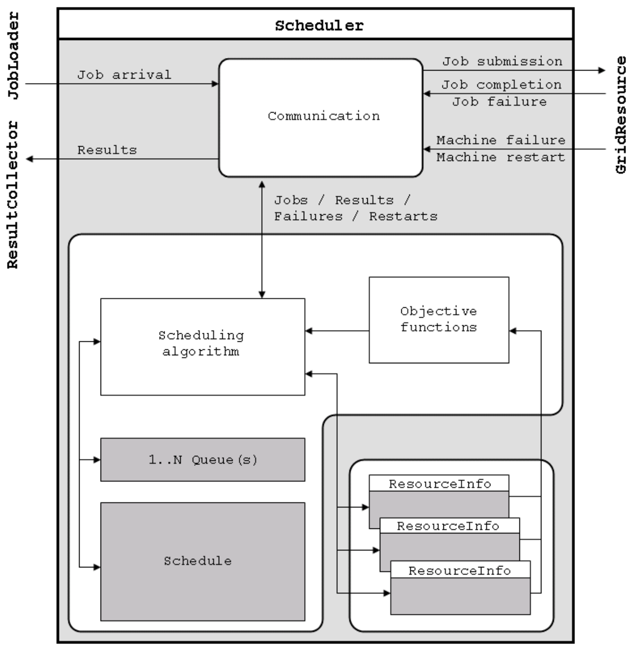


Figure 5.4: Main parts of the Scheduler entity.

the schedule upon its arrival which defines its expected start time, expected completion time and the target machine(s). The use of the schedule allows to use advanced scheduling and optimization algorithms [138] such as the local search-based methods [78, 68]. These techniques are represented by the Best Gap — Earlier Deadline First (BG-EDF) and Best Gap (BG) [104, 105] policies (see Section 4.6) and *local search*-based optimization routines Random Search (RS) [103], Gap Search (GS) [102] and Tabu Search (TS) [104, 105] (see Section 4.7).

Several objective functions (see Section 2.6) are supported which can be used for decision making or optimization. During the simulation, the `Scheduler` is capable of collecting various data such as the number of waiting and running jobs, current machine utilization, and the information related to all other objectives which were discussed in Section 2.6. Once the simulation is finished, output files containing these data are generated. Moreover, selected objectives can be used as an input for either the "on the fly" or the "post mortem" visualization provided by the `Visualizator` graphical tool.

5.3.2 Extensibility

The Alea can be easily extended thanks to the adopted object oriented paradigm. The simulator is modular, meaning that different functionality is implemented in different classes. Also, the crucial `Scheduler` class is divided into separate parts. Thus, a new scheduling algorithm or a new objective function can be added through the extension or the modification of the existing classes, using the provided data structures and interfaces. Similarly, if a new job property or a new job type is requested, only the `ComplexGridlet` shall be extended or modified, leaving other classes intact. Therefore, all changes are encapsulated and the development is straightforward.

5.4 Performance of the Alea Simulator

In this section we demonstrate the simulation capability of the Alea simulator. We also briefly mention simulator's performance, focusing on the speed and the scalability with respect to the GridSim based GSSIM simulator and the original GridSim toolkit. Finally, the correctness of the simulator is mentioned. This section is a summary of the results and deep performance analysis presented in our recent paper on the Alea simulator [101].

5.4.1 Simulation Capability

Let us start with a brief demonstration of the simulation capabilities of the Alea simulator. In fact, especially the usefulness of visualization interface will be presented here. For this demonstration, we have used the complex data from the Czech national Grid infrastructure MetaCentrum[127], that allow us to perform very realistic simulations, involving all characteristics discussed in Sections 2.4 and 2.5, such as the use of precise information concerning the machine parameters, the information about machine failures and restarts, or the specific job requirements concerning the target machine properties [103]. The experiment involved 103,656 jobs that were originally executed during the first five months of the year 2009 on 14 clusters having 806 CPUs[5].

As other simulators, Alea stores main simulation results into output files. However, it also provides complex visualization interface as shown in Figure 5.3. As we demonstrate now, these graphical outputs can be very useful, helping the user to understand and compare the functionality of different scheduling algorithms. For demonstration, two basic scheduling algorithms were used: FCFS and EASY Backfilling (EASY). Figure 5.5 presents graphs depicting the weighted machine usage per cluster (left) and the number of waiting and running jobs per day (right) as generated by the Alea's visualization interface during the experiment. These graphs nicely demonstrate major differences among the algorithms, allowing the user to understand their behavior. Concerning the machine usage — as expected — FCFS generates very poor results. FCFS is not able to utilize available resources when the first job in the queue requires some specific and currently unavailable machine(s). At this point, other "more flexible" jobs in the queue can be executed increasing the machine utilization. This is the main goal of the EASY Backfilling algorithm. As we can see, EASY is able to increase the machine usage by using the backfilling approach.

In case of the second criteria, similar reasons as in the previous example caused that FCFS is not able to schedule jobs fluently, generating huge peak of waiting jobs during the time (see Figure 5.5 top right). For the same reason, the resulting makespan is also much higher than in the remaining algorithm (by 60 days). EASY significantly reduces the number of waiting jobs through the time.

[5]Detailed description of MetaCentrum data set is presented in Section 6.1.1.

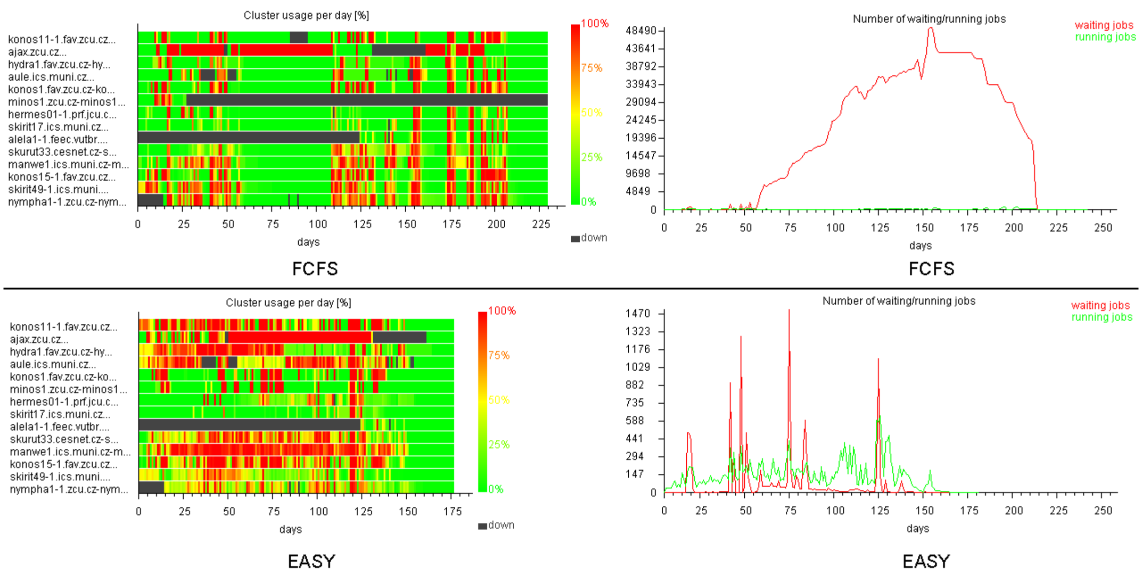


Figure 5.5: Comparison of FCFS and EASY using complex data set.

As was demonstrated, the visualization output is very useful. It may help to understand and to compare the scheduling process of different scheduling algorithms. In case that there is some problem with the studied algorithm, visualization can be very helpful when finding possible explanation. Last but not least, graphical outputs can be stored as bitmap files and later used, e.g., for educational or scientific purposes.

5.4.2 Simulator Performance

Beside the general suitability, we have also studied the performance of our simulator with respect to other GridSim based solutions. In this case the speed and scalability of the simulator has been evaluated. Detailed description of this experiment has been given in our paper [101]. For the purpose of this thesis we briefly recapitulate the findings presented in the paper. All experiments were performed on the Intel Core2 Duo 2.4 GHz PC with 4 GB of RAM. Unless otherwise indicated, the JVM (Java Virtual Machine) was limited by 1 GB of available RAM.

In the first experiment, the memory requirements of Alea's `JobLoader` entity has been compared with the memory requirements of the original GridSim's `Workload` solution. As we already mentioned in Section 5.3, the GridSim provides the `Workload` class to read job descriptions from the workload file. However, the `Workload` reads all jobs *before* the simulation starts, therefore all jobs have to be stored in the RAM, which may be rather inefficient for very large data sets. On the other hand, our solution using the `JobLoader` reads and creates jobs "on the fly" as simulation proceeds, meaning that only currently running and waiting jobs are stored in the RAM at any moment, allowing to maintain memory requirements in a decent level. In the experiment, we have demonstrated that the original GridSim's solution is not suitable for large data sets since it quickly consumes all available RAM. For evaluation, we have used the largest publicly available workload trace that covers 1,195,242 jobs[6]. When the Java's Virtual Machine (JVM) memory limit was set to be 1, 2 or 3 GB, respectively, GridSim's `Workload` solution was able to load at most 25.5%, 51.4% or 77.4% jobs, respectively. Then it failed due to the Out-Of-Memory error while our `JobLoader` solution performed smoothly thanks to the memory efficient "on the fly" approach. It requested at most 850 MB of RAM (during few load peaks). The results of this experiment are shown in Figure 5.6.

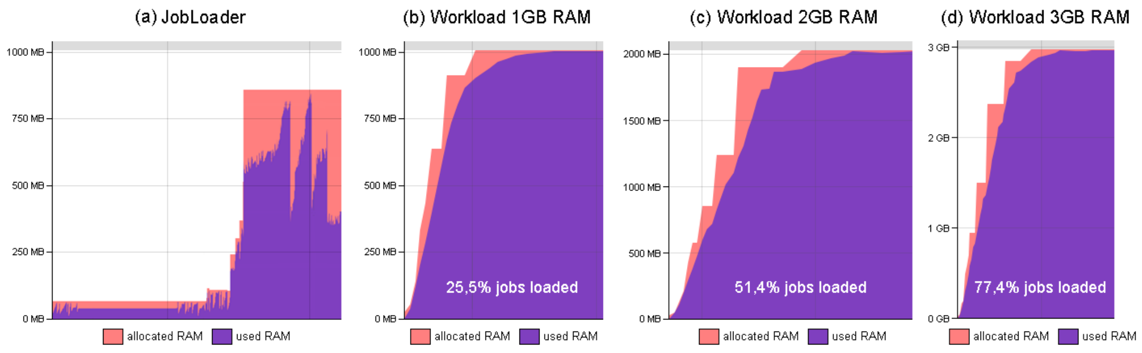


Figure 5.6: Experimental results for the Alea and the GridSim.

Next, we have compared our `AdvancedSpaceShared` policy and the original GridSim's `SpaceShared` policy with respect to simulation speed. These policies are used to simulate job execution on a cluster, and significantly influence the speed of the whole simulation. For the `SpaceShared`, the more jobs were executed the slower the simulation was. We managed to eliminate this issue by implementing a more efficient message passing model in our `AdvancedSpaceShared` policy. Figure 5.7(a) demonstrates the difference between the original `SpaceShared` and the new `AdvancedSpaceShared` policy. The y-axis is in logarithmic scale. In this experiment, either the original `SpaceShared` or our `AdvancedSpaceShared` was used in the Alea and the number of successfully completed jobs in one day (86,400 seconds) was measured. Our solution have simulated all 1,195,242 jobs in just 125 minutes, while the original GridSim's solution slowed down, finishing only 96,000 jobs within 24 hours.

Finally, we have compared the Alea and the GridSim based GSSIM simulator through an experimental execution focusing on the simulation speed[7]. In this case, two different criteria were taken into account — the amount of time needed to execute the experiment and the RAM requirements of the simulator. For the given JVM's RAM limit being 1 GB, the maximum number of jobs that the GSSIM was able to simulate was 15,000 as we realized through several experiments. For larger data, the GSSIM's execution always finished with an Out-Of-Memory error. With this setup, using FCFS as the applied scheduling algorithm, we compared the GSSIM with the Alea focusing on the simulation speed and memory requirements.

[6]It is the SHARCNET data set from the Parallel Workloads Archive (PWA). PWA workload traces are closely discussed in Chapter 6, Section 6.1.2.

[7]GridSim does not provide any scheduling algorithms, thus was not included in this case.

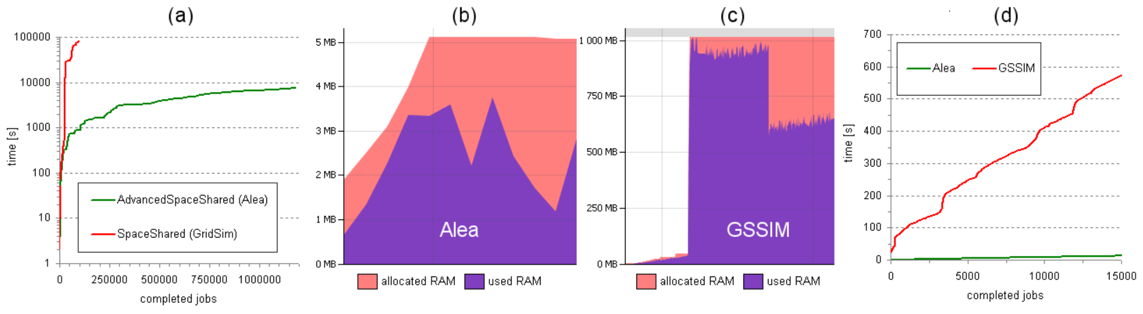


Figure 5.7: Simulation time with the `SpaceShared` and the `AdvancedSpaceShared` policy (a) and results for the Alea and the GSSIM (b–d).

There was a large difference concerning the RAM requirements. The Alea (see Figure 5.7(b)) required less than 4 MB of memory, while the GSSIM needed all dedicated memory (1,024 MB) as is shown in Figure 5.7(c). Figure 5.7(d) shows the execution time of both simulators with respect to the number of completed jobs. Clearly, Alea is much faster than the GSSIM during the whole simulation. Moreover, due to the Java's Out-Of-Memory error, the GSSIM's graph-creating plugin failed to generate any graphical output for every setup that involved more than 100 jobs. In general terms, Alea was faster and less memory demanding than the GSSIM[8].

Beside the previous experiments, there are also other known results concerning the GridSim scalability [42]. Basically the limitation of the GridSim is based on the applied thread model which limits the number of communicating entities to approximately 10,000. Since the Alea uses the GridSim, the same limitations also applies. We are aware that for some types of simulations such limitation may be severe. However, the Alea was designed to simulate scheduler's behavior rather than the behavior of thousands of Grid users. From this point of view, the ability to simulate large number of jobs and machines is necessary. Job representation is not a problem since it is a simple object, not using a thread. On the other hand `GridResource` representing one computer cluster requires four threads. Still, in the real world there are usually dozens or at most hundreds of clusters which fit fine within the GridSim limitations. It is necessary to notice that a machine within a `GridResource` is a simple object — not a thread — thus hundreds of machines placed within one `GridResource` require no additional thread.

5.4.3 Correctness of the Simulator

The correctness of the simulator was checked by analyzing the implementations of algorithms and simulation outputs. First, the implementations of scheduling algorithms were checked against their known pseudo-codes [180, 131, 172]. Next, the simulation outputs of existing algorithms such as FCFS, EDF, EASY or Conservative Backfilling were compared with the known results from the literature. In case of Flexible Backfilling, both the

[8]We have discussed our findings with the GSSIM developers who informed us that a new GSSIM version should solve the problems we have encountered.

implementation details and the simulation outputs were directly checked by the authors of this algorithm [105]. Proposed policies and optimization algorithms were checked through several experiments where the expected behavior (algorithm's specification) was compared with the simulation trace and simulation outputs. Also the graphical simulation output played an important role when analyzing the implementation. When some anomaly was identified, the code was traced and analyzed using classical debugging techniques and the implementation was fixed accordingly.

5.5 Summary

In this chapter, we presented the platform independent Alea scheduling simulator which is an extension of the popular GridSim toolkit. The goal of the simulator is to offer an easy way for the researchers to study, modify and extend various scheduling algorithms. Unlike the GSSIM or the solution described in [4], the Alea stays fully compatible with the original GridSim, which is very important for the future development. It was shown that it significantly outperforms the GridSim based GSSIM simulator by means of simulation speed and scalability. Moreover, the newly developed `JobLoader` and the `AdvancedSpaceShared` solutions replacing the original GridSim's `Workload` and the `SpaceShared` solutions lead to better performance covering both the scalability (`JobLoader`) and the simulation speed (`AdvancedSpaceShared`). Provided functionality covers typical researchers' requirements involving "ready to use" simulator that includes full implementations of common scheduling algorithms and supports common objective functions. Moreover, the visualization interface allows faster debugging and tuning of the studied algorithms as well as direct export of the simulation results into the bitmap and the *csv* files that can be easily used later. We also received positive feedback from several researchers from around the world who found using it very helpful[9].

The Alea, including the sources and full documentation, can be downloaded from `http://www.fi.muni.cz/~xklusac/alea`.

[9]Since 2007, we are aware of 28 foreign Alea users. Within our faculty, the Alea has been used by 5 students so far.

Chapter 6

Evaluation

In this chapter, the suitability and the performance of the proposed approach is evaluated through several experiments. The goal is to demonstrate how the applied techniques are able to deal with various application scenarios. To be more precise, the evaluation demonstrates how the proposed scheduling techniques perform when different problems are considered. For this purpose, different data sets are used during the experiments. All such experiments were performed using the *Alea* simulator described in Chapter 5.

This chapter is organized as follows. We start with a description of all data sets that have been used to evaluate the proposed techniques. Then we closely describe the experimental setup, the experimental test bed, applied optimization criteria and scheduling algorithms. Four major experiments follow, demonstrating the performance of our solution.

The experiments evaluate the major characteristics of the proposed solution using different problems, i.e., using different data sets and different objective functions. The first experiment demonstrates the importance of *complete data sets* for realistic experiments, using a complex data sets from the MetaCentrum [127]. Using it, we demonstrate that more complex simulation data sets constitute more complex scheduling problem where the benefits of advanced optimization techniques clearly appear. Beside that, the experiment also demonstrates the suitability of proposed schedule-based techniques by means of solution quality as well as acceptable runtime of proposed algorithms.

The second experiment focuses on the problem involving a specific QoS-related *deadline* parameter. It demonstrates that the proposed solution can be successfully applied when solving problems with added "non standard" parameters. It also shows the suitability of applied incremental approach with respect to solution quality and algorithm runtime.

The experiments demonstrate how the proposed approaches allow to achieve the desired performance involving good quality of the solution and acceptable runtime (scalability) as well as the flexibility concerning different objective functions. In this case, especially the suitability of the applied event-based and incremental approach is analyzed, together with the performance of the proposed policies and optimization techniques.

While the first two experiments use precise processing time estimates, the third experiment is dedicated to the problem of *inaccurate job processing time estimates* (see

Sections 3.3.5 and 4.3). We show how our algorithms as well as existing algorithms behave with respect to the decreasing accuracy of job processing time estimates. In this experiment we measure the influence of such an inaccuracy on the overall performance of both existing and proposed techniques. For our algorithms, the suitability of "correction techniques" proposed in Section 4.3 is evaluated.

The last experiment analyzes the actual *algorithm runtime* with respect to the expected computational complexity discussed in Chapter 4.

6.1 Data Sets

In this section we present all data sets that have been used to evaluate the proposed solution. Except for the last one, they all come from the workload logs of real systems.

6.1.1 Data Sets from MetaCentrum

MetaCentrum [127] is the Czech National Grid Infrastructure. It coordinates and operates the computational and storage resources in the Czech Republic. We were allowed to use the existing workload logs to prepare two real life-based data sets. The first data set called *MetaCentrum'08* is based on the data coming from the year 2008. The second data set *MetaCentrum'09* is based on the data collected during the first five months of 2009. MetaCentrum'08 contains trace of 187,370 jobs while MetaCentrum'09 contains 103,620 jobs. The specific job requirements (see Section 2.4) are included for both data sets. Beside that, the detailed clusters' descriptions with the information about machine architecture, CPU speed, memory size and the supported properties are provided. For MetaCentrum'08 there are 13 clusters (610 CPUs) while MetaCentrum'09 represents 14 clusters (806 CPUs). Also, the list of available queues including their priorities and associated time limits is provided. Moreover, the trace of machine failures and the description of machines temporarily dedicated for special purposes is provided as well. In case of MetaCentrum'08, this trace is rather incomplete, since the monitoring system was not fully operational at that time.

The average utilization of MetaCentrum varies per cluster. Interpretation of the average utilization strongly depends on information about dedicated machines which are not involved in processing of all jobs. Instead, these machines are dedicated to resource owners. Even more, jobs that are processed on dedicated machines are often submitted outside of the standard PBS Pro submission interface. To handle this issue, we simulate neither dedicated machines nor their jobs and we focus strictly on the problem involving machine failures and specific job requirements. Based on the selection of dedicated machines, the overall machine utilization has corresponded to approximately 43% (MetaCentrum'09) and 36% (MetaCentrum'08) in our experiments. The lower machine usage of MetaCentrum'08 is related to the incomplete information regarding failed and dedicated machines. Nowadays much higher load is reported since the information about dedicated machines are more precise.

To conclude, both data sets represent exhaustive description of the real behavior of the system, since detailed job, machine and queue parameters are provided as well as information concerning down times of the system resources. Even though the average utilization is rather low, these valuable information make this data very suitable for complex experiments [103, 100]. Moreover, due to the gracious help of the MetaCentrum staff, we were allowed to analyze the main features of the actual MetaCentrum scheduler and "reconstruct" it in our own work as was shown in Section 3.3.4, where the PBS-like algorithm has been presented. Together, it allows us to simulate the Grid behavior in a realistic fashion. The MetaCentrum'09 data set is publicly available at `http://www.fi.muni.cz/~xklusac/workload`.

6.1.2 Grid Workloads Archive and Parallel Workloads Archive

Two main publicly available sources of cluster and Grid workloads exist. Those are the Parallel Workloads Archive (PWA) [53] and the Grid Workloads Archive (GWA) [50]. There are two major differences between them. First of all, PWA maintains workloads coming from one site or cluster only (with few exceptions), while each workload in the GWA covers several sites. Second, the Grid Workloads Format (GWF) [84] is an extension to the Standard Workloads Format (SWF) [30] used in the PWA, reflecting some Grid specific job aspects. For example, each job in the GWF file has an identifier of the cluster where the job was executed. Moreover, the GWF format contains several fields to store specific job requirements. However, none of the six currently available traces uses them. Since SWF format does not support specific job requirements, it is natural that these are not included in the logs[1]. These archives also often lack detailed and systematic description of the Grid or cluster resources where the data were collected.

PWA currently contains 26 different workload logs while GWA contains 6 logs[2]. In our evaluation, we use four logs from PWA. Those are SDSC SP2, HPC2N, SDSC DataStar and LLNL Thunder [53]. If available, the recommended "cleaned" versions are always used. These logs were selected for several reasons. First of all, they all contain users' processing time estimates, that allow to simulate scenarios where the estimates are (highly) inaccurate. Second, these logs include identifiers of the job owners which allow to measure fairness. Moreover, they all contain sufficiently large amount of jobs (73,496–527,371) and represent systems of various sizes, starting with rather small systems having 128 and 240 CPUs (SDSC SP2 and HPC2N) then continuing to large systems of 1,664 and 4,008 CPUs (SDSC DataStar and LLNL Thunder). Last but not least, all these logs represent systems with reasonably high utilization ($> 60\%$). Detailed descriptions of these logs are available in the Parallel Workloads Archive [53].

So far, the Grid Workloads Archive provides only 5 logs that are not already provided by the PWA. Sadly, three of them contain only sequential jobs, thus two candidates remained[3]: Grid'5000 and DAS-2. However, these logs are not very useful, since their overall utilization is very low — 17% and 10% respectively. As we observed [103], in such case jobs are usually executed immediately upon their arrival since lots of resources are

[1]It is quite probable that in case of PWA there were no specific job requirements since the original systems mostly consisted of a single, uniform cluster.

[2]In April 2011.

[3]If all jobs are sequential (non-parallel), then most scheduling algorithms would more or less follow the FCFS approach.

free at that moment. Clearly, in such situation even simple algorithms would deliver acceptable solution. Therefore, Grid'5000 and DAS-2 are not used during the evaluation.

6.1.3 Synthetic Data Sets with Deadlines

The last data set used in our evaluation is a synthetic one. It has been generated with a generator that has been implemented by our Italian colleagues from the *Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" – Consiglio Nazionale delle Ricerche (ISTI-CNR)*, with whom we have cooperated within the *European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies (CoreGRID)* [36].

The reason for using this generator is that it allows to generate jobs having *deadlines*. While the deadline parameter is widely used in the literature [4, 3, 166, 77, 43], as far as we know, there are no publicly available data sets that would involve job deadlines. Therefore, we found this generator very useful. Beside our own [104, 99, 96] or joint works [105, 98], these data sets were also used by the authors of the generator in their own works [172, 27].

Using five different setups of the generator, five data sets have been generated simulating different scenarios. Different data sets have different inter-arrival times between the jobs [27]. For these five data sets, the average inter-arrival time goes from 1 to 5 seconds. According to the job inter-arrival times a different system load is generated through a simulation. The smaller this time is, the greater the system load is. The first data set produces a very high load due to the frequent job arrivals that — on average — appear every second. On the other hand, the last set generates lower load since the average inter-arrival time is increased to five seconds. Figure 6.1 shows the number of requested CPUs in time for different data sets, depicting different system load. When the average inter-arrival time is low, jobs arrive frequently and the number of requested CPUs is very high (y-axis). The green line depicts the number of available CPUs in the system and the x-axis depicts simulation time. Clearly, the number of requested CPUs at the given moment depends also on the applied scheduling algorithm. This figure is based on the data obtained when executing FCFS algorithm.

Each data set represents a stream of 3,000 jobs. It also contains description of 150 machines. Machines may have different numbers of CPUs and different speeds. The generator uses uniform distribution to generate job and machine parameters. Following ranges are used in the data sets: job execution time (500–3,000) seconds, 70% of jobs have the deadline, number of CPUs required by a job (1–8), number of CPUs per machine (1–16), machine speed (200–600). The job inter-arrival times are generated according to a negative exponential distribution.

To obtain reliable results, twenty instances of each of those five data sets have been generated using different seeds in the generator. Together, $5 \cdot 20 = 100$ instances with different job and machine parameters have been generated and used in the evaluation.

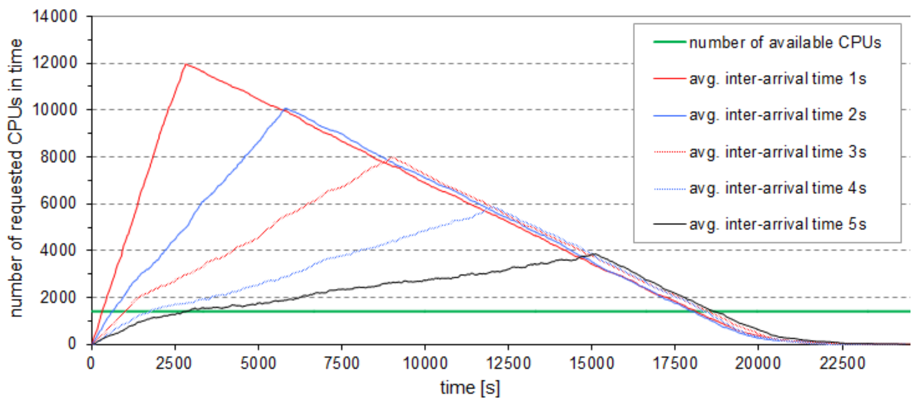


Figure 6.1: Number of requested CPUs in time for different synthetic data sets.

6.2 Experimental Evaluation

In this section we show and discuss the performance of the proposed schedule-based techniques with respect to several existing scheduling algorithms. The evaluation uses various data sets and objective functions to simulate different scheduling problems. It is divided into four major parts, showing the performance and flexibility of the proposed solution using four different problems. As already mentioned, the first experiment introduces our techniques and demonstrates the importance of complete data sets. The second experiment shows that the proposed techniques can be extended to solve problems with specific optimization criteria, i.e., to minimize the number of late jobs, while the third experiment focuses on the problem of inaccurate processing time estimates. The last experiment compares the actual algorithm runtime of the proposed algorithms with the expected computational complexity. We start the discussion with a simulation setup description.

6.2.1 Simulation Setup

All simulations were performed using the GridSim [163] based Alea simulator [101] that has been described in Chapter 5. The first and the third experiment have been computed using six nodes of the *quark* computer cluster in MetaCentrum. The first four nodes each contained 4 Intel Xeon CPUs running at 3.00 GHz, having 7.1 GB of shared RAM. The remaining two nodes each contained 16 Intel Xeon CPUs running at 2.93 GHz, having 16 GB of shared RAM. The second experiment has been computed using an Intel QuadCore 2.6 GHz desktop machine with 2 GB of RAM. The last experiment have been computed on an Intel Xeon 3.0 GHz desktop machine with 12 GB of RAM. All nodes and machines have been fully dedicated to us during the experimental phase. Since the experimental platform is not uniform the resulting values of algorithm runtime for different experiments are not directly comparable. However, within a single experiment, such comparison can be done since all results of such an experiment were obtained on machines having the same parameters. All experiments that involved some stochastic mechanism were repeated 10 times and their results were averaged[4].

Depending on the studied problem, appropriate algorithms and optimization criteria are used, measuring the performance of applied scheduling algorithms. Since such a selection is a problem specific issue, the exact setup is always discussed separately for each experiment in the following text.

6.2.2 The Importance of Complete Data Sets

We demonstrate the performance of our algorithms in the context of different amount of information in the problem. Workload traces from the Parallel Workloads Archive (PWA) [53] or Grid Workloads Archive (GWA) [50] usually do not contain several parameters that are important for realistic simulations. Typically, very limited information is available about the Grid or cluster resources such as the architecture, the CPU speed, the RAM size or the resource specific policies. However, these parameters often significantly influence the decisions and performance of the scheduler. Moreover, no information concerning background load, resource failures, or specific users' requests are available. In heterogeneous environments, users often specify some subset of machines or clusters that can process their jobs. This subset is usually defined either by the resource owners' policy (user is allowed to use such cluster), or by the user who requests some properties (library, software license, execution time limit, etc.) offered by some clusters or machines only. Also, the combination of both owners' and users' restrictions is possible, as we have discussed in Section 2.4.

When one tries to create a new scheduling algorithm and compare it with current approaches such as EASY Backfilling [156], Conservative Backfilling [52], etc., all such information and constraints are crucial, since they make the algorithm design much more complex. The natural question is whether these additional data are important, i.e., if resulting simulations may provide misleading or unrealistic results when these features are ignored. In the following text — which is based on the results that we have published in [103, 100] — we demonstrate that these data are necessary and that it is very suitable to collect and use as complete data sets as possible. We show the suitability of proposed schedule-based approach by means of the quality of the solution and runtime requirements of our new algorithms with respect to other well known and widely used queue-based algorithms.

Experimental Setup

As was discussed in Section 6.1.1, we were able to collect rather complete real life data sets from the Czech national Grid infrastructure MetaCentrum. These data cover many previously mentioned issues such as machine parameters and supported properties, specific job requirements or machine failures. Using these complete data sets we were able to

[4]Typically, this happened whenever the Random Search (RS), Tabu Search (TS) or Gap Search (GS) algorithms were applied. The results of all such experiments including values of standard deviation can be found in the attached CD-ROM.

perform several experiments that demonstrate the importance of such additional data. The demonstration is based on four different problems. The first (BASIC) problem does not involve machine failures and specific job requirements. Therefore, all jobs can be executed on any cluster (if enough CPUs are available), which represents the typical amount of information available in GWA or PWA workloads. Three remaining problems extend the BASIC problem through the use of additional information available in our data sets. In order to closely identify the effects of machine failures and specific job requirements, we have considered three different problems using the "extended" workloads. In EXT-FAIL only the machine failures are used and the specific job requirements are ignored. EXT-REQ represents the opposite problem, where the failures are ignored and only the specific job requirements are simulated. Finally, EXT-ALL uses both machine failures and specific job requirements representing the use of all available information provided in our data sets.

In this initial experiment, the differences among algorithms and the four evaluated problems were measured using five objective functions. We have used the bounded slowdown, the response time and the wait time as were described in Section 2.6.2 that represent common objectives used in the literature. Moreover, the algorithm runtime per job criterion which was described in Section 2.6.4 was used to show the time requirements of selected algorithms with respect to the studied problem. Finally, the number of killed jobs was used[5] to measure the amount of jobs that terminated prematurely due to a machine failure (see Section 2.6.2). We did not include the (weighted) machine usage criterion in this experiment since the resulting values were very similar for all considered algorithms. This is not a surprising fact [61, 112] because both data sets represent several months of execution while the average processing time of job is less than six hours. In such case, the resulting makespan — which is used to calculate the usage — is not controllable by the scheduler since it can never be smaller than the arrival time of the last job plus its processing time. Then, the utilization is rather a function of user activity than of scheduler's performance [61].

The resulting values were obtained by simulating the execution of six different scheduling algorithms. We have compared well known solutions such as FCFS, EASY Backfilling (EASY) and Conservative Backfilling (CONS) as well as the PBS-like (PBS) multi-queue priority based algorithm that were all described in Chapter 2. Our schedule-based solution was represented by Best Gap (BG) policy and Random Search (RS) optimization routine, that were described in Sections 4.6 and 4.7. In this experiment, RS operated over the initial schedule that has been incrementally built using BG, thus it is depicted as BG+RS in the figures. The *AcceptCandidate* function which is used in BG and RS (see Section 4.5.4) considered three criteria when determining the suitability of each new job assignment. Those were the bounded slowdown, the response time and the wait time[6]. RS executed periodically each 5 minutes. Here we were inspired by the actual setup of the PBS Pro scheduler [88] used in the MetaCentrum which performs priority updates of jobs waiting in the queues with a similar periodicity. The maximum number of iterations is equal to the

[5]Naturally, this criterion was used only for problems involving machine failures, i.e., for EXT-FAIL and EXT-ALL problems.

[6]The results of *fairness* criterion are not presented in this initial experiment since they will be closely discussed in the complex third experiment presented in Section 6.2.4.

number of currently waiting jobs (schedule size) multiplied by 2. The maximum time limit was set to be 2 seconds which is usually enough to try all iterations. At the same time, it is still far less than the average job inter-arrival time of the denser MetaCentrum'09 trace (120 seconds). In this evaluation, the actual job processing time was known in advance, i.e., the processing time estimates were all precise.

Experimental Results

Figure 6.2 shows the results for the MetaCentrum'08 (top) and Metacentrum'09 (bottom) data sets. As we can see the highest differences in the values of objective functions appear

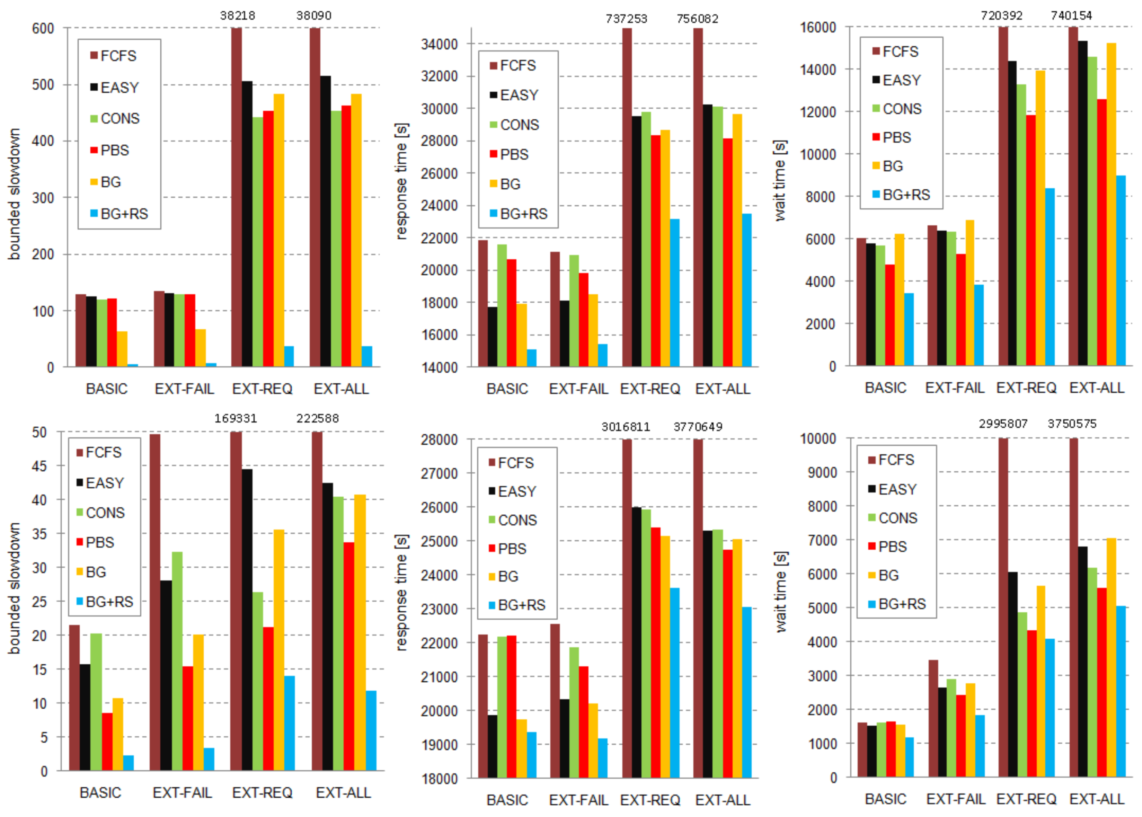


Figure 6.2: Observed values of objective functions the MetaCentrum'08 (top) and MetaCentrum'09 (bottom) workloads.

between BASIC and EXT-ALL experiments, which corresponds with our expectations. Except for the schedule-based RS optimization algorithm (BG+RS) which significantly improves especially the bounded slowdown, the differences between all algorithms are not very large for all considered objectives in case of the BASIC problem. On the other hand, when the EXT-ALL problem is applied, large differences appear for all criteria. It is most significant in case of FCFS which generates the worst results among all applied algorithms.

The values of FCFS are truncated for better visibility. EASY and CONS perform much better, while the best results are achieved by BG+RS in all cases. Interesting results are related to the EXT-FAIL and EXT-REQ scenarios. As we can see, the inclusion of machine failures (EXT-FAIL) has usually a smaller effect than the inclusion of specific job requirements (EXT-REQ). Clearly, it is "easier" to deal with machine failures than with specific job requirements when the overall system utilization is not extreme. In case of a failure, the scheduler has usually other options where to execute the job. On the other hand, if the specific job requirements are taken into account, other possibilities may not exist, and jobs with specific requests have to wait until the suitable machines become available. This is clearly visible especially for the response time and the wait time criteria. Clearly, as soon as specific job requirements are included (EXT-REQ and EXT-ALL), both response and wait time increase with respect to the BASIC and EXT-FAIL problems.

For all problems, the results of BG are usually similar to those produced by EASY, CONS or PBS. As soon as the RS optimization routine is applied (BG+RS), it always improve the quality of the solution and generates the best results among all algorithms. In case of MetaCentrum'08, a closer attention is required when analyzing the huge difference between the bounded slowdown of BG+RS and of remaining algorithms. It is a well known issue, that the slowdown criterion can exhibit such a "wild" behavior [54]. As discussed in Section 2.6.2, slowdown measures the responsiveness of the system with respect to the job length, meaning that jobs should be completed within the time proportional to their length. In another words, if a short job is not completed within a short time horizon its (bounded) slowdown can easily be huge although its actual wait time is not dramatically large. Few such huge values can significantly deviate the final (average) value. Therefore, we prefer to accompany the bounded slowdown with the response time metric, which places more weight on long jobs and basically ignores if a short job waits for a few minutes [54]. In case of RS in MetaCentrum'08, the low value of bounded slowdown is a combination of good optimization performance of RS and few extremely bad decisions of remaining algorithms.

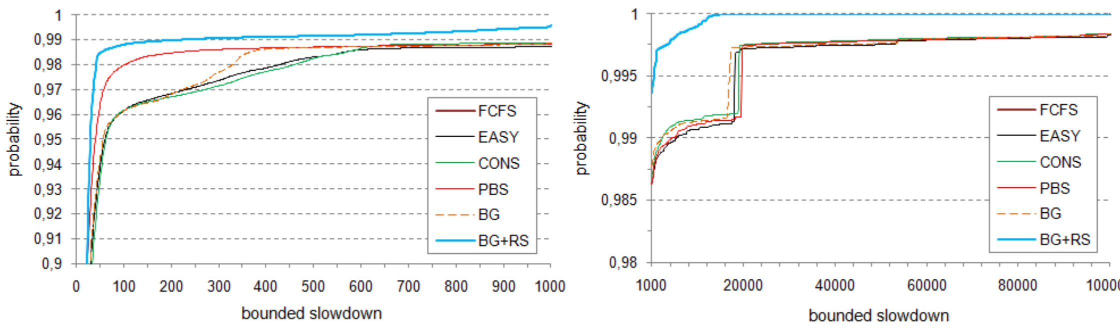


Figure 6.3: The CDFs of bounded slowdowns in MetaCentrum'08 EXT-ALL problem. The left figure's x-axis is bounded by 1000 for better visibility. The right figure's x-axis starts at 1000 and is bounded by 100,000.

The Figure 6.3 shows the cumulative distribution functions (CDF) [65] of bounded slowdowns for the MetaCentrum'08 EXT-ALL experiment. The x-axis depicts the bounded job slowdown, while y-axis is the percentage of jobs[7]. These CDFs are $f(x)$-like functions that shows the probability that a given bounded job's slowdown is less than or equal to x. In another words, the CDF represents percentage of jobs having their bounded slowdown less than or equal to x. The figure on the left clearly demonstrates the general improvement obtained by BG+RS. However, it does not indicate why the difference in the average value is so large. The reason lies in the CDF's "long tail" which — for better visibility — is displayed on the right side of Figure 6.3. Clearly, the maximum bounded slowdown is always less than 20,000 for BG+RS while 0.2% of jobs (approximately 370 jobs) have their bounded slowdown bigger than 100,000 when other algorithms are used. Apparently, the evaluation phase of RS optimization routine is capable to identify such extremely bad decisions and fix them. Since remaining algorithms do not use evaluation they are neither able to identify nor fix these types of problems.

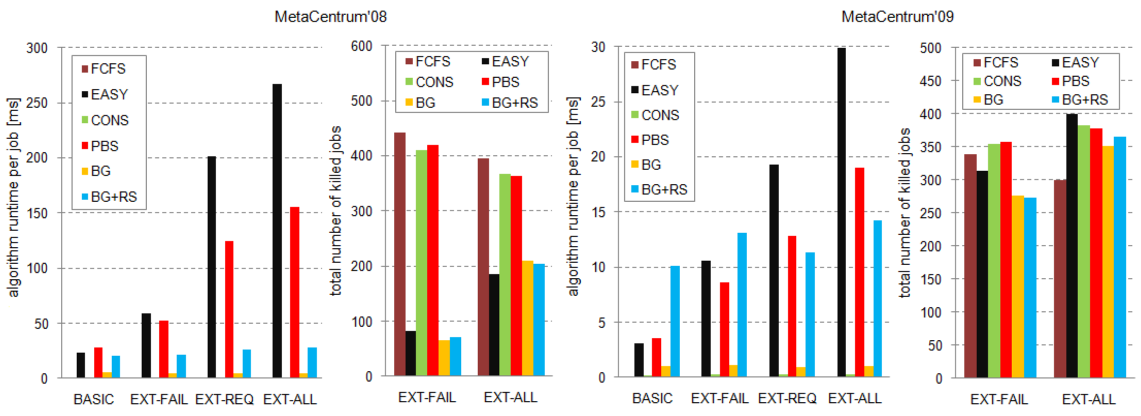


Figure 6.4: The algorithm runtime per job and the total number of killed jobs for MetaCentrum'08 (left) and MetaCentrum'09 (right).

Figure 6.4 shows the algorithm runtime per job and the number of killed jobs for MetaCentrum'08 (left) and MetaCentrum'09 (right) data sets. Following the expectations, FCFS, BG and CONS are very fast. As the problem gets more complex the algorithm runtime grows especially for EASY. CONS, BG and FCFS deals with a given a job only twice — when it arrives and when it is finally selected for execution[8]. By definition, EASY has to deal with a given job multiple times — when it arrives or when it is selected for execution, and every time it is tested to be backfilled from the queue (see Section 3.3.2), which happens repeatedly at each scheduling round. Each such additional test represents an overhead since several checks must be made concerning the available time and possible

[7]For better visibility, the y-axis of left figure starts at 0.9 and the x-axis is bounded by 1000. The right figure displays the "long tail" of the left CDF in a detail.

[8]Since the processing time estimates are accurate in this experiment, neither CONS nor BG have to perform the potentially time consuming schedule compression procedure.

collision with existing reservation of the first queued job. The runtime of RS grows as well but it is much more stable with respect to EASY and the resulting value is rather related to the algorithm setup than the problem complexity. Concerning the total number of killed jobs, there is no clear pattern indicating the best algorithm which is not surprising since the applied algorithms do not use any special technique that would aim to decrease the number of killed jobs. In our work [103], we have shown that in such situation the actual number of killed jobs is rather arbitrary, especially when the number of failures is not extreme.

Discussion

In this initial evaluation, we focused on the quality of the solution as delivered by the existing queue-based algorithms and compared it with the performance of our algorithms. Using accurate processing time estimates, we have demonstrated that our algorithms work very well. Moreover, the more complex and realistic data were applied, the larger was the improvement of our solution with respect to remaining algorithms.

We also demonstrated that the use of complex and realistic data involving specific job requirements and machine failures significantly influence the values of generated results. In case of MetaCentrum, an experimental evaluation ignoring these features may be quite misleading. As was shown, the optimistic results for the BASIC problem are very far from those appearing when a more realistic EXT-ALL problem is considered. While the effects of machine failures on the cluster [108, 194, 151] or the Grid [108, 83] performance are widely discussed, we are not aware of similar works that would also cover the effects of specific job requirements. Therefore, inspired by our own interesting results, we have performed further analysis of Grid'5000 and DAS-2 workloads from the Grid Workloads Archive. We tried to recover additional information "hidden" in these data sets covering both machine failure intervals and specific job requirements. When informations were insufficient we generated synthetic machine failures using a statistical model based on existing model of Zhang et al. [194] that has been carefully modified based on the findings describing common failure patterns in Grids [151, 83, 149]. Once created, these "extended" workloads were compared through experiment with their original simpler versions. As expected, we have often discovered disproportion in the values of objective functions similar to the differences observed for the MetaCentrum data set. Detailed description of these experiments has been published in our work [103].

Not only that there are large differences between the BASIC and EXT-ALL problems, also the differences among algorithms tend to increase when more realistic problem is applied. We have shown that the use of the proposed schedule based solution involving optimization procedure makes a great sense. The applied setup resulted in a good algorithm runtime and the inclusion of anytime approach guaranteed that no delays were introduced due to the application of the optimization procedure. Using these findings we have naturally decided to use the *complete versions* of MetaCentrum data sets (EXT-ALL) since they represent more realistic and demanding problems.

6.2.3 Experiments Involving Job Deadlines

The second experiment was performed using the synthetic data sets described in Section 6.1.3 that contain job deadlines. The motivation for this experiment was based on the promising results of the optimization procedure (BG+RS) observed in the first experiment. The idea was to determine whether the proposed techniques can be modified to deal with a different optimization problem. In this case, the major goal was to *minimize the number of late jobs*, i.e., maximize the number of jobs that meet their deadline. Minimization of the late jobs allows to support specific needs of users and provides them higher QoS. It introduces necessary step toward advanced reservation where both completion and start time are tightly constrained [104]. Following presentation is based on the research and results that we published in [104, 105, 99, 96].

Experimental Setup

Since the simulated problem was different and only synthetic data sets were available the experiment setup was different with respect to the first experiment.

First of all, several different algorithms were used while others were not used at all. Instead of "general" BG and RS, we have developed "deadline oriented" policy Best Gap — Earlier Deadline First (BG-EDF) and Tabu Search (TS) based optimization algorithm as were described in Sections 4.6.2 and 4.7.3 and published in [104, 99, 105]. Since the synthetic data sets contain neither information about users nor information about queue priorities, PBS algorithm was not used at all. Instead of that, a "deadline oriented" variant of EASY Backfilling named Flexible Backfilling (Flex-BF) [172] was applied. Flex-BF has been described in Section 3.3.3. The rest of the algorithms remained the same, i.e., FCFS, EASY and CONS were used.

To evaluate the quality of the solution delivered by the algorithms, four criteria were applied: the number of late jobs, the weighted machine usage, the bounded slowdown and the algorithm runtime per one job. Unlike the first experiment that involved large data sets covering several months of execution, the synthetic data sets covered just few hours of execution (approx. 8 hours), therefore the use of machine usage did make sense. The first three criteria were used in the *AcceptCandidate* function in BG-EDF and TS[9]. For these two algorithms, we also explored a slightly different settings where the bounded slowdown criterion was not used, meaning that only the weighted machine usage and late jobs were optimized. Again, the goal was to study the flexibility of the proposed solution when different criteria are used during the evaluation process. This setup is denoted as BG-EDF-2 and TS-2.

The maximum time limit of one TS and TS-2 execution was the same for all experiments, being set to one second, since it is the average amount of time available between two successive job arrivals in the first and most demanding data set. Several experiments were performed to determine the most suitable frequency of TS and TS-2 executions. TS and TS-2 was finally run after each five job arrivals, since it represented a reasonable tradeoff

[9]As discussed in Section 2.6.4, algorithm runtime is rather "post mortem" criterion that cannot be directly used in the *AcceptCandidate* function.

between the optimization capability and the runtime requirements. Otherwise, the setup of TS and TS-2 procedures was similar to the setup of RS in the first experiment. As in the first experiment, accurate processing time estimates were used during the simulation.

Experimental Results

The results of the experiment are shown in Figure 6.5. In each figure the x-axis always represents the average inter-arrival time between two successive jobs, i.e., it denotes which data set has been used[10]. Both TS and TS-2 operate over the schedule that has been incrementally built using BG-EDF and BG-EDF-2 respectively, thus they are depicted as BG-EDF+TS and BG-EDF+TS-2 in the figures.

In Figure 6.5 (top left), the percentage of weighted machine usage is presented. It shows that our schedule-based approach, especially BG-EDF+TS, outperforms the remaining queue-based algorithms in all cases. Using the schedule evaluation, our algorithms prefer to wait for a while until the fast machine is available. It allows to finish the job earlier than if executed immediately on a slow machine. Such situation will never happen for any of the described queue-based algorithms since they make their decisions "ad hoc" according to the current situation. In contrast to the schedule-based solution, they are unable to evaluate their decisions and adjust their behavior with respect to the applied criteria.

It is worth noticing that the weighted machine usage is also tightly related to the overall makespan. As the usage increases the makespan decreases and vice versa as is shown in Figure B.1 in Appendix B.

Figure 6.5 (top right) presents the bounded slowdown. As expected, the higher the system load is, the higher the bounded slowdown is. It is clear that inclusion of slowdown criteria into the *AcceptCandidate* function is important since TS achieves best results while results of other algorithms are significantly worse including the BG-EDF+TS-2 that does not consider bounded slowdown.

Figure 6.5 (bottom left) shows the number of jobs that failed to meet their deadline. As expected, when the job inter-arrival time increases, the number of delayed jobs decreases. BG-EDF+TS-2 together with BG-EDF-2 concentrating on the optimization of late jobs perform best. Since BG-EDF+TS and BG-EDF optimize late jobs as well as bounded slowdown, it may cause delays in start times for some jobs, thus increase the number of late jobs as can be observed in the figure. With the exception of BG-EDF in data sets 4–5, our solutions work very well in comparison with the queue-based approaches. Here EASY and Flex-BF performed better. Still, as soon as the schedule optimization is applied (BG-EDF+TS), our solution again performs better.

Figure 6.5 (bottom right) shows the algorithm runtime spent by the scheduler to compute scheduling decisions for one job. The y-axis is in logarithmic scale. It is not surprising that the runtime of the simple FCFS algorithm is very low. For EASY and Flexible Backfilling, the runtime grows as a function of the job queue length, which increases as the average inter-arrival time decreases. The runtime of both Tabu search algorithms is quite

[10]Clearly, inter-arrival time = 1 means that the first data set was used while inter-arrival time = 5 denotes the last data set.

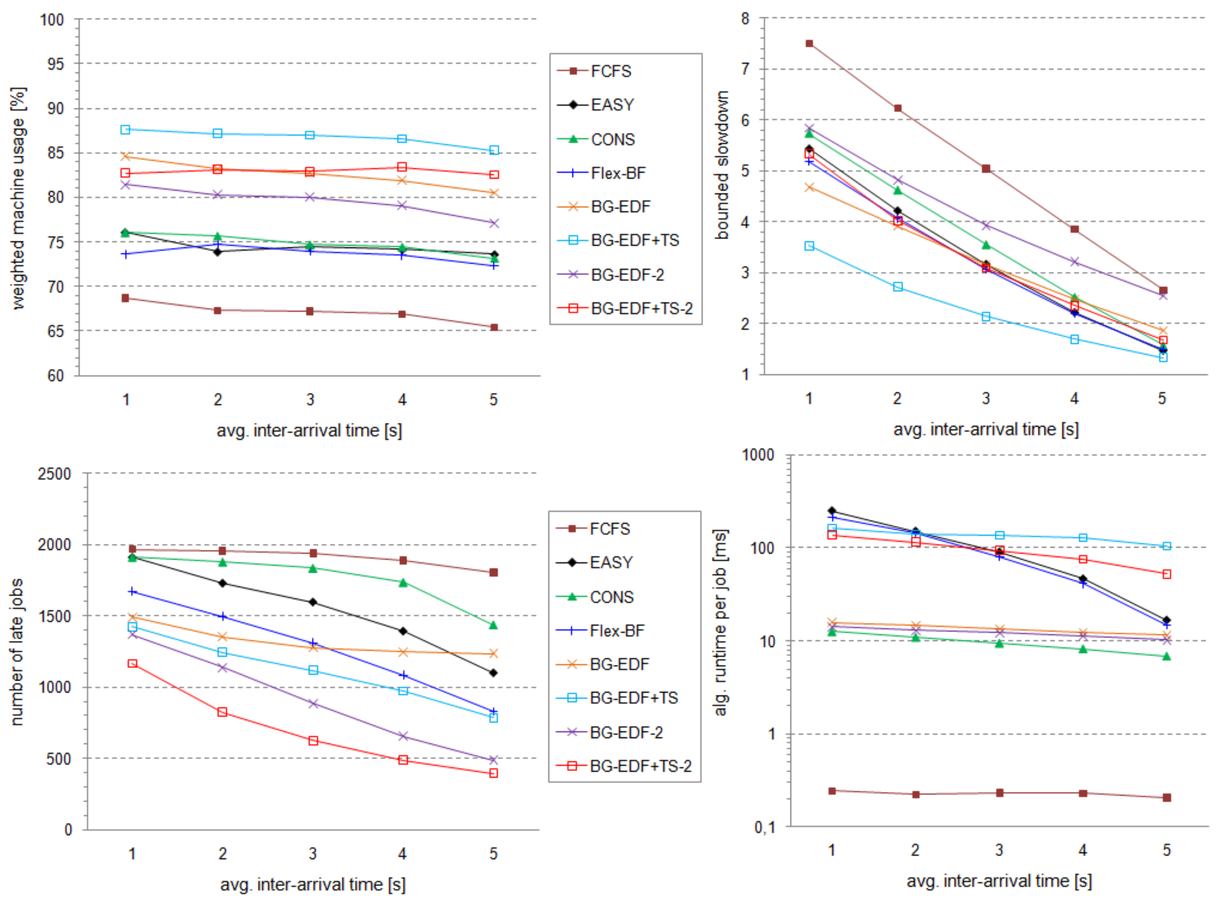


Figure 6.5: The weighted machine usage (top left), the bounded slowdown (top right) and the number of late jobs (bottom left) and the algorithm runtime per job (bottom right).

stable and grows slowly with the load of the system. Even more, bounded time for computation of the solution as well as anytime approach guarantees availability of the solution in necessary time. The BG-EDF and BG-EDF-2 policies require reasonably short runtime due to the application of the incremental approach. As expected, it is similar to CONS since they are all based on the gap filling approach. Again, since the processing time estimates are accurate in this experiment, neither CONS, BG-EDF nor BG-EDF-2 have to perform the potentially time consuming schedule compression procedure.

In our recent publication [104] we have modified the BG-EDF to follow the non-incremental “re-compute from scratch” approach applied in the production systems such as CCS or GORBA. This modification works in the following way. Every time a new job arrives the existing schedule is removed, the jobs are stored in a list and ordered according to their deadlines using the Earliest Deadline First policy. Then, the BG-EDF is repeatedly applied, taking one job at a time and placing it into the newly constructed schedule.

It starts with the first job in the list (the one having the minimal deadline) and continues until all jobs are placed into the schedule. The non-incremental BG-EDF has been used to show the available speedup of the incremental approach. Figure B.2 in Appendix B shows the results of the non-incremental BG-EDF. Not surprisingly, it had the worst runtime of all algorithms. Moreover, it required more time than it was available for traces 1 to 3. The average inter-arrival time is 1, 2 and 3 seconds and the algorithm required 11.2, 7.1 and 4.3 seconds per job on average. Clearly, in such situation the algorithm cannot be applied in a real system. This confirmed our expectations that incremental approach is very suitable to keep the runtime requirements of algorithms in acceptable levels.

Concerning the obtained values of objective function, the non-incremental version performed better than the incremental BG-EDF but it was always worse than (incremental) BG-EDF+TS. Therefore, the application of the time-efficient incremental approach did not cause any significant inefficiency with respect to the quality of the resulting solution.

Discussion

In this experiment, the proposed incremental schedule-based approaches were exploited to efficiently address multiple goals, involving the QoS related requirements of the Grid users represented by job deadlines as well as the bounded slowdown, the overall Grid utilization and the algorithm runtime. As in the first experiment, the proposed algorithms showed significant improvement in the quality of generated solution in comparison with the queue-based methods, including Flexible Backfilling which has been specially designed to deal with similar problems [172]. While optimizing different criteria on different problems, the flexibility of the proposed algorithms was demonstrated.

6.2.4 Job Processing Time Estimates

The first two experiments clearly demonstrated the suitability and flexibility of proposed schedule-based solution when dealing with different optimization problems. We have demonstrated the importance of complete data sets as well as the importance of schedule evaluation which is the crucial part of the proposed policies and optimization procedures. Also the suitability of applied incremental approach has been demonstrated. Still, both experiments used precise estimates of job processing time. As discussed in Sections 3.3.5 and 3.5, the processing time estimates are usually (highly) inaccurate in the real life applications. On the other hand, the intuition suggests that the more precise these estimates are, the better results can be expected from the scheduling techniques [32, 67, 157, 179]. Therefore, the main goal of this experiment is to determine whether the proposed solution remains suitable when the processing time estimates will be realistic, i.e., (very) imprecise.

The main idea behind this experiment is therefore straightforward — compare the performance of proposed techniques with other existing algorithms when the level of inaccuracy of the processing time estimates is increasing. In general, the expectation is that the increasing inaccuracy will decrease the quality of the generated solution. The main goal is to identify the threshold when the disproportion between the actual and the estimated processing times will cause serious problems to the studied algorithms. Therefore, in the following comparison, the inaccuracy of processing time estimates is subsequently increased before reaching the full imprecision, starting with the *accurate* estimates and continuing up to the *fully inaccurate* estimates. The first case represent the ideal situation while the latter represent the worst case scenario. This comparison has two major benefits. First, it allows to measure the effect of growing inaccuracy on the performance of evaluated algorithms, and to detect whether there is some threshold since when the algorithms' performance start to (quickly) deteriorate. Second, it represents a valuable opportunity to identify possible benefits achievable when the users' estimates would be more precise. Such information may be very important both for the users and the system administrators, since it may clearly reveal that better performance can be obtained with relatively little effort if the users' estimates are, e.g., at most twenty times larger than is the actual job processing time.

Once the main purpose of this experiment has been described, we may now proceed to the detailed description of the experimental setup.

Experimental Setup

The experiment involved six data sets. First of all, MetaCentrum'08 and MetaCentrum'09 data sets described in Sections 6.1.1 and 6.2.2 were used. The four remaining data sets were SDSC SP2, HPC2N, SDSC DataStar and LLNL Thunder, that all come from the PWA as we discussed in Section 6.1.2.

Concerning the applied objective functions, the setup was similar to the first experiment (see Section 6.2.2), i.e., the differences among applied algorithms were measured using five objective functions — the bounded slowdown, the response time, the wait time, the algorithm runtime per job and the user wait time. The latter was applied to measure the level of fairness with respect to different users of the system as discussed in Section 2.6.3.

Unlike the first experiment, the number of killed jobs was not used since the data sets from PWA do not include failure traces, thus no jobs were killed at all during the simulation. Moreover, for both MetaCentrum data sets, the results related to the number of killed jobs have already been discussed in Section 6.2.2. Finally, the (weighted) machine usage criterion was not used for similar reasons discussed in Section 6.2.2.

The resulting values were obtained by simulating the execution of six different scheduling algorithms. We used FCFS, EASY Backfilling (EASY) and Conservative Backfilling (CONS) as well as the PBS-like (PBS) multi-queue priority based algorithm. The proposed schedule-based solution was represented by Best Gap (BG) policy and periodically executed Random Search (RS) optimization routine (see Sections 4.6 and 4.7). Since this experiment involved the use of processing time estimates in most cases, the schedule compression algorithm discussed in Section 4.2 was applied when necessary, i.e., at every early job completion. In such case, an "on demand" Gap Search (GS) schedule correction procedure was launched immediately after the schedule was compressed due to an early job completion, as proposed in Sections 4.3 and 4.7.2. Since this may happen very often, the time limit for each such GS execution was set to be 50 ms. The setup of RS was identical to the first experiment, i.e, RS executed periodically each 5 minutes, the maximum number of iterations was equal to the number of currently waiting jobs (schedule size) multiplied by 2 and the maximum time limit was 2 seconds. For simplicity, the use of RS and GS that operate over the initial schedule incrementally built by BG is depicted as BG+RS in all figures[11]. The *AcceptCandidate* function (see Section 4.5.4) used in BG, RS and GS included four criteria when evaluating candidate solutions. Those were the bounded slowdown, the response time, the wait time, and the fairness.

Let us now describe the method used for increasing the inaccuracy of processing time estimates. It is inspired by the "f-model" proposed by Feitelson et al. in [55]. The goal was to subsequently increase the inaccuracy of estimates, starting with exact estimate (actual processing time $p_{i,j}$) and finishing with the roughest estimate (ep_j), which is provided in the data set. This was achieved by multiplying the actual processing time by given constant f (so called "badness factor" [180]), i.e., $p_{i,j} \cdot f$. For example, if the constant f was equal to 2, the resulting estimate would be twice as long as the actual job processing time. We used five different values of f, such that $f \in \{2, 5, 10, 20, 50\}$. Therefore, the resulting value of estimated processing time ranged from being two times to fifty times larger than the actual processing time. Moreover, each such new estimate was multiplied by a constant k from the interval $\langle 0.9, 1.1 \rangle$. It is used to avoid a situation where every new estimate is exactly f-times as long as the actual processing time. Using k, the length of the new estimate may differ from the exact product $p_{i,j} \cdot f$ up to plus or minus ten percent. The actual value of k was generated using the normal distribution with the mean equal to 1.0. Together, the resulting processing time estimate ($estim_j$) was created according to the following Formula 6.1.

$$estim_j = \min(ep_j,\ p_{i,j} \cdot f \cdot k) \tag{6.1}$$

Since the resulting estimate should not exceed its original (roughest) estimate, $estim_j$ is the minimum of the product and the original estimate ep_j.

Together, for each algorithm and data set there are seven scenarios that differs by the level of accuracy of processing time estimates. In all following figures, the use of exact processing time is depicted as *exact*, generated estimates are depicted as *x2, x5, x10, x20* and *x50* and the use of original estimate is depicted as *estim*.

Experimental Results

The main results of the experiment involving all six data sets are shown in Figures 6.6, 6.8 and 6.10. Each figure captures the resulting values of considered objective functions for two data sets. The figure is always divided into two columns and four rows. Each column represents results for one data set.

The first row contains a graph showing the cumulative distribution functions (CDF) of the actual job processing time (*exact*) together with the CDFs of generated estimates (*x2 ... x50*) as well as the CDF of the original, roughest estimate (*estim*)[12]. The x-axis depicting the processing time is in logarithmic scale. These graphs allows to analyze the

[11]We expect that it would be confusing to use "BG+RS+GS" for experiments that involve estimates and "BG+RS" for those that use actual processing times, i.e., do not need on demand GS optimization.

[12]The CDF of the actual processing time is always the leftmost red curve while the CDF of the original

precision and diversity of original estimates. The closer the CDF of *estim* is to the CDF of *exact*, the more precise is the original estimate. Moreover, the smoother the *estim* line is the higher is the diversity of user estimates. On the other hand, if the CDF of *estim* is rather staircase-shaped, it means that only few popular estimates are used through the whole data set. Remaining rows show the results for the response time, bounded slowdown and the algorithm runtime per job, respectively. Here, the x-axis depicts the precision of applied estimate, starting from the left with *exact* and continuing through *x2 ... x50* to the fully inaccurate original *estim* on the right side. Since the shape of the curve of the wait time criterion is usually very similar to the shape of the curve showing the response time, the results of the wait time for all data sets are shown in a separate Figure B.3, that can be found in Appendix B. Due to the space limitations, the results showing the fairness of considered algorithms are presented in separate Figures 6.7, 6.9 and 6.11. The figures show the cumulative distribution functions of normalized user wait times ($NUWT_o$). The x-axis depicts the $NUWT_o$, while y-axis is the probability that the $NUWT_o$ of given user o is less than or equal to x. In another words, the CDF shows the percentage of users having their $NUWT_o$ less than or equal to x. Each figure presents CDFs for *exact, x5, x20* and *estim* scenarios. Remaining CDFs (*x2, x10, x50*) can be found on the attached CD-ROM.

Although the results of FCFS were collected for all data sets, the resulting values of objective functions were always very bad (off scale)[13], thus we do not present them in the figures.

Let us start with Figure 6.6, showing the results for MetaCentrum'09 (left) and MetaCentrum'08 (right) data sets. It can be seen, that there were only three types of processing time estimates provided with the data set — 2 hours (120 minutes), 24 hours (1,440 minutes) and 30 days (43,200 minutes). Moreover, the most popular one was 24 hours (approx. 80% of jobs).

Except for the case when fully inaccurate estimates were used (*estim*), the response time was always significantly lower when the proposed BG+RS solution was applied. All algorithms that massively use processing time estimates to build the schedule or reservations' plan (CONS, BG, BG+RS) exhibited same behavior — the higher is the inaccuracy, the higher is the response time. The increase is especially visible for *estim*, which is not surprising. In such situation nearly every estimate is equal to 24 hours, thus the chance of creating reasonable schedules (reservations) is very low. The second best results were obtained by PBS. Since the PBS-like algorithm does not use processing time estimates, its results are always the same, no matter what type of estimate is currently used.

Concerning the bounded slowdown, BG+RS again performed best in most situations. In case of MetaCentrum'08 there is a large difference between BG+RS and all remaining algorithms. The reason of such large disproportion was already closely analyzed in Section 6.2.2 (see Figure 6.3).

The algorithm runtime per job is shown in the bottom of Figure 6.6. As soon as inaccurate estimates are used, the runtime of CONS and BG+RS grows since the relatively expensive schedule compression algorithm (CONS) and the on-demand optimiza-

estimate is always the rightmost black curve. CDFs of the artificially generated estimates are located between these two curves.

[13]Except for the algorithm runtime per job which — as expected — was always minimal.

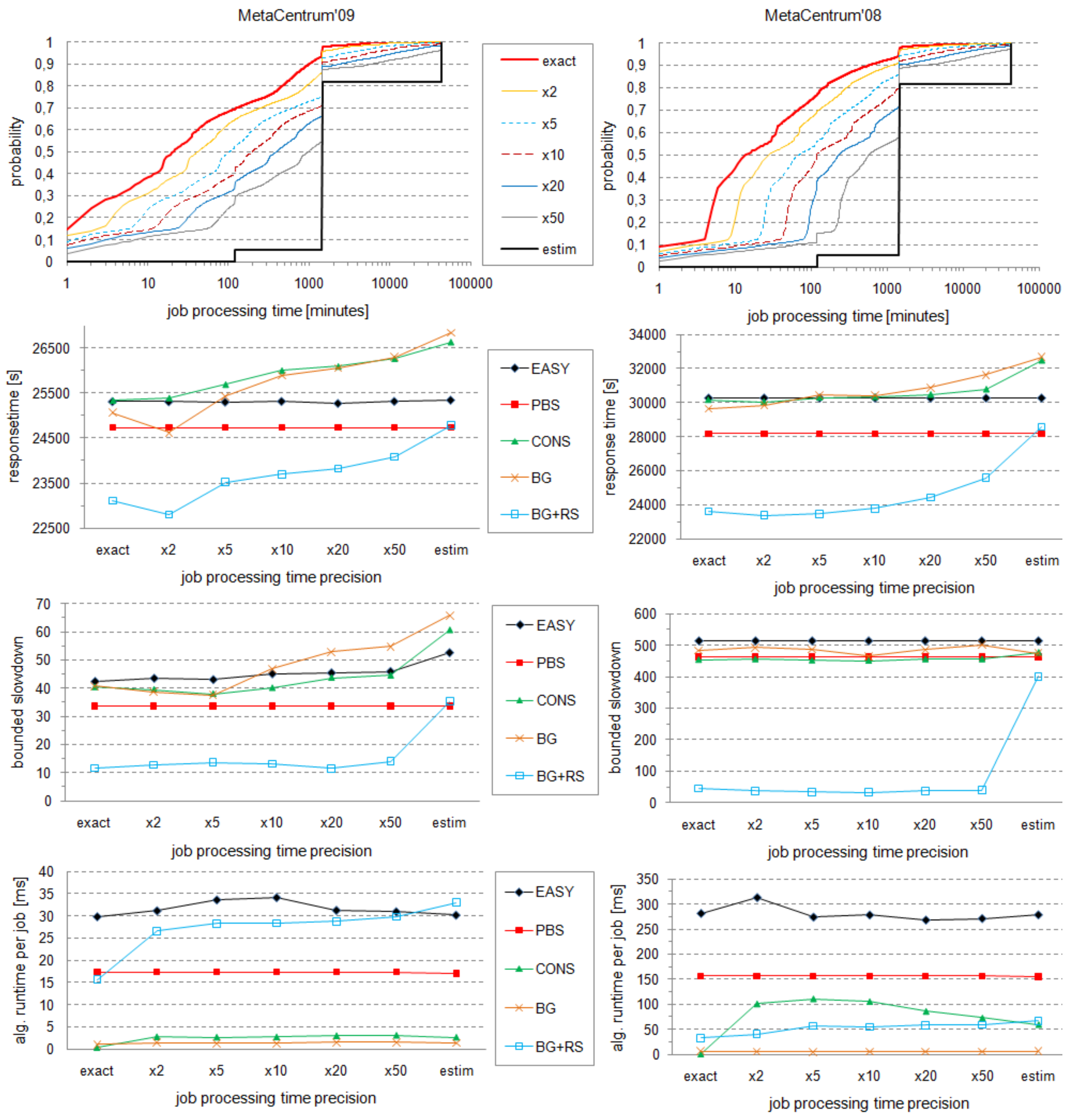


Figure 6.6: The results for MetaCentrum'09 (left) and MetaCentrum'08 (right).

tions (BG+RS) are performed frequently. The runtime of EASY is often the highest of all algorithms, especially for MetaCentrum'08, where the difference is rather large. As we already discussed in Section 6.2.2, EASY has to deal with a given job multiple times—when it arrives or when it is selected for execution, and every time it is tested to be backfilled from the queue, which happens repeatedly at each scheduling round. Moreover, the runtime also increases when the queue length is large. MetaCentrum'08 data set con-

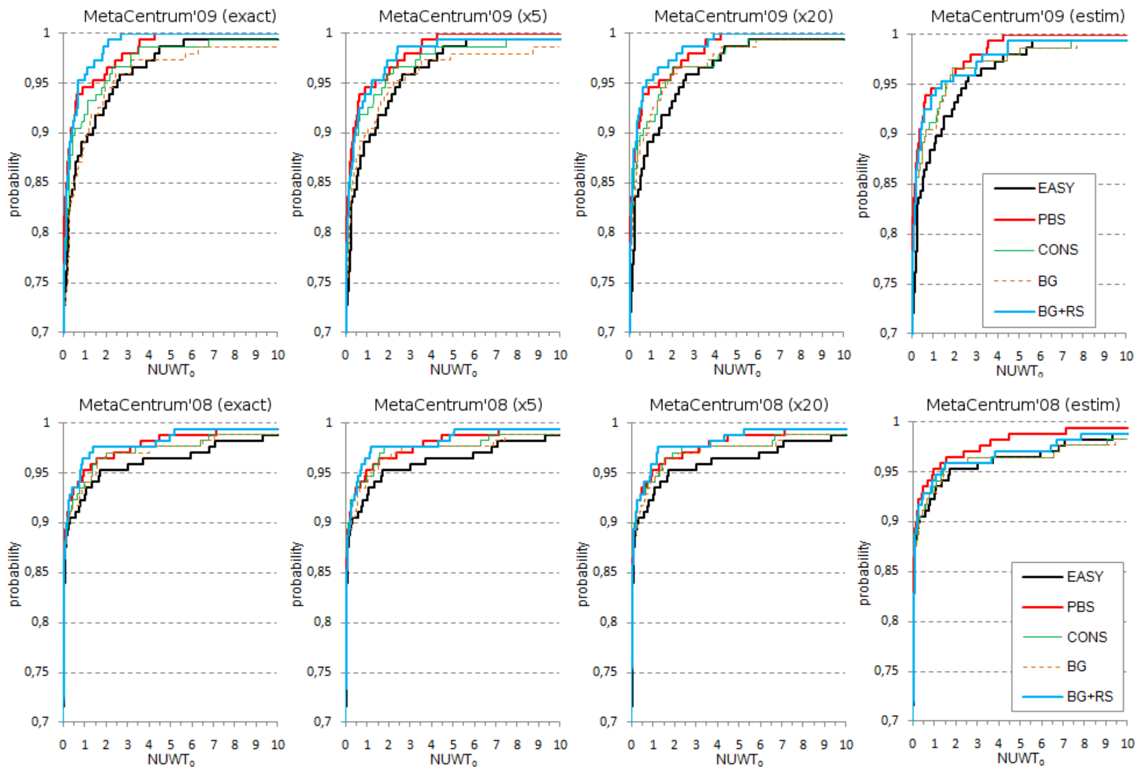


Figure 6.7: The CDFs of normalized user wait time for MetaCentrum'09 (top) and MetaCentrum'08 (bottom).

tains several peaks of high user activity which temporarily created very long queue, thus increasing the runtime of EASY. Similarly, the runtime of PBS increased for the same reason.

As discussed in Section 2.6.3, the normalized user wait time $NUWT_o$ is suitable to measure the fairness of the system with respect to its users. The normalization is used to reflect the fair share principle. The closer were the $NUWT_o$ values to each other, the higher was the fairness. If the $NUWT_o$ value was less than 1.0, it means that the given user spent more time by computing than by waiting[14]. The graphs in Figure 6.7 show that the normalized user wait time was usually near 0.0 for most users, i.e., the wait time was very low with respect to the utilized CPU time. When either PBS or BG+RS were applied the fairness has increased with respect to other solutions. This is not surprising since PBS uses priorities to maintain fairness, preferring less active users who, so far, did not utilize the system very much, while BG+RS involves fairness criterion during the evaluation of its optimization steps. Except for the *estim* scenario, BG+RS has always

[14]This represents reasonable situation for the user. On the other hand, results greater than 1.0 indicate that the total user wait time was larger than the computational time of his or hers jobs.

worked very well with respect to PBS. As soon as fully inaccurate estimates are applied (*estim*) BG+RS slightly deteriorates since nearly every estimate is equal to 24 hours, thus the chance of creating accurate evaluations concerning fairness is very low. All remaining algorithms performed slightly worse than PBS or BG+RS.

So far, mainly the performance of BG+RS was discussed. Let us briefly focus on the performance of the BG policy. It can be seen, that BG works relatively similar to CONS, which is not surprising since they both use similar principle. As soon as the inaccuracy of processing time estimates increases, CONS usually performs better, thanks to the application of more aggressive schedule compression algorithm. Since CONS does not follow incremental approach, it requires more computational time, as can be seen in the bottom graph in Figure 6.6, depicting the algorithm runtime. What is more important, is the large difference between BG and BG+RS results. It clearly demonstrates the great importance of optimization algorithms. Clearly, optimization requires an additional computational time. However, thanks to the application of anytime approach, it can never cause any significant delays with respect to job processing or scheduler's reactions on incoming events. From this point of view, the very low runtime of BG is very important, since BG policy is responsible of handling new job arrivals. Clearly, BG is able to do this job very quickly, once again demonstrating the importance of incremental approach.

Figure 6.8 shows the results for SDSC DataStar (left) and HPC2N (right) data sets. In contrast to the previous MetaCentrum data sets, the processing time estimates of both SDSC DataStar and HPC2N exhibit great variability as can be seen in the top graph in Figure 6.8.

The response time and bounded slowdown were always best when the schedule-based BG+RS was used. Even for very inaccurate estimates, the results of BG+RS are quite near to those obtained when the estimates were precise (*exact*). It may seem surprising that the response time of BG, CONS and EASY in HPC2N data set is better when inaccurate estimates are used. However, this is a well known feature that may appear at particular data sets when backfilling algorithms are used, and has been widely discussed in the literature [180, 131, 52, 72].

Again, as soon as the inaccuracy of processing time estimates increases, CONS performs better than BG at the cost of increased algorithm runtime. Using the incremental approach, the runtime of BG is stable and very low, while for CONS it grows due to the relatively expensive (non-incremental) schedule compression algorithm. The runtime of BG+RS is the highest of all algorithms for both data sets, and slowly increases as the inaccuracy grows which sounds with the expectations as on-demand optimizations are performed more frequently. Again, the application of anytime approach guarantees that no significant delays appear.

Figure 6.9 presents the CDFs of normalized user wait times reflecting the fairness criterion. For SDSC DataStar the best normalized user wait time is always obtained either by BG+RS or PBS, and the difference between PBS and BG+RS is rather minimal. For HPC2N, BG+RS delivered the best results, while PBS was near to it.

Figure 6.10 shows the results for LLNL Thunder (left) and SDSC SP2 (right) data sets. The original processing time estimates of LLNL Thunder are similar to those seen in case of MetaCentrum data sets, i.e., only few popular estimates are used through the

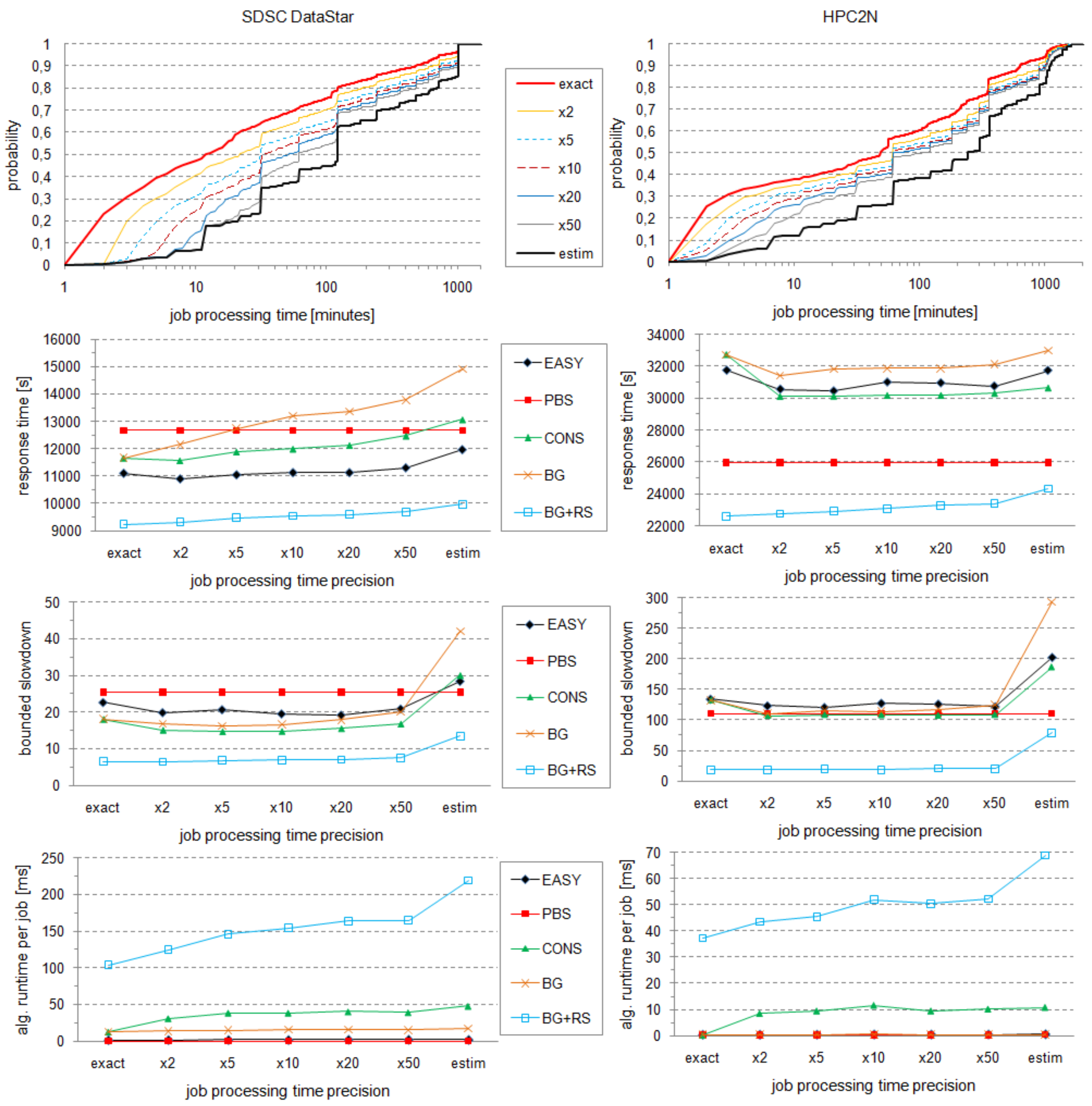


Figure 6.8: The results for SDSC DataStar (left) and HPC2N (right).

data set. On the other hand, the estimates of SDSC SP2 exhibit great variability as can be seen in the top graph in Figure 6.10.

Concerning LLNL Thunder, the best results are always delivered either by PBS or BG+RS. The performance of BG+RS is slightly worse when fully inaccurate estimates are used. This is not surprising, since the original estimates are very imprecise and exhibit little variability, thus the chance of creating reasonable schedules is very low. This is even more obvious in case of BG, which performs very badly as the inaccuracy increases. CONS

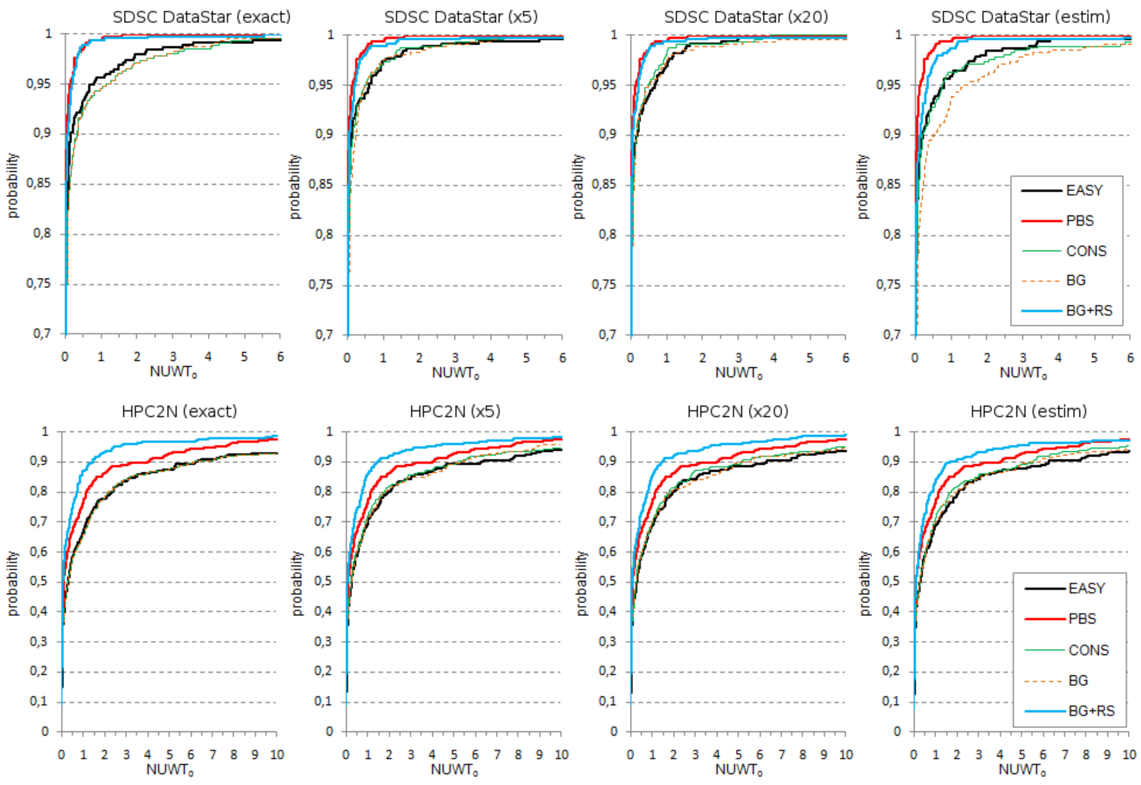


Figure 6.9: The CDFs of normalized user wait time for SDSC DataStar (top) and HPC2N (bottom).

delivers only slightly better results, while EASY performs quite well.

The response time of PBS, EASY and BG+RS is nearly the same which is quite surprising at the first sight. This is caused by very small wait time which is usually less than 3 minutes (see Figure B.3). It means that in most cases newly arriving jobs are executed with a very little waiting since suitable machines are soon available. In such situation, the resulting values of response time, wait time or bounded slowdown are very close for all algorithms. Moreover, thanks to a minimal waiting, the bounded slowdown of PBS and BG+RS is near the optimal value (1.0).

Since the average wait time in Thunder was very small also the fairness indicator — the normalized user wait time — was very low and nearly identical for PBS and BG+RS as shown in Figure 6.11 (top). Except for BG and CONS, also EASY was often close to the optimal results.

BG delivered the worst solution (especially) when the *estim* scenario was applied. In this situation, the quality of the original processing time estimates was very poor and the "less aggressive" compression algorithm was not very efficient. In this case, the compression method adopted in CONS worked better as can be seen in case of response

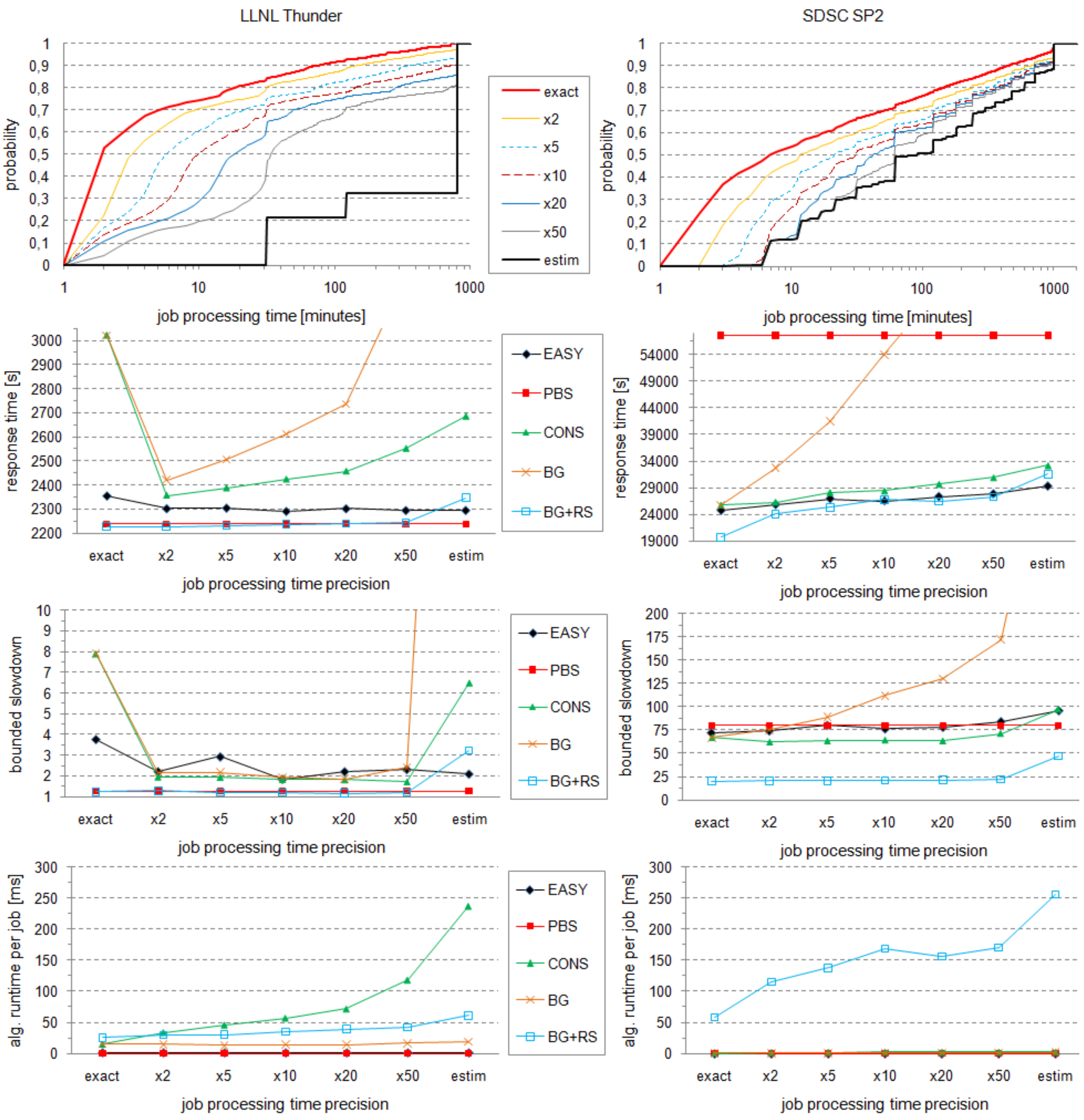


Figure 6.10: The results for LLNL Thunder (left) and SDSC SP2 (right).

time, wait time or bounded slowdown. On the other hand, CONS has the worst algorithm runtime of all algorithms. It grows rapidly as the inaccuracy grows. Once again, it clearly demonstrates that the non-incremental schedule compression algorithm is potentially quite time consuming solution[15].

[15]In this case the difference is even more visible since LLNL Thunder is a very large system (4,008 CPUs), thus the data representation of its schedule is also very large and all schedule-related operations

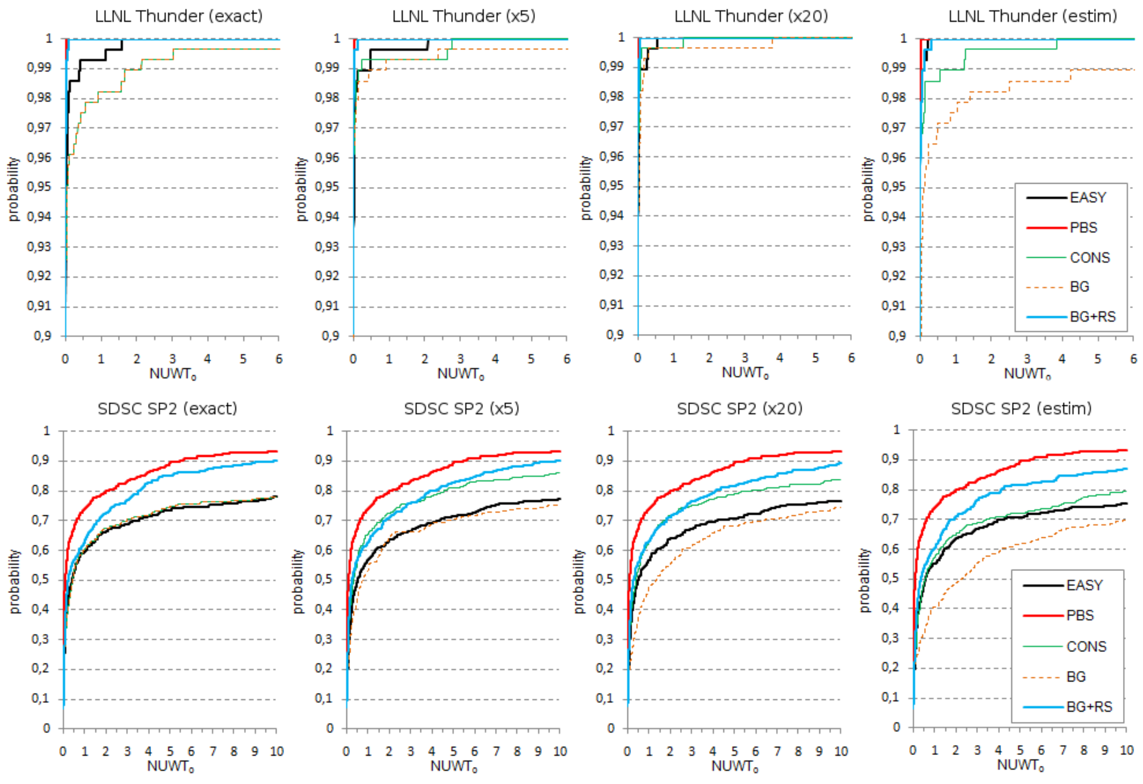


Figure 6.11: The CDFs of normalized user wait time for LLNL Thunder (top) and SDSC SP2 (bottom).

In case of the last data set SDSC SP2, the best results are achieved by BG+RS in most cases. Especially the bounded slowdown is significantly better than for all remaining algorithms. While for all previous data sets PBS usually delivered very good solutions, it has not performed very well in this case. Especially the response time was huge (57,600 seconds) with respect to CONS, EASY and BG+RS. Similar situation appeared for the wait time (see Figure B.3).

Except for BG+RS, the runtime of all algorithms was very low, including BG policy. For BG+RS, the runtime increases in a similar pattern as was observed in all previous data sets. Again, anytime approach guarantees that no significant delays will appear. Also, the great importance of optimization algorithm (BG+RS) was visible again, largely improving otherwise poor performance of the basic BG policy.

Concerning the fairness, PBS delivered the best results in all cases, while BG+RS was the second best as shown in Figure 6.11 (bottom). All remaining algorithms performed worse. The good performance of PBS demonstrates an interesting issue related to the applied optimization criteria. We must be very careful when comparing the wait time and

are naturally more time consuming.

our fairness indicator — the normalized user wait time. In case that some user has — up until now — utilized a lot of system resources, the wait time of his or hers jobs can become quite large since other (less active) users will be prioritized. It is a natural result of the priority scheme which is applied in the PBS algorithm. It can introduce large wait times for users who have utilized a lot of system resources. However, the normalization used during the computation of the normalized user wait time may easily "neutralize" these very high wait times. Together, the average wait time of the PBS can be large while the normalized user wait time is small. The performance of PBS for SDSC SP2 is an example of such situation. The wait time of PBS is very large (see Figure B.3) while the normalized user wait time is very low as shown in Figure 6.11 (bottom). From this point of view, BG+RS represents a good compromise delivering good and balanced results for all considered optimization criteria.

Discussion

In this experiment, the proposed incremental schedule-based approach was evaluated subject to subsequently increasing inaccuracy of processing time estimates. Six different data sets have been used to demonstrate the suitability of proposed techniques using five different objective functions. The experiments indicate that the proposed solution is very tolerable to the inaccurate processing time estimates. Our techniques perform very well when the estimates are not fully inaccurate. In case that the users are able to improve their job processing time estimates, our techniques provide better opportunities to increase the quality of the solution with respect to the queue-based algorithms.

Moreover, the experimental results suggest that if the estimates are significantly heterogeneous, our solution is often able to outperform remaining techniques even when the actual estimates are fully inaccurate (*estim*). On the other hand, when there are only few popular estimates (MetaCentrum'08/'09 and LLNL Thunder), the performance of the proposed techniques is similar — but not better — to the performance of the queue-based solutions. At such situation the limited diversity of job processing times basically prevents us from building efficient schedules, since there is no good opportunity for successful optimization. Therefore, it would be suitable to either motivate users to use more diverse estimates, or apply some prediction technique as were shown in Section 3.5.

6.2.5 Algorithm Runtime

In the last experiment the expected computational complexity of each proposed algorithm was experimentally compared with its actual runtime. This experiment involved Best Gap (BG) and Best Gap—Earlier Deadline First (BG-EDF) policies and Random Search (RS), Gap Search (GS) and Tabu Search (TS) optimization algorithms.

Experimental Setup

Let n is the number of jobs in schedule, *cpus* is the total number of CPUs in the system and l is the number of clusters. Then, the complexity of BG and BG-EDF is in

$\mathcal{O}\left(cpus \cdot n + l^2\right)$ as was discussed in Section 4.6. Let *iterations* is the number of iterations of the optimization procedure. Then, the complexity of RS, GS and TS is in $\mathcal{O}\left(iterations \cdot l^2 + iterations \cdot cpus \cdot n\right)$ as was shown in Section 4.7.

Clearly, there are several variables (degrees of freedom) that influence the expected complexity. Still, the complexity grows especially as the *cpus*, n and *iterations* grow while the l variable has much smaller effect because the total number of clusters is always a rather small number. Therefore, we have decided to scale the system size using *cpus* only, and the system always consisted of only one cluster which eliminated the l variable. In case of RS, GS and TS we have decided to measure the average time needed to perform one iteration. It allowed us to eliminate *iterations* from the formula. For such a scenario, the computational complexity of BG or BG-EDF as well as the complexity of one iteration of RS, GS or TS were all in $\mathcal{O}\left(cpus \cdot n\right)$.

In the experiment, both *cpus* and n were varying, simulating different size and different load of the system respectively. For *cpus*, we considered following values: 100, 200, 500, 1,000, 2,000, ..., 10,000. For n, following values were applied: 10, 50, 100, 500, 1,000, 2,000, 5,000 and 10,000. Next, the actual algorithm runtime for each combination of *cpus* and n was measured 20 times and the mean was calculated. In case of RS, GS and TS, 100 iterations were computed in each such run and then the mean time to perform one iteration was computed. Job parameters were generated synthetically using uniform distribution. Following ranges were used: job execution time (1–86,400) seconds, number of CPUs required by a job (1–16)[16]. Once all n jobs were created the initial schedule was generated randomly and the experiment started.

Experimental Results

The results of the experiment are shown in Figure 6.12. The top left graph shows the expected number of computational steps (y-axis). The x-axis shows *cpus* values while different n values are depicted by different colors of the line. The lines are constructed as the $cpus \cdot n$ product, thus representing the $\mathcal{O}\left(cpus \cdot n\right)$.

The remaining graphs in Figure 6.12 show the actual runtime of BG policy (top middle), BG-EDF policy (top right) and the average time needed for one iteration of RS (bottom left), GS (bottom middle) and TS (bottom right). All graphs show that the actual algorithm runtime grows as expected, following the trends of the top left graph. For all algorithms, there is a step increase of the algorithm runtime between 4,000 and 5,000 *cpus* which grows as the n grows. This increase is most likely caused by the fact that the growing size of the data structure representing CPUs of the system (see Section 4.5.2) can no longer be stored in the fast cache memory, and has to be stored in the main memory. When the same experiment was executed on a different machine with a different size of the cache memory, similar increase appeared again but at a different location. Otherwise, the graphs confirm the expectations and correlate with the expected computational complexity. BG-EDF needs more time than BG, which is natural since its design — involving the use of EDF policy in each iteration — is more complex with respect to the basic BG

[16]For the purpose of this experiment, arrival times were the same ($r_j = 0$) for all jobs.

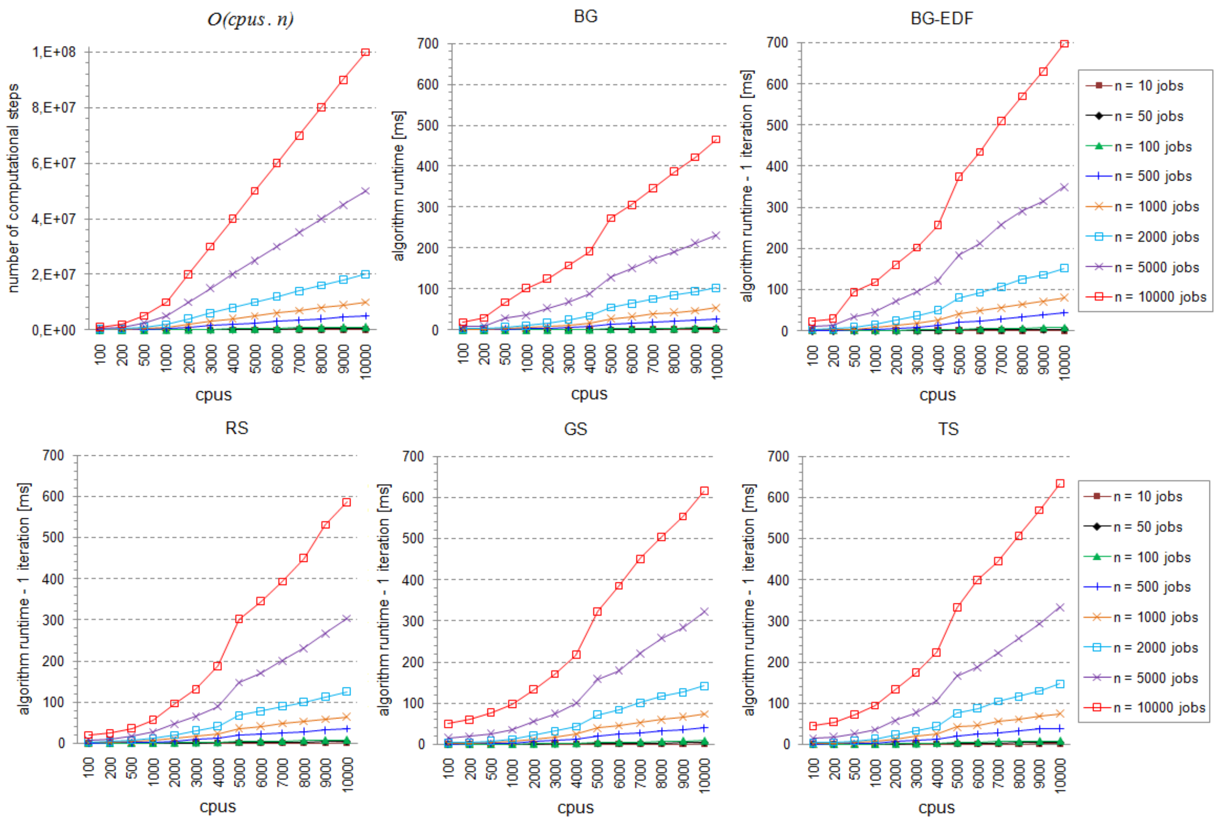


Figure 6.12: Expected computational complexity (top left) and the actual algorithm runtime of BG (top middle), BG-EDF (top right) and the average time needed for one iteration of RS (bottom left), GS (bottom middle) and TS (bottom right).

policy. GS and TS perform similarly, requiring slightly higher runtime than RS due to the applied gap-filling approach (GS) and the necessary maintenance of the tabu list (TS).

Discussion

In this experiment we have compared the actual algorithm runtime with the expected complexity that was discussed in Sections 4.6 and 4.7. Using a synthetic experiment we have evaluated several scenarios covering both changing size of the system (*cpus*) and changing load of the system (*n*). The results suggest that the actual algorithm runtime grows similarly to the predicted values.

6.3 Summary

In this chapter the proposed schedule-based solution was experimentally evaluated using different data sets and different optimization criteria. All used data sets, applied algorithm settings, and other experiment-related setup information have been described. Then, four major experiments were presented. First of all, the basic evaluation of the proposed solution have been performed, while demonstrating the importance of complete data sets for more realistic simulations. Next, the flexibility of our solution was shown when simulating problems involving job deadlines. Also, the suitability of applied incremental approach was shown here. The third complex experiment involving various processing time estimates demonstrated the suitability and tolerance of the solution to inaccurate processing time estimates. Finally, the last experiment demonstrated the good correlation between expected computational complexity and the actual algorithm runtime of our solution. To conclude, the proposed techniques performed very well with respect to various state-of-the-art queue-based solutions. The success of the solution is based on the fast, incremental scheduling policy and optimization routine that both use evaluation to guide themselves toward good solutions with respect to applied optimization criteria.

Chapter 7

Conclusion

7.1 Thesis Summary

We started our thesis with a detailed description of the complex Grid scheduling problem followed by an overview of related work, that covered several widely used queue-based approaches as well as the description of few experimental schedule-based systems. Using a formal notation, all scheduling algorithms have been described in an uniform fashion. Next, we have proposed our own model to solve complex Grid scheduling problem. It is based on the use of a schedule. It combines fast and simple scheduling policies with advanced scheduling techniques. Scheduling policies are used to construct the initial schedule while local search-based optimization techniques improve the quality of the initial solution with respect to the applied criteria. Since the schedule's construction requires upper bounds of jobs' processing times (estimates) that are typically rather inaccurate, an efficient optimization routine to perform schedule corrections has been also proposed. Also, the computational complexity of all proposed algorithms and auxiliary procedures has been explained.

The success of the proposed solution is based on several principles and approaches. First of all, the use of a schedule is very suitable when dealing with multiple objectives, since schedule de facto represents execution plan, maintaining rich information which can be used by the scheduling algorithms. For each job, specific machine(s) is selected in advance as well as the time interval when the job will be executed. Using such information, the schedule can be modified by the optimization algorithm to better meet selected criteria. Most importantly, the effect of these modifications can be evaluated by computing the values of applied objective functions. Therefore, only good solutions can be chosen from more candidate solutions, based on the values of applied objective functions.

The proposed model is able to deal with several real life situations such as dynamic job arrivals, machine failures and restarts or specific job requirements. The solution is based on a general event-based approach that handles all incoming events and then adjusts the schedule accordingly. Since the problem is not off-line, the solution must be very fast, preventing job starvation or delays in job execution. Especially the optimization algorithms must be carefully applied since in some situations they can be quite time consuming. Therefore, a time-efficient approach was greatly emphasized. For this purpose, several principles were proposed and applied. Incremental approach has been used to guarantee fast reactions to new events while anytime approach has been used to guarantee minimal delays due to the use of optimization algorithms. Last but not least, gap filling approach has been used since it works in the incremental fashion while helping to efficiently utilize available computing resources.

The proposed schedule-based solution has been experimentally evaluated, demonstrating that all important characteristics of the studied problem are properly maintained. Also, the performance of the proposed scheduling algorithms has been analyzed with respect to several state-of-the-art queue-based algorithms. It has been shown that the proposed solution usually outperforms all considered queue-based techniques. Thanks to the "on demand" correction routine, our solution remained competitive even when the processing time estimates were fully inaccurate. When multiple processing time estimates were used, our solution performed best in most situations. On the other hand, when only few popular estimates were used, the proposed solution performed similarly to the best queue-based solution, because the low diversity of job processing time estimates seriously limited the capabilities of optimization procedures.

We have also presented the platform independent fast and scalable job scheduling simulator Alea, covering typical researchers' requirements involving "ready to use" simulator that includes full implementations of common scheduling algorithms, supports common objective functions and includes popular workload format parsers. Thanks to the visualization interface it allows faster debugging and tuning of the studied algorithms as well as graphical simulation output. So far, approximately 30 foreign students and researchers found using our simulator useful for their simulations and gave us valuable feedback concerning Alea functionality.

7.2 Future Work

Although the proposed techniques have been exhaustively evaluated using several data sets, objective functions as well as variously inaccurate processing time estimates, several goals are yet ahead of us. First of all, it is the application of proposed solution in a real, production scheduler. We have decided to use the Torque Resource Manager [8] which is about to replace existing PBS Pro scheduler in the Czech NGI MetaCentrum. Our goal is to prepare fully operational implementations of our algorithms for this production scheduler. One of the students of Faculty of Informatics is currently preparing a prototype implementation of our policies and optimization algorithms in the Torque Resource Manager as a part of his Master thesis[1].

Secondly, we want to investigate whether existing techniques designed to refine or predict processing time estimates could guide the existing solution toward better performance. As was shown during the evaluation, the more precise the processing time estimates are, the better performance can be expected from our techniques. Moreover, these prediction techniques may be very useful when there are only few popular estimates used through

[1] *Grid scheduling with local search*, author Václav Chlumský.

the data set. As we have shown in Section 6.2.4, in such situation the performance of proposed techniques is similar but not better than the performance of the queue-based solutions. Our initial expectation is that the prediction techniques should introduce more diverse estimates, thus improving the performance of proposed algorithms by increasing the opportunities for successful optimization. The CDFs of processing time estimates of SDSC DataStar, HPC2N or SDSC SP2 (see Figures 6.8 and 6.10) clearly show that it is possible to receive a highly diversified processing time estimates from the users. Therefore, we believe that similar improvement can be achieved for the data sets from the MetaCentrum. For this purpose, techniques that have been presented in Section 3.5 will be primarily investigated and experimentally evaluated. One of our students is currently implementing some of these solutions as a part of his Bachelor thesis[2].

Our approach currently consider neither job preemption nor job migration. Also, the effects of resource virtualization are not considered in the current work. We plan to investigate these issues in the future. An initial implementation and introductory analysis of these features is currently done by one of the students from the Faculty of Informatics as a part of his Bachelor thesis[3].

Also, we are currently focusing on the possibility of making our current (sequential) implementation of optimization algorithm parallel. The parallel version should profit from the potential of nowadays multi-core CPUs, performing the same number of iterations in a shorter time with respect to the existing sequential implementation. A prototype implementation of parallel Random Search is currently under the development by one of the students from the Faculty of Informatics as a part of his Bachelor thesis[4].

The problem of maintaining fairness will be further investigated. In the current solution the total user wait time and the total user squashed area are used to determine the "user's priority" (see Section 2.6.3). However, existing production systems usually use time constrained input data where only recent data (e.g., user's wait times) are used to determine these priorities. For example, data older than, e.g., one month are either no longer considered or are regarded as less important [127, 88]. Once such extension is applied, also the applied objective functions have to be adjusted, since current versions such as normalized user wait time $NUWT_o$ or fairness F are computed using all data.

We also plan further improvements of our job scheduling simulator. The Alea currently supports centralized scheduling approach. We plan its extension to allow decentralized and multi-level scheduling, which is common for the large scale Grids. Naturally, this involves the application or development of new scheduling algorithms that will be made available with the future simulator's releases. Last but not least, we plan to offer some of our solutions to become a part of the future GridSim's distributions.

[2] *Scheduling with uncertain processing time*, author Miroslav Žůrek.
[3] *Grid scheduling for jobs with preemptions*, author Tomáš Svoboda.
[4] *Parallelization of optimization algorithms*, author Maroš Lipták.

Bibliography

[1] Ajith Abraham, Rajkumar Buyya, and Baikunth Nath. Nature's heuristics for scheduling jobs on computational Grids. In *The 8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000)*, pages 45–52, 2000.

[2] Ajith Abraham, Hongbo Liu, Crina Grosan, and Fatos Xhafa. Nature inspired meta-heuristics for Grid scheduling: Single and multi-objective optimization approaches. In *Metaheuristics for Scheduling in Distributed Computing Environments* [187], pages 247–272.

[3] David Abramson, Rajkumar Buyya, and Jonathan Giddy. A computational economy for Grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8):1061–1074, 2002.

[4] David Abramson, Rajkumar Buyya, Manzur Murshed, and Srikumar Venugopal. Scheduling parameter sweep applications on global Grids: A deadline and budget constrained cost-time optimisation algorithm. *Software: Practice and Experience*, 35(5):491–512, 2005.

[5] Ivo Adan and Jacques Resing. *Queueing Theory.* FreeScience, 2001.

[6] Adaptive Computing Enterprises, Inc. *Maui Scheduler Administrator's Guide, version 3.2*, April 2011. `http://www.clusterresources.com/products/maui/docs/`.

[7] Adaptive Computing Enterprises, Inc. *Moab workload manager administrator's guide, version 6.0.2*, April 2011. `http://www.clusterresources.com/products/mwm/docs/`.

[8] Adaptive Computing Enterprises, Inc. *TORQUE Admininstrator Guide, version 3.0*, April 2011. `http://www.clusterresources.com/products/torque/docs/`.

[9] David P. Anderson. BOINC: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (GRID '04)*, pages 4–10. IEEE Computer Society, 2004.

[10] Vinicius A. Armentano and Denise S. Yamashita. Tabu search for scheduling on identical parallel machines to minimize mean tardiness. *Journal of Intelligent Manufacturing*, 11:453–460, 2000.

[11] Jack J. Dongarra Asim YarKhan. Experiments with scheduling using simulated annealing in a Grid environment. In Manish Parashar, editor, *GRID*, volume 2536 of *LNCS*. Springer, 2002.

[12] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. *ACM SIGPLAN Notices*, 26(7):213–223, 1991.

[13] Ranieri Baraglia, Renato Ferrini, and Pierluigi Ritrovato. A static mapping heuristics to map parallel applications to heterogeneous computing systems: Research articles. *Concurrency and Computation: Practice and Experience*, 17(13):1579–1605, 2005.

[14] Sanjoy Baruah, Shelby Funk, and Joel Goossens. Robustness results concerning EDF scheduling upon uniform multiprocessors. *IEEE Transactions on Computers*, 52(9):1185–1195, 2003.

[15] Russel Bent and Pascal Van Hentenryck. Online stochastic and robust optimization. In *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04)*, volume 3321 of *LNCS*, pages 286–300. Springer, 2004.

[16] Jacek Blazewicz, Jan K. Lenstra, and Alexander H. G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[17] Robert Bosch and Michael Trick. Integer programming. In Burke and Kendall [23], chapter 3, pages 69–95.

[18] Peter Brucker. *Scheduling Algorithms.* Springer Verlag, 1998.

[19] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

[20] Anca Bucur and Dick Epema. Local versus global schedulers with processor co-allocation in multicluster systems. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 184–204. Springer Verlag, 2002.

[21] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The state of the art of nurse rostering. *Journal of Scheduling*, 7(6):441–499, 2004.

[22] Edmund K. Burke and Graham Kendall. Introduction. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques* [23], chapter 1, pages 5–18.

[23] Edmund K. Burke and Graham Kendall, editors. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques.* Springer, 2005.

[24] Stephen Burke, Simone Campana, Antonio Delgado Peris, Flavia Donno, Patricia Méndez Lorenzo, Roberto Santinelli, and Andrea Sciaba. *gLite 3 user guide.* Worldwide LHC Computing Grid, January 2007.

[25] Rajkumar Buyya and Manzur Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for Grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175–1220, 2002.

[26] Yves Caniou and Jean Sebastien Gay. Simbatch: An API for simulating and predicting the performance of parallel resources managed by batch systems. In *Euro-Par 2008 Workshops – Parallel Processing*, volume 5415 of *LNCS*, pages 223–234. Springer, 2009.

[27] Gabriele Capannini, Ranieri Baraglia, Diego Puppin, Laura Ricci, and Marco Pasquali. A job scheduling framework for large computing farms. In *SC'07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10. ACM, 2007.

[28] Javier Carretero and Fatos Xhafa. Using genetic algorithms for scheduling jobs in large scale Grid applications. *Journal of Technological and Economic Development – A Research Journal of Vilnius Gediminas Technical University*, 12(1):11–17, 2006.

[29] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[30] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, James Patton Jones, Scott T. Leutenegger, Uwe Schwiegelshohn, Warren Smith, and David Talby. Benchmarks and standards for the evaluation of parallel job schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1659 of *LNCS*, pages 67–90. Springer, 1999.

[31] Jenny Yan Chen, Michele E. Pfund, John W. Fowler, Douglas C. Montgomery, and Thomas E. Callarman. Robust scaling parameters for composite dispatching rules. *IIE Transactions*, 42(11):842–853, 2010.

[32] Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K. Vernon. The impact of more accurate requested runtimes on production job scheduling performance. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 103–127. Springer Verlag, 2002.

[33] Steve Chien, Russell Knight, and Gregg Rabideau. An empirical evaluation of the effectiveness of local search for replanning. In *Local Search for Planning and Scheduling*, volume 2148 of *LNAI*, pages 79–94. Springer, 2001.

[34] Walfredo Cirne and Francine Berman. A comprehensive model of the supercomputer workload. In *2001 IEEE International Workshop on Workload Characterization (WWC 2001)*, pages 140–148. IEEE Computer Society, 2001.

[35] Robert B. Cooper. *Introduction to Queueing Theory.* North Holland, second edition, 1981.

[36] CoreGRID, April 2011. `http://www.coregrid.net/`.

[37] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* McGraw-Hill, second edition edition, 2001.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2001.

[39] Kevin Coulomb, Mathieu Faverge, Johnny Jazeix, Olivier Lagrasse, Jule Marcoueille, Pascal Noisette, Arthur Redondy, and Clément Vuchener. Visual trace explorer (ViTE), April 2011. `http://vite.gforge.inria.fr/`.

[40] George B. Dantzig and Mukund N. Thapa. *Linear programming 1: Introduction.* Springer-Verlag, 1997.

[41] J. Chassin de Kergommeaux, B. de Oliveira Stein, and Bernard P.E. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253–1274, 2000.

[42] Wim Depoorter, Nils Moor, Kurt Vanmechelen, and Jan Broeckhove. Scalability of Grid simulators: An evaluation. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, volume 5168 of *LNCS*, pages 544–553. Springer, 2008.

[43] M. Dertouzos and A. K. Mok. Multiprocessor scheduling in a hard real-time environment. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 1989.

[44] Murthy V. Devarakonda and Ravishankar K. Iyer. Predictability of process resource usage: A measurement based study on UNIX. *IEEE Transactions on Software Engineering*, 15(12):1579–1586, 1989.

[45] Ciprian Dobre, Florin Pop, and Valentin Cristea. A simulation framework for dependable distributed systems. In *Proceedings of the 2008 International Conference on Parallel Processing – Workshops*, pages 181–187. IEEE, 2008.

[46] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[47] Allen B. Downey. Predicting queue times on space-sharing parallel computers. In *11th International Parallel Processing Symposium*, pages 209–218, 1997.

[48] Allen B. Downey. Using queue time predictions for processor allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *LNCS*, pages 35–57. Springer Verlag, 1997.

[49] Kathryn A. Dowsland. Classical techniques. In Burke and Kendall [23], chapter 2, pages 19–68.

[50] Dick Epema, Shanny Anoep, Catalin Dumitrescu, Alexandru Iosup, Mathieu Jan, Hui Li, and Lex Wolters. Grid workloads archive (GWA), April 2011. `http://gwa.ewi.tudelft.nl/pmwiki/`.

[51] Carsten Ernemann, Volker Hamscher, and Ramin Yahyapour. Benefits of global Grid computing for job scheduling. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 374–379. IEEE, 2004.

[52] Dror G. Feitelson. Experimental analysis of the root causes of performance evaluation results: A backfilling case study. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):175–182, 2005.

[53] Dror G. Feitelson. Parallel workloads archive (PWA), April 2011. `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[54] Dror G. Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C. Sevcik, and Parkson Wong. Theory and practice in parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *LNCS*, pages 1–34. Springer Verlag, 1997.

[55] Dror G. Feitelson and Ahuva M. Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *12th International Parallel Processing Symposium*, pages 542–546. IEEE, 1998.

[56] Pavel Fibich, Luděk Matyska, and Hana Rudová. Model of Grid scheduling problem. In *Exploring Planning and Scheduling for Web Services, Grid and Autonomic Computing*, pages 17–24. AAAI Press, 2005.

[57] Filippo Focacci, Francois Laburthe, and Andrea Lodi. Local search and constraint programming. In Glover and Kochenberger [68], chapter 13, pages 369–404.

[58] David B. Fogel. Applying evolutionary programming to selected traveling salesman problems. *Cybernetics and Systems*, 24:27–36, 1993.

[59] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[60] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure, second edition*. Morgan Kaufmann, 2004.

[61] Eitan Frachtenberg and Dror G. Feitelson. Pitfalls in parallel job scheduling evaluation. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *LNCS*, pages 257–282. Springer Verlag, 2005.

[62] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[63] Saul I. Gass and Carl M. Harris. *Encyclopedia of Operations Research and Management Science.* Kluwer, 2001.

[64] Joachim Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, second edition edition, 2003.

[65] James E. Gentle. *Computational Statistics.* Springer, 2009.

[66] Wolfgang Gentzsch. Sun Grid Engine: towards creating a compute power Grid. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[67] Richard Gibbons. A historical application profiler for use by parallel schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *LNCS*, pages 58–77. Springer Verlag, 1997.

[68] Fred W. Glover and Gary A. Kochenberger, editors. *Handbook of metaheuristics.* Kluwer, 2003.

[69] Fred W. Glover and Manuel Laguna. *Tabu search.* Kluwer, 1998.

[70] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples.* Wiley, 2002.

[71] Ronald L. Graham, Eugene L. Lawler, Jan K. Lenstra, and Alexander Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[72] Francesco Guim, Julita Corbalan, and Jesus Labarta. Impact of qualitative and quantitative errors of the job runtime estimation in backfilling based scheduling policies. Technical Report UPC-DAC-RR-CAP-2006-24, Computer Architecture Department, Technical University of Catalonia, 2006.

[73] Mor Harchol-Balter. The effect of heavy-tailed job size distributions on computer system design. In *ASA-IMS Conference on Applications of Heavy Tailed Distributions in Economics*, 1999.

[74] Ligang He, Stephen A. Jarvis, Daniel P. Spooner, Xinuo Chen, and Graham R. Nudd. Hybrid performance-oriented scheduling of moldable jobs with QoS demands in multiclusters and Grids. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun, editors, *Grid and Cooperative Computing – GCC 2004: Third International Conference*, volume 3251 of *LNCS*, pages 217–224. Springer, 2004.

[75] Darrall Henderson, Sheldon H. Jacobson, and Alan W. Johnson. Theory and practice of simulated annealing. In Glover and Kochenberger [68], chapter 10, pages 287–320.

[76] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search.* MIT Press, 2005.

[77] Kwang S. Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992.

[78] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search Foundations and Applications.* Elsevier, 2005.

[79] Matthias Hovestadt, Odej Kao, Axel Keller, and Achim Streit. Scheduling in HPC resource management systems: Queuing vs. planning. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 1–20. Springer, 2003.

[80] F. Howell and R. McNab. Simjava: A discrete event simulation library for Java. In *International Conference on Web-Based Modeling and Simulation*, pages 51–56. Society for Computer Simulation International (SCS), 1998.

[81] Rodney R. Howell. On asymptotic notation with multiple variables. Technical Report 2007-4, Department of Computing and Information Sciences, Kansas State University, 2008.

[82] Eduardo Huedo, Rubén Montero, and Ignacio Llorente. The GridWay framework for adaptive scheduling and execution on Grids. *Scalable Computing: Practice and Experience*, 6(3):1–8, 2005.

[83] Alexandru Iosup, Mathieu Jan, Ozan Sonmez, and Dick H. J. Epema. On the dynamic resource availability in Grids. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 26–33. IEEE Computer Society, 2007.

[84] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick H. J. Epema. The Grid workloads archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[85] Michael Iverson and Fusun Ozguner. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In *HCW'98: Proceedings of the Seventh Heterogeneous Computing Workshop*, pages 70–78. IEEE Computer Society, 1998.

[86] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the Maui scheduler. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *LNCS*, pages 87–102. Springer Verlag, 2001.

[87] Wilfried Jakob, Alexander Quinte, Karl-Uwe Stucky, and Wolfgang Süß. Optimised scheduling of Grid resources using hybrid evolutionary algorithms. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005*, volume 3911 of *LNCS*, pages 406–413. Springer, 2005.

[88] James Patton Jones. *PBS Professional 7, administrator guide.* Altair, April 2005.

[89] William M. Jones, Walter B. Ligon, III, Louis W. Pang, and Dan Stanzione. Characterization of bandwidth-aware meta-schedulers for co-allocating jobs across multiple clusters. *The Journal of Supercomputing*, 34(2):135–163, 2005.

[90] Michal Kafka. Estimation of job execution time in the MetaCentrum job planning system. Master's thesis, Faculty of Informatics, Masaryk University, 2010.

[91] Peter J. Keleher, Dmitry Zotkin, and Dejan Perkovic. Attacking the bottlenecks of backfilling schedulers. *Cluster Computing*, 3(4):245–254, 2000.

[92] Axel Keller and Alexander Reinefeld. Anatomy of a resource management system for HPC clusters. *Annual Review of Scalable Computing*, 3:1–31, 2001.

[93] Stephen D. Kleban and Scott H. Clearwater. Fair share on high performance computing systems: What does fair really mean? In *Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 146 – 153. IEEE Computer Society, 2003.

[94] Jon Kleinberg and Éva Tardos. *Algorithm Design.* Addison Wesley, 2006.

[95] Dalibor Klusáček. Dealing with uncertainties in Grids through the event-based scheduling approach. In *Fourth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2008)*, pages 91–98, 2008.

[96] Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Local search for deadline driven Grid scheduling. In *Third Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2007)*, pages 74–81, 2007.

[97] Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Alea – Grid scheduling simulation environment. In *7th International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, volume 4967 of *LNCS*, pages 1029–1038. Springer, 2008.

[98] Dalibor Klusáček, Luděk Matyska, Hana Rudová, Ranieri Baraglia, and Gabriele Capannini. Local Search for Grid Scheduling. Doctoral Consortium at the International Conference on Automated Planning and Scheduling (ICAPS'07), Providence, RI, USA, 2007.

[99] Dalibor Klusáček and Hana Rudová. Improving QoS in computational Grids through schedule-based approach. In *Scheduling and Planning Applications Workshop (SPARK) at the Eighteenth International Conference on Automated Planning and Scheduling (ICPAS'08)*, Sydney, Australia, 2008.

[100] Dalibor Klusáček and Hana Rudová. Complex real-life data sets in Grid simulations. In *Cracow Grid Workshop 2009 Abstracts (CGW'09)*, Cracow, Poland, 2009.

[101] Dalibor Klusáček and Hana Rudová. Alea 2 – job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST, 2010.

[102] Dalibor Klusáček and Hana Rudová. Handling inaccurate runtime estimates by event-based optimization. In *Cracow Grid Workshop 2010 Abstracts (CGW'10)*, Cracow, Poland, 2010.

[103] Dalibor Klusáček and Hana Rudová. The importance of complete data sets for job scheduling simulations. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 6253 of *LNCS*, pages 132–153. Springer Verlag, 2010.

[104] Dalibor Klusáček and Hana Rudová. Efficient Grid scheduling through the incremental schedule-based approach. *Computational Intelligence: An International Journal*, 27(1):4–22, 2011.

[105] Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. Comparison of multi-criteria scheduling techniques. In *Grid Computing Achievements and Prospects*, pages 173–184. Springer, 2008.

[106] Donald Knuth. *The Art of Computer Programming*, volume Volume 1: Fundamental Algorithms. Addison-Wesley, third edition edition, 1997.

[107] Derrick Kondo. SimBOINC: A simulator for desktop Grids and volunteer computing systems, April 2011. `http://simboinc.gforge.inria.fr/`.

[108] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. Technical Report 00433523, INRIA, November 2009.

[109] Jochen Krallmann, Uwe Schwiegelshohn, and Ramin Yahyapour. On the design and evaluation of job scheduling algorithms. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1659 of *LNCS*, pages 17–42. Springer Verlag, 1999.

[110] Krzysztof Kurowski, Jarek Nabrzyski, Ariel Oleksiak, and Jan Weglarz. Grid scheduling simulations with GSSIM. In *Proceedings of the 13th International Conference on Parallel and Distributed Systems (ICPADS'07)*, volume 2, pages 1–8. IEEE, 2007.

[111] Domenico Laforenza. European strategies towards next generation Grids. In *Proceedings of The Fifth International Symposium on Parallel and Distributed Computing (ISPDC '06)*, page 11. IEEE Computer Society, 2006.

[112] Cynthia Bailey Lee. *On the User-Scheduler Relationship in High-Performance Computing.* PhD thesis, University of California, San Diego, 2009.

[113] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate? In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *LNCS*, pages 253–263. Springer Verlag, 2004.

[114] Walter Lee, Diego Puppin, Shane Swenson, and Saman Amarasinghe. Convergent scheduling. *Journal of Instruction Level Parallelism*, 6(1):1–23, 2004.

[115] Arnaud Legrand, Loris Marchal, and Henri Casanova. Scheduling distributed applications: the SimGrid simulation framework. In *Proceedings of the Third International Symposium on Cluster Computing and the Grid*, pages 138–145. IEEE, 2003.

[116] Iosif C. Legrand and Harvey B. Newman. The MONARC toolset for simulating large network-distributed processing systems. In *Proceedings of the 32nd Conference on Winter Simulation*, pages 1794–1801. Society for Computer Simulation International, 2000.

[117] Vitus Joseph Leung, Gerald Sabin, and Ponnuswamy Sadayappan. Parallel job scheduling policies to improve fairness: a case study. Technical Report SAND2008-1310, Sandia National Laboratories, 2008.

[118] Anany Levitin. *Introduction to The Design & Analysis of Algorithms.* Addison Wesley, second edition, 2007.

[119] Bo Li and Dongfeng Zhao. Performance impact of advance reservations from the Grid on backfill algorithms. In *Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, pages 456 –461, 2007.

[120] Keqin Li. Job scheduling for Grid computing on metacomputers. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 180–188. IEEE Computer Society, 2005.

[121] David A. Lifka. The ANL/IBM SP Scheduling System. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 295–303. Springer-Verlag, 1995.

[122] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973.

[123] Xin Liu, Huaxia Xia, and Andrew Chien. Validating and scaling the MicroGrid: A scientific instrument for Grid dynamics. *Journal of Grid Computing*, 2(2):141–161, 2004.

[124] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available at `http://cs.gmu.edu/~sean/book/metaheuristics/`.

[125] Elisa Heymann María M. López and Miquel A. Senar. Sensitivity analysis of workflow scheduling on Grid systems. *Scalable Computing: Practice and Experience*, 8(3):301–311, 2007.

[126] N. Melab, E-G. Talbi, and S. Cahon. *On Parallel Evolutionary Algorithms on the Computational Grid*, volume 22 of *Studies in Computational Intelligence*, chapter 6, pages 117–132. Springer, 2006.

[127] MetaCentrum, April 2011. `http://www.metacentrum.cz/`.

[128] Ian Miguel. *Dynamic Flexible Constraint Satisfaction and its Application to AI Planning*. Distinguished Dissertations Series. Springer, 2004.

[129] Jakub T. Moscicki. DIANE – distributed analysis environment for GRID-enabled simulation and analysis of physics data. In *50th IEEE 2003 Nuclear Science Symposium, Medical Imaging Conference*, volume 3, pages 1617–1620. IEEE, 2003.

[130] Jakub T. Moscicki. DIANE: Distributed analysis environment, April 2011. `http://cern.ch/diane`.

[131] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[132] Keith Murray, Tomáš Müller, and Hana Rudová. Modeling and solution of a complex university course timetabling problem. In *Practice and Theory of Automated Timetabling VI, Selected Revised Papers*, volume 3867 of *LNCS*, pages 189–209. Springer, 2007.

[133] Patrick Niemeyer and Jonathan Knudsen. *Learning Java*. O'Reilly Media, third edition, 2005.

[134] Paul Nilsson. Experience from a pilot based system for ATLAS. *Journal of Physics: Conference Series*, 119(6):1–6, 2008.

[135] Daniel Nurmi, John Brevik, and Rich Wolski. QBETS: Queue bounds estimation from time series. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 4942 of *LNCS*, pages 76–101. Springer Verlag, 2007.

[136] Sanja Petrovic and Edmund K. Burke. University timetabling. In Leung J., editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 45. CRC Press, 2004.

[137] Michael Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, second edition, 2002.

[138] Michael Pinedo. *Planning and scheduling in manufacturing and services.* Springer, 2005.

[139] Platform Computing Corporation, Canada. *Administering Platform LSF*, 6.2 edition, 2006.

[140] Trichy N. Ravi, D. I. George Amalarethinam, and R. Balasubramanian. Evaluation of speed-up's of the improvised job scheduling strategies with overriding concept in parallel systems. *International Journal of Research and Reviews in Computer Science*, 1(4):125–130, 2010.

[141] Colin Reeves. Genetic algorithms. In Glover and Kochenberger [68], chapter 3, pages 55–82.

[142] Colin R. Reeves. Moder heuristic techniques. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, chapter 1, pages 1–25. Wiley, 1996.

[143] Graham Ritchie and John Levine. A fast, effective local search for scheduling independent jobs in heterogeneous computing environments. Technical report, Centre for Intelligent Systems and their Applications, University of Edinburgh, 2003.

[144] Graham Ritchie and John Levine. A hybrid ant algorithm for scheduling independent jobs in heterogeneous computing environments. In *PlanSIG2004: Proceedings of the 23rd annual workshop of the UK Planning and Scheduling Special Interest Group*, 2004.

[145] M. Romberg. UNICORE: Beyond web-based job-submission. In *Proceedings of the 42nd Cray User Group Conference*, pages 22–26, 2000.

[146] Gerald Sabin. *Unfairness in parallel job scheduling.* PhD thesis, The Ohio State University, 2006.

[147] Gerald Sabin, Garima Kochhar, and P. Sadayappan. Job fairness in non-preemptive job scheduling. In *International Conference on Parallel Processing (ICPP'04)*, pages 186–194. IEEE Computer Society, 2004.

[148] Gerald Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *LNCS*, pages 238–256. Springer, 2005.

[149] Ramendra K. Sahoo, Anand Sivasubramaniam, Mark S. Squillante, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 772–784. IEEE Computer Society, 2004.

[150] Vivek Sarkar. Determining average program execution times and their variance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[151] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 249–258. IEEE Computer Society, 2006.

[152] Igor Sfiligoi. glideinWMS – A generic pilot-based Workload Management System. *Journal of Physics: Conference Series*, 119(6):7–15, 2008.

[153] Jiří Sgall. On-line scheduling – a survey. In Amos Fiat and Gerhard J. Woeginger, editors, *Online Algorithms: The State of the Art*, volume 1442 of *LNCS*, pages 196–231. Springer, 1998.

[154] Edi Shmueli and Dror G. Feitelson. Backfilling with lookahead to optimize the performance of parallel job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 228–251. Springer Verlag, 2003.

[155] Oliver Sinnen. *Task Scheduling for Parallel Systems.* John Wiley & Sons, 2007.

[156] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. The EASY – LoadLeveler API project. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *LNCS*, pages 41–47. Springer, 1996.

[157] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times using historical information. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *LNCS*, pages 122–142. Springer, 1998.

[158] Warren Smith, Valerie Taylor, and Ian Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1659 of *LNCS*, pages 202–219. Springer, 1999.

[159] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 55–71. Springer Verlag, 2002.

[160] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subrarnani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. In *Proceedings of 2002 International Workshops on Parallel Processing*, pages 514–519. IEEE Computer Society, 2002.

[161] Karl-Uwe Stucky, Wilfried Jakob, Alexander Quinte, and Wolfgang Süß. Solving scheduling problems in Grid resource management using an evolutionary algorithm. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1252–1262. Springer, 2006.

[162] Riky Subrata, Albert Y. Zomaya, and Bjorn Landfeldt. Artificial life techniques for load balancing in computational Grids. *Journal of Computer and System Sciences*, 73(8):1176–1190, 2007.

[163] Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic, and Rajkumar Buyya. A toolkit for modelling and simulating data Grids: an extension to GridSim. *Concurrency and Computation: Practice & Experience*, 20(13):1591–1609, 2008.

[164] Wolfgang Süß, Wilfried Jakob, Alexander Quinte, and Karl-Uwe Stucky. GORBA: A global optimising resource broker embedded in a Grid resource management system. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005*, pages 19–24. IASTED/ACTA Press, 2005.

[165] Atsuko Takefusa, Satoshi Matsuoka, Kento Aida, Hidemoto Nakada, and Umpei Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 97–104. IEEE, 1999.

[166] 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 (HPDC-10 2001)*, pages 406–405. IEEE, 2001.

[167] El-Ghazali Talbi. *Metaheuristics – From Design to Implementation.* Wiley, 2009.

[168] David Talby and Dror G. Feitelson. Supporting priorities and improving utilization of the IBM SP scheduler using slack-based backfilling. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 513–517. IEEE Computer Society, 1999.

[169] Xueyan Tang and Samuel T. Chanson. Optimizing static job scheduling in a network of heterogeneous computers. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, pages 373–382. IEEE, 2000.

[170] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – A distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux.* MIT Press, October 2001.

[171] STAR/ITD & US-CMS MonALISA team. Development and use of MonALISA high level monitoring services for the STAR Unified Meta-Scheduler (SUMS). Technical Report PPDG-46, Particle Physics Data Grid, 2004.

[172] Ariel David Techiouba, Gabriele Capannini, Ranieri Baraglia, Diego Puppin, and Marco Pasquali. Backfilling strategies for scheduling streams of jobs on computational farms. In *Making Grids Work*, pages 103–115. Springer, USA, 2008.

[173] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 2002.

[174] Vincent T'kindt and Jean-Charles Billaut. *Multicriteria Scheduling, Theory, Models and Algorithms*. Springer, second edition, 2006.

[175] Nicola Tonellotto, Philipp Wieder, and Ramin Yahyapour. A proposal for a generic Grid scheduling architecture. In Sergei Gorlatch and Marco Danelutto, editors, *Proceedings of the Integrated Research in Grid Computing Workshop*, pages 337–346. University di Pisa, 2005.

[176] Ibaraki Toshihide, Nonobe Koji, and Yagiura Mutsunori, editors. *Metaheuristics: Progress as Real Problem Solvers*, volume 32 of *Operations Research Computer Science Interfaces Series*. Springer, 2005.

[177] Dan Tsafrir. A model/utility to generate user runtime estimates and append them to a standard workload file. `http://www.cs.huji.ac.il/labs/parallel/workload/m_tsafrir05`, April 2011.

[178] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Modeling user runtime estimates. In Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3834 of *LNCS*, pages 1–35. Springer, 2005.

[179] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789 –803, 2007.

[180] Dan Tsafrir and Dror G. Feitelson. The dynamics of backfilling: Solving the mystery of why increased inaccuracy may help. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 131–141. IEEE, 2006.

[181] Sangsuree Vasupongayya and Su-Hui Chiang. On job fairness in non-preemptive parallel job scheduling. In S. Q. Zheng, editor, *International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, pages 100–105. IASTED/ACTA Press, 2005.

[182] Xiaohui Wei, Zhaohui Ding, Shutao Yuan, Chang Hou, and Huizhen Li. CSF4: A WSRF compliant meta-scheduler. In Hamid R. Arabnia, editor, *Proceedings of the 2006 International Conference on Grid Computing & Applications, GCA 2006*, pages 61–67. CSREA Press, 2006.

[183] John Wolberg. *Data Analysis Using the Method of Least Squares: Extracting the Most Information from Experiments.* Springer, 2006.

[184] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[185] Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunisitc and dedicated scheduling with Condor. In *Conference on Linux Clusters: The HPC Revolution, Champaign-Urbana*, June 2001.

[186] Fatos Xhafa and Ajith Abraham. Meta-heuristics for Grid scheduling problems. In *Metaheuristics for Scheduling in Distributed Computing Environments* [187], pages 1–37.

[187] Fatos Xhafa and Ajith Abraham. *Metaheuristics for Scheduling in Distributed Computing Environments*, volume 146 of *Studies in Computational Intelligence.* Springer, 2008.

[188] Fatos Xhafa and Ajith Abraham. Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, 2010.

[189] Fatos Xhafa, Javier Carretero, Enrique Alba, and Bernabé Dorronsoro. Design and evaluation of Tabu search method for job scheduling in distributed environments. In *International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, pages 1–8. IEEE, 2008.

[190] Jin XU, Albert Y.S. Lam, and Viktor O.K. Li. Chemical reaction optimization for the Grid scheduling problem. In *2010 IEEE International Conference on Communications*, pages 1–5. IEEE, 2010.

[191] Ming Q. Xu. Effective metacomputing using LSF multicluster. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, pages 100–105. IEEE, 2001.

[192] Jingan Yang and Yanbin Zhuang. An improved ant colony optimization algorithm for solving a complex combinatorial optimization problem. *Applied Soft Computing*, 10(2):653–660, 2010.

[193] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for Grid computing. *SIGMOD Record*, 34(3):44–49, 2005.

[194] Yanyong Zhang, Mark S. Squillante, Anand Sivasubramaniam, and Ramendra K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 3277 of *LNCS*, pages 233–252. Springer Verlag, 2004.

Appendix A

Complexity of the Schedule Update Procedure

Introduction

In Section 4.5.2 we have discussed the complexity of the *UpdateSchedule*(k, t) procedure. Its complexity is based mainly on the repeated execution of *UpdateJob*$(job, first\_free\_slots)$ procedure. We claimed that for all n_k jobs, the time complexity of all n_k executions of *UpdateJob*$(job, first\_free\_slots)$ procedure was in $\mathcal{O}(n_k \cdot cpus_k)$ (see Formula 4.8 and related discussion). At the first sight it may not be clear why this statement holds. Therefore, a detailed explanation of algorithm's computational complexity is presented in the following text.

Algorithm and Data Structures

Let us once again summarize the functionality of the *UpdateJob* procedure. This procedure is subsequently executed for each job j and performs following steps.

(I) The earliest start time is found (S_j).

(II) The expected completion time is updated (C_j) as well as the set of assigned CPUs ($\mathcal{CPU}_j$).

(III) A list of gaps that could have appeared "in front" of j is constructed ($gaps_j$).

(IV) The auxiliary data structure that stores information about earliest free time slot on each CPU is updated, such that the new value for every CPU assigned to j is set to be equal to the completion time C_j.

To understand the complexity we must closely specify implementation details of the auxiliary data structure. For simplicity, this data structure ($first\_free\_slots$) was represented as an array of size $cpus_k$ in Section 4.5.2. However, such data structure is not very efficient when searching for minimal values. From this point of view, a *binary heap* or similar structure is much more suitable [38].

In this algorithm, the binary heap is applied in the following fashion. Each node of the heap stores a key representing the earliest available time (t_e) on some CPU(s). Moreover, each node contains a list of CPU IDs that all become free at the time t_e. In another words, if the time t_e is the same for two or more CPUs, their IDs are stored in this list, while t_e becomes the node's key in the heap. The heap is constructed according to the key values. The minimal key is the root of the heap. Let h is the number of nodes in the heap. Let IDs_i is the list of CPU IDs at the i-th node of the heap. Then following formulas hold for our data structure.

$$h > 0 \tag{A.1}$$
$$h \leq cpus_k \tag{A.2}$$
$$\sum_{i=1}^{h} |IDs_i| = cpus_k \tag{A.3}$$

Clearly, as soon as $schedule[k]$ contains at least one CPU then Formula A.1 must hold. Formula A.2 guarantees that there are at most $cpus_k$ nodes in the heap, while Formula A.3 guarantees that the total number of CPU IDs stored in the heap is the same as the number of CPUs ($cpus_k$). Using a binary heap, the complexity when finding the minimal node is in $\mathcal{O}(1)$, the minimal node can be extracted in $\mathcal{O}(\log cpus_k)$ and a new node is inserted within $\mathcal{O}(\log cpus_k)$ [38].

Complexity Analysis

We may now proceed to the complexity analysis. For one job, the complexity of each step of the *UpdateJob* procedure is following[1].

(I) At most $usage_j$ nodes must be extracted from the heap to find the earliest start time[2]: $\mathcal{O}(usage_j \cdot \log cpus_k)$.

(II) $usage_j$ steps are required to update job's internal values: $\mathcal{O}(usage_j)$.

(III) The $gaps_j$ list is constructed using at most $usage_j$ steps: $\mathcal{O}(usage_j)$.

(IV) Auxiliary data structure is updated. If the C_j is the same as key of some existing node in the heap, this existing node is extended. At most $usage_j + cpus_k$ steps are needed as there are at most $cpus_k$ nodes in the heap (see Formula A.2) and $usage_j$ steps are needed to update the IDs_i list in the selected node i. Otherwise, one new node is inserted in the heap ($usage_j + \log cpus_k$ steps). Together, the update is in: $\mathcal{O}(usage_j + cpus_k + \log cpus_k)$.

[1] The numbering of algorithm steps corresponds to the algorithm steps shown on the previous page

[2] This is the worst case when all extracted nodes contain only one CPU ID. On the other hand, when the root node i contains $|IDs_i|$ CPU IDs such that $|IDs_i| > usage_j$, then no node has to be extracted and the earliest start time is found in $\mathcal{O}(usage_j)$ time, since $usage_j$ steps are needed to remove the requested IDs from the list in node i.

Without the loss of generality we assume that all CPUs are currently free and working, i.e., there are no failed machines. Therefore, at the beginning the heap contains only one node that stores all CPU IDs ($cpus_k$). This initial node is created in $\mathcal{O}\left(cpus_k\right)$ steps. The update is performed for all n_k jobs. Clearly, for each job the algorithm extracts at most $usage_j$ nodes from the heap (see (I)) and inserts at most one node in the heap (see (IV)). Therefore, at most n_k nodes can be inserted into the heap during the n_k executions of the *UpdateJob* algorithm. Since at the beginning the heap contains only one node and every job inserts at most one node into the heap then — for all jobs together — the algorithm cannot extract more than n_k nodes during its execution, therefore for all n_k jobs the (I) step is bounded by $\mathcal{O}\left(n_k \cdot \log cpus_k\right)$. Then, for all n_k jobs the (II) and (III) steps are performed. Together, each of (II) and (III) requires $\sum_{j=1}^{n_k} usage_j$ steps. Finally, the auxiliary structure is updated (IV), which requires $\left(\sum_{j=1}^{n_k} usage_j\right) + n_k \cdot (cpus_k + \log cpus_k)$ steps. Together, the complexity for n_k jobs is shown in Formula A.4.

$$\begin{aligned}
UpdateSchedule(k,t) &= \mathcal{O}\left(cpus_k\right) + \mathcal{O}\left(n_k \cdot \log cpus_k\right) \\
&+ \mathcal{O}\left(\sum_{j=1}^{n_k} usage_j\right) + \mathcal{O}\left(\sum_{j=1}^{n_k} usage_j\right) \\
&+ \mathcal{O}\left(\left(\sum_{j=1}^{n_k} usage_j\right) + n_k \cdot (cpus_k + \log cpus_k)\right) \qquad \text{(A.4)}
\end{aligned}$$

Since $usage_j \leq cpus_k$ and $cpus_k + \log cpus_k = \mathcal{O}\left(cpus_k\right)$, this formula can be further simplified as shown in Formula A.5. Clearly, the complexity of *UpdateSchedule*(k,t) is in $\mathcal{O}\left(n_k \cdot cpus_k\right)$.

$$\begin{aligned}
UpdateSchedule(k,t) &= \mathcal{O}\left(cpus_k\right) + \mathcal{O}\left(n_k \cdot \log cpus_k\right) + \mathcal{O}\left(n_k \cdot cpus_k\right) \\
&+ \mathcal{O}\left(n_k \cdot cpus_k\right) + \mathcal{O}\left((n_k \cdot cpus_k) + (n_k \cdot cpus_k)\right) \\
&= \mathcal{O}\left(cpus_k\right) + \mathcal{O}\left(n_k \cdot \log cpus_k\right) + \mathcal{O}\left(n_k \cdot cpus_k\right) \\
&= \mathcal{O}\left(cpus_k + n_k \cdot (\log cpus_k + cpus_k)\right) \\
&= \mathcal{O}\left(n_k \cdot cpus_k\right) \qquad \text{(A.5)}
\end{aligned}$$

Appendix B

Additional Simulation Results

Experiments Involving Job Deadlines

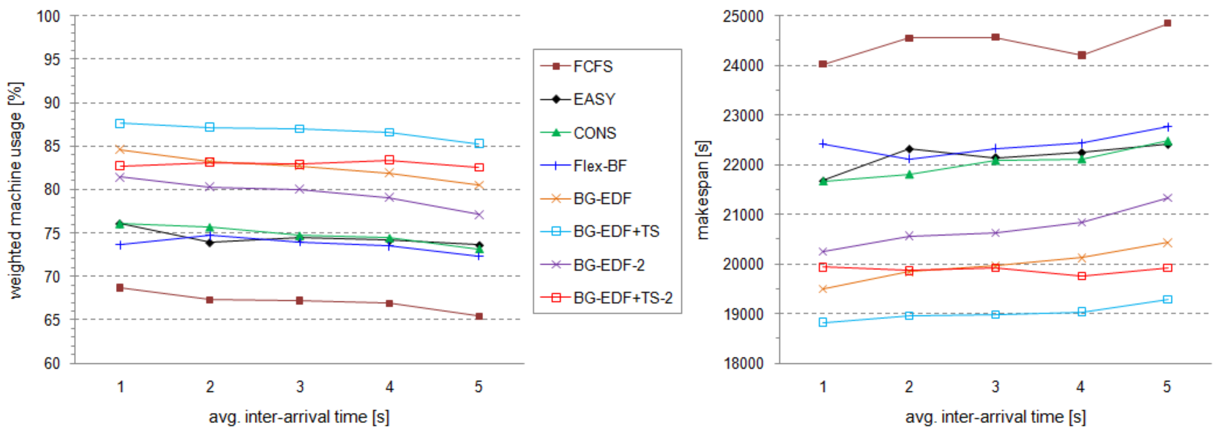


Figure B.1: The correlation between the weighted machine usage (left) and the makespan (right).

Figure B.1 shows the correlation between the weighted machine usage and makespan. It clearly demonstrates that the makespan is inversely proportional to the utilization. The higher the weighted machine usage is the lower is the resulting makespan.

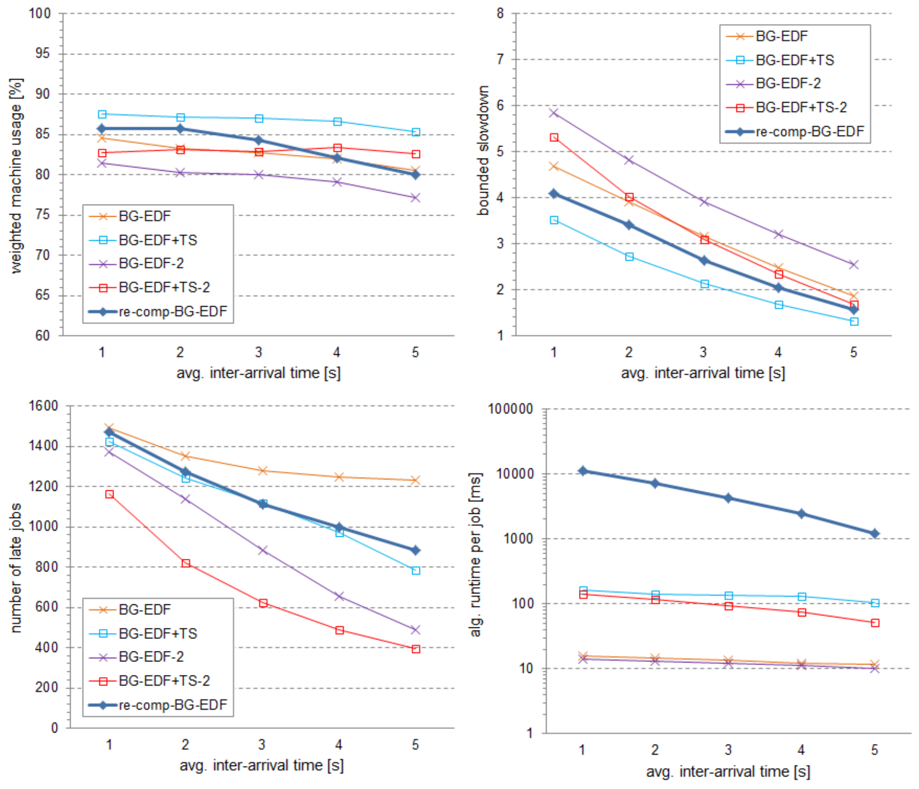


Figure B.2: Comparison of non-incremental (re-comp-BG-EDF) solution with respect to the incremental algorithms.

Figure B.2 shows the performance of non-incremental version of BG-EDF depicted as re-comp-BG-EDF in the figure. It works in a ”re-compute from scratch” fashion as was discussed in Section 6.2.3. Please note that for the first three data sets the algorithm runtime per job is actually higher than is the available amount of time between two successive job arrivals.

Experiments Involving Job Processing Time Estimates

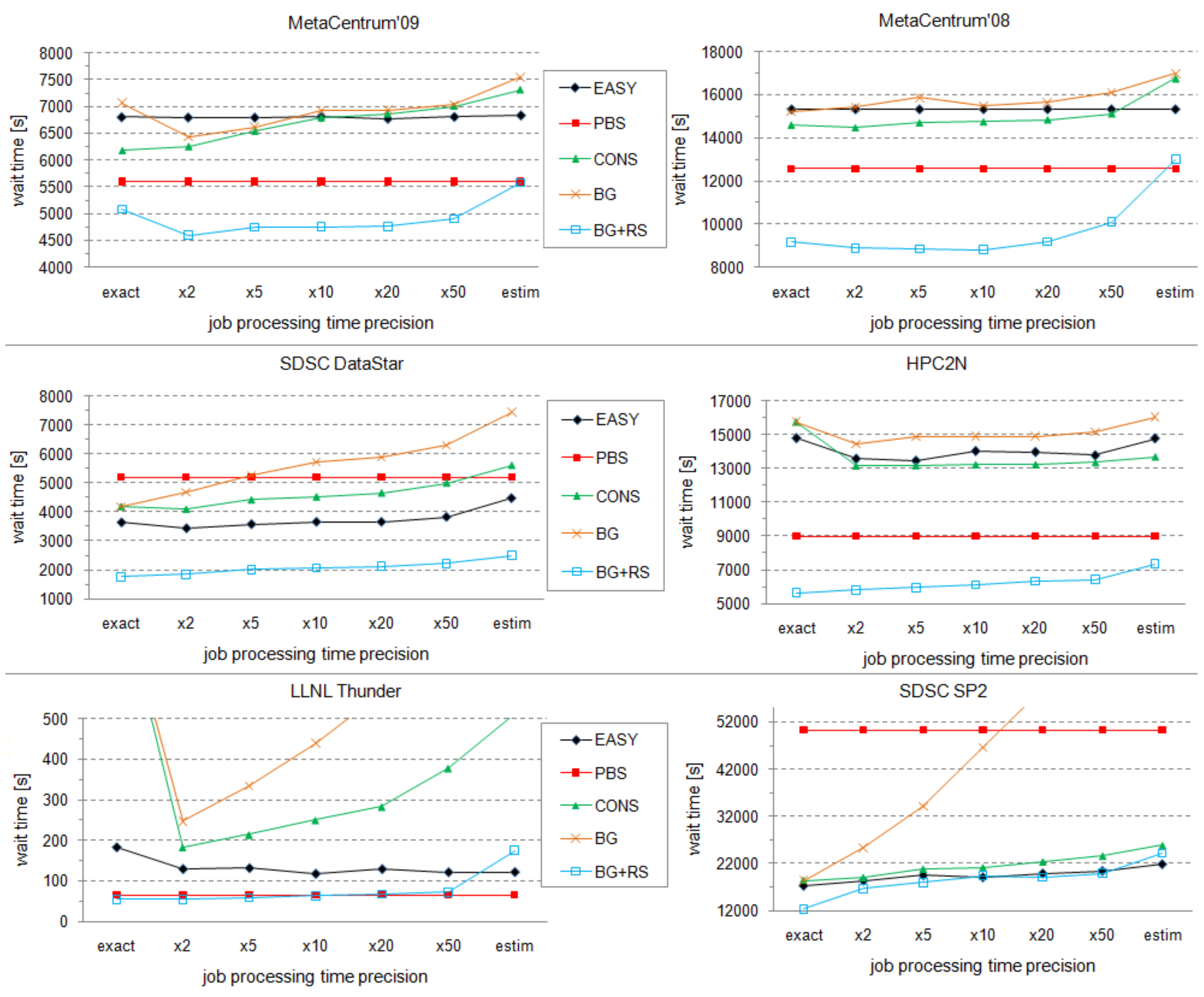


Figure B.3: The wait time for all data sets.

Figure B.3 shows the wait time for all six considered data sets.

Author's Publications

Journal Papers

- Dalibor Klusáček and Hana Rudová. Efficient Grid scheduling through the incremental schedule-based approach. *Computational Intelligence: An International Journal*, 27(1):4–22, 2011.
  ISI Journal Citation Reports®Ranking: 2009: Computer Science, Artificial Intelligence: 1/102

Reviewed Papers

- Dalibor Klusáček and Hana Rudová. The importance of complete data sets for job scheduling simulations. In Eitan Frachtenberg and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 6253 of *LNCS*, pages 132–153. Springer Verlag, 2010.
- Dalibor Klusáček and Hana Rudová. Alea 2 – job scheduling simulator. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques (SIMUTools 2010)*. ICST, 2010.
- Dalibor Klusáček and Hana Rudová. Improving QoS in computational Grids through schedule-based approach. In *Scheduling and Planning Applications Workshop (SPARK) at the Eighteenth International Conference on Automated Planning and Scheduling (ICPAS'08)*, Sydney, Australia, 2008.
- Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Alea – Grid scheduling simulation environment. In *7th International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, volume 4967 of *LNCS*, pages 1029–1038. Springer, 2008.
- Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. Comparison of multi-criteria scheduling techniques. In *Grid Computing Achievements and Prospects*, pages 173–184. Springer, 2008.
- Dalibor Klusáček, Hana Rudová, Ranieri Baraglia, Marco Pasquali, and Gabriele Capannini. Comparison of multi-criteria scheduling techniques. In *CoreGRID Integration Workshop*, pages 153–164. Crete University Press, 2008.

- Dalibor Klusáček. Dealing with uncertainties in Grids through the event-based scheduling approach. In *Fourth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2008)*, pages 91–98, 2008.
- Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Local search for deadline driven Grid scheduling. In *Third Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2007)*, pages 74–81, 2007.
- Dalibor Klusáček, Luděk Matyska, Hana Rudová, Ranieri Baraglia, and Gabriele Capannini. Local Search for Grid Scheduling. Doctoral Consortium at the International Conference on Automated Planning and Scheduling (ICAPS'07), Providence, RI, USA, 2007.
- Dalibor Klusáček, Luděk Matyska, and Hana Rudová. Problematika plánování úloh v prostředí Gridu. In *Širokopásmové sítě a jejich aplikace*, pages 55–59. Univerzita Palackého v Olomouci, 2007.

Reviewed Abstracts

- Dalibor Klusáček and Hana Rudová. The use of incremental schedule-based approach for efficient job scheduling. In *Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS 2010)*, page 206. NOVPRESS s.r.o., 2010.
- Dalibor Klusáček and Hana Rudová. Handling inaccurate runtime estimates by event-based optimization. In *Cracow Grid Workshop 2010 Abstracts (CGW'10)*, Cracow, Poland, 2010.
- Dalibor Klusáček and Hana Rudová. Complex real-life data sets in Grid simulations. In *Cracow Grid Workshop 2009 Abstracts (CGW'09)*, Cracow, Poland, 2009.