

Adaptive online scheduling of tasks with anytime property on heterogeneous resources

István Módos, Přemysl Šůcha, Roman Václavík, Jan Smejkal and Zdeněk Hanzálek

DOI: <https://doi.org/10.1016/j.cor.2016.06.008>

Cite as: I. Módos, P. Šůcha, R. Václavík, J. Smejkal, and Z. Hanzálek. Adaptive online scheduling of tasks with anytime property on heterogeneous resources. *Computers Operations Research*, 76:95 – 117, 2016

Adaptive online scheduling of tasks with anytime property on heterogeneous resources

István Módos^a, Přemysl Šůcha^a, Roman Václavík^a, Jan Smejkal^b, Zdeněk Hanzálek^a

^a*Department of Control Engineering, Faculty of Electrical Engineering, Czech Technical University, Karlovo náměstí 13, 121 35 Prague 2, Czech Republic*

^b*Merica, U Ládek 353/37, 251 01 Říčany – Strašín, Czech Republic*

Abstract

An acceptable response time of a server is an important aspect in many client-server applications; this is evident in situations in which the server is overloaded by many computationally intensive requests. In this work, we consider that the requests, or in this case *tasks*, generated by the clients are instances of optimization problems solved by anytime algorithms, i.e. the quality of the solution increases with the processing time of a task. These tasks are submitted to the server which schedules them to the available computational resources where the tasks are processed. To tackle the overload problem, we propose a scheduling algorithm which combines traditional scheduling approaches with a quality control heuristic which adjusts the requested quality of the solutions and thus changes the processing time of the tasks. Two efficient quality control heuristics are introduced: the first heuristic sets a global quality for all tasks, whereas the second heuristic sets the quality for each task independently. Moreover, in practice, the relationship between the processing time and the quality is not known *a priori*. Because it is crucial for scheduling algorithms to know at least the estimation of these relationships, we propose a general procedure for estimating these relationships using information obtained from the already executed tasks. Finally, the performance of the proposed scheduling algorithm is demonstrated on a real-world problem from the domain of personnel rostering with very good results.

Keywords: Online scheduling, anytime algorithms, machine learning, adaptive systems

1. Introduction

An important aspect of client-server applications is the response time of the server. In a case of computationally intensive requests, e.g. optimization problems, the issue of the response time is even more pressing because the server can be easily overwhelmed even by a small number of requests.

Due to financial reasons, the computational capacity of a server is commonly scaled to handle a typical workload, i.e. the arrival rate and the computational complexity of the requests, so that the response time during this typical workload is kept at an acceptable level. In a case of sudden increase in the requests, the server may become easily overloaded and the response time increases significantly resulting in user dissatisfaction. One possibility of how to mitigate the increased response time during the overload is to buy more computational resources, but such solution is not financially suitable if the overload occurs a few times a day. However, if the requests or some of the requests are instances of optimization problems, it is possible to maintain an acceptable response time by moderate degradation of the solution quality, i.e. to trade-off a small decrease in a solution quality for a significantly shorter response time.

In this paper, we consider a scheduling problem illustrated in Figure 1. *Users* work with *client applications* which generate *tasks*. The tasks are sent to a *scheduling system* which schedules the received

Email addresses: modosist@fel.cvut.cz (István Módos), suchap@fel.cvut.cz (Přemysl Šůcha), vaclarom@fel.cvut.cz (Roman Václavík), smejkal@merica.cz (Jan Smejkal), hanzalek@fel.cvut.cz (Zdeněk Hanzálek)

tasks to *computational resources*. The resources are *heterogeneous*, i.e. each resource may have a different processing power and, therefore, the processing time of the tasks may vary on each resource. The task is processed on the assigned resource, and once it is finished, its result is sent back to the scheduling system which distributes the result to the respective client application. The scheduling system receives the tasks progressively through time, i.e. we are dealing with an *online scheduling* problem [14, 26].

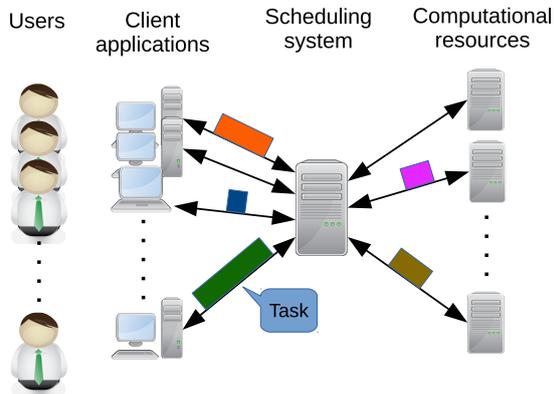


Figure 1: Overview of the environment.

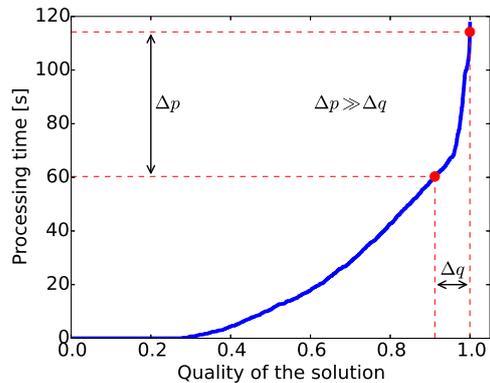


Figure 2: A typical example of the processing time function of one task.

The tasks are instances of some optimization problems and are solved by *anytime algorithms*. The property of an anytime algorithm is that the processing of a task can be interrupted at any time and the algorithm returns a feasible solution if such solution exists. The quality of the solution depends on the processing time of a task, i.e. a longer processing time may result in a better solution (the quality of a solution is defined using the objective function of the tasks' optimization problem, i.e. how close the solution is to the optimal/near optimal solution). This behavior is typical for the majority of metaheuristics and hyperheuristics solving optimization problems. The relationships between the processing time and the solution quality are defined by *processing time functions*. A typical example of a processing time function of one task is illustrated in Figure 2. In general, these functions have an increasing character: to get a better solution, an anytime algorithm must perform more operations or explore a larger part of the solution space. From Figure 2, it can also be seen that a slight deterioration of the solution quality can significantly shorten the processing time of the task and thus reduce the response time of the system. From the user point of view, a good solution is better than excessive waiting time for the near-optimal/optimal solution.

In reality, the processing time functions are not known *a priori* as is usually considered in the related literature. The reason for this is that the anytime algorithms search the solution space and the algorithms are not generally aware where the good solutions are. Without any knowledge of the processing time functions, the scheduling system cannot guarantee the response time because the scheduling system does not know how long the processing of the tasks will take. However, using either statistical or machine learning methods, the processing time for the given quality can be estimated from the previous executions of similar tasks.

In this study, we focus on the situations in which the scheduling system is *overloaded*, i.e. the response time of the system increases significantly due to increased workload. The idea of how to tackle the system overload is to control the requested quality of solutions, i.e. when the overload of the system is detected, the system trades off the quality of solutions so that the response time is kept close to the acceptable level. On the other hand, the system requests the highest quality of the solutions if the overload is not detected.

We want to emphasize that our solution does not substitute *clouds* [9]. In fact, a cloud can be integrated into the scheduling system as a cluster of computational resources. However, the proposed scheduling system replaces the default cloud load balancer because the scheduling policy (see Section 4.1) makes more informed decisions than the default cloud load balancer and thus increases the responsiveness more efficiently. For example, Amazon Elastic Load Balancer uses simple Round-Robin algorithm for TCP listeners and Least-Number-Of-Requests for HTTP listeners [1].

1.1. Case study

The validity of the proposed scheduling system is evaluated on an existing web-based personnel rostering application called Roslab¹. *Personnel rostering* [8] is a combinatorial optimization problem where a set of shifts is assigned to employees so that *hard constraints* (typically given by the labor code) are satisfied and the penalty accumulated due to *soft constraints* violations (representing the quality of a solution) is minimized. An assignment of shifts to employees is called a *roster*.

The main functionality provided by the Roslab client application is: (i) rostering, i.e. an automatic design of a roster, (ii) rostering, i.e. a partial roster correction due to external events such as illness of an employee, and (iii) validation of the roster changes which were introduced by users. Different user interactions with the client application generate different computational intensive tasks (e.g. the automatic roster design) and these tasks are solved by the anytime heuristic algorithms [4] on the server side.

At present, the available computational capacity of Roslab is scaled to handle the average workload. However, the response time increases considerably when many users interact with the system at the same time, which causes a higher risk of not prolonging the service contract. Because these critical situations occur only a few times a day, it is not financially suitable to buy new computational resources. To increase the responsiveness, we can exploit the fact that the algorithms implemented in Roslab for solving the tasks have typically non-linear progress of the solution quality, i.e. the quality difference between the optimal and a high-quality solution is small while the processing time difference is significant (see Figure 2). Therefore, it is possible to apply the proposed algorithms to control the trade-off between the solution quality and responsiveness.

1.2. Related work

1.2.1. Scheduling and overload

In distributed systems such as grids [7], most practical scheduling problems are \mathcal{NP} -hard [30]. The schedulers must also consider and balance different objective functions, e.g. response time, the fairness of resource sharing [19], load balancing, etc. Therefore, most schedulers use some heuristics which strive to find a reasonable solution to the given problem in an acceptable amount of time, i.e. the solution does not have to be optimal, but it should be found quickly. Various heuristics, such as *Min-Min* [16], *Sufferage* [22], *Local search* [2, 30] or *Genetic algorithms* [2, 30] have been presented. A good scheduling algorithm can utilize all available resources and, therefore, handle many tasks in parallel. However, in the case of overload when the capacity of resources is not enough, additional methods must be employed. Buying a new computational resource is reasonable if the system is constantly overloaded. However, if the overload is sparse, the purchase of a new resource is not justified because the resource would just consume energy without actually processing any tasks.

A common approach to handling the overload is to employ an *admission control* [29, 10] which drops the tasks of some users when the overload is detected to guarantee the acceptable response time for the rest of the users. However, such a solution is not acceptable in our case because all users must be served.

Another approach is to shorten the processing time of each task so that some performance objective is maximized. The control of the processing times has different names in the literature: *imprecise computation* [20], *controllable processing time* [27, 28], *partial jobs* [5], *increasing reward with increasing service* [6], and *Quality of Service degradation* [24, 11]. The closest work to our problem is [28], where an algorithm for balancing the tardiness and total cost of the processing time compression on a single machine is given. The relationship between the compression cost and the processing time of the tasks is linear. Although the presented algorithm seems to have a good asymptotic complexity of $\mathcal{O}(n^2)$, where n is the number of the tasks, we think that the actual complexity should be $\mathcal{O}(p_{max} \cdot n^2)$ where p_{max} is the maximum processing time of the tasks. The reason for this is that the presented algorithm in each iteration decreases the processing time of some task by 1 and, in the worst case, the algorithm will iterate until the processing time of the tasks cannot be further compressed. The authors probably considered the maximum processing time as a constant - in this case, the former complexity would be correct. Therefore, the scheduling algorithm

¹<http://www.merica.cz/products/roslab/benefits>

is not suitable for the online environment where the range of the processing times is large. In [11], the relationship between the solution quality and the processing time is assumed to be increasing and concave. This assumption is reasonable for problems solved by anytime algorithms. However, the presented algorithm has a high complexity of $\mathcal{O}(n^4)$ and the authors assume that all tasks share the same quality profile function. Other works allow preemption [20, 6, 5] which is not suitable because if the tasks consume a lot of memory, then context switching could incur a big overhead. Another assumption used in the cited works is that the processing time of the tasks is known or that the due dates are strict [20]. In our environment, it is not critical to meet due dates for all tasks but for the majority of tasks, i.e. we deal with *soft real-time* scheduling.

Our first idea for the quality control was to use *control theory* [12]. An example of an application of the control theory in computing systems is [21] where a combination of the admission control and Quality of Service degradation using a *Proportional-Integral-Derivative* controller is described. However, the control theory assumes that the state of the controlled system is in the neighborhood of the operating point. If the controlled system is non-linear and the state is far away from the operating point, the control could produce undesirable behavior such as oscillations. As an example, consider a sudden increase in the arrival rate of tasks to the server which changes the effect of the actuators on the measured output considerably. A possible approach to solving this problem is to adapt the parameters of the control law dynamically when a change in the system state is detected [18]. The disadvantage of the parameters adaptation approach is that it cannot adapt to rapid changes in a workload because it needs some time to acquire a sufficient number of past measurements before the parameters are adapted. For the previously mentioned reason, we designed quality control algorithms which do not rely on the parameters describing the operating point of the system (see Section 4.2).

1.2.2. Estimation of the processing time functions

Because the processing time functions of the tasks (see Figure 2) are not known *a priori*, they have to be estimated. Some simple methods were proposed in the literature such as using the average of the last n values of the processing time. However, these methods cannot be used if the processing times of the tasks are significantly different. A more successful approach is to use statistical methods or methods from machine learning [25, 15]. In comparison with single resource environment, estimation of the processing time of the tasks in heterogeneous systems is even more complicated, i.e. the same task may have different processing times on the resources with a different processing power. Therefore, the same estimation cannot be directly used for different resources. In [17], this problem is solved by *benchmarking* the resources. When a new resource connects to the scheduling system, series of benchmarking tests are run on this resource. The result of these tests is a vector of numbers where each number denotes how well the resource performed on the corresponding test. The authors assume that the processing time of one particular task is the same on resources with similar benchmark results and, therefore, they use observations of the processing time on one resource to estimate the processing time on a different resource. In [17], the benchmark result is appended to a feature vector of a task and the k -nearest neighbor method is used to find similar observations from which an estimation of the processing time is computed. An alternative approach to appending a benchmark result to a feature vector is to scale the processing time of a task by the benchmark result of the resource on which the task is processed [25]. However, none of these works deal with scheduling of tasks solved by anytime algorithms.

1.3. Contribution and outline

From the summarized related work it can be seen that no work fully addresses the problem of online scheduling of computationally intensive tasks which are solved by anytime algorithms and for which the processing time functions are not known beforehand. The existing literature either assumes exact knowledge or specific shape of the processing time functions which is not realistic in practice. Moreover, some works allow preemption [20, 6, 5], optimize a different objective function of the scheduling problem [28] or the presented algorithms have a high complexity [11]. Therefore, we introduce a new modular scheduling system which can guarantee an acceptable response time even in the case of overload. Moreover, since the

processing time functions of the tasks are not known *a priori*, we propose an estimator which can provide the estimation of the whole processing time function using knowledge acquired from the execution of similar tasks. The proposed scheduling system is evaluated on a real client-server application from the domain of personnel rostering.

Our approach to tackling the overload problem combines a traditional scheduling policy with a quality control algorithm. We propose two novel and efficient quality control algorithms: (i) *bisection control*, which sets one global quality for all tasks and (ii) *independent control*, which controls the quality of the solutions for each task independently. The experiments in Section 6 show that the proposed scheduling policy and quality control algorithms: (i) can decrease the response time significantly in situations in which the system would be overloaded if the quality control was disabled, (ii) outperform a simple control approach which always stops the task that was running for the longest time, and (iii) are robust to small errors in the estimation of the processing time functions.

The rest of the paper is organized as follows: In Section 2 we formulate our scheduling problem formally. The overview of the scheduling system is presented in Section 3. Section 4 contains the description of the presented scheduling policy and quality control algorithms. Section 5 explains how the estimation of the processing time functions works. In Section 6, the proposed system is experimentally tested on real world instances. Finally, the last section concludes the paper.

2. Problem statement

In the whole text, it is assumed that the time is discrete and that one time unit equals one millisecond. $T = \mathbb{N}$ denotes a set of time instants t . The *quality of solutions* (or just quality) is denoted as $\phi \in \Phi$, where $\Phi = [0, 1]$. Interval Φ can be understood as normalized values of the tasks' objective functions, i.e. 1 represents the best possible solution while 0 represents the worst possible solution.

The scheduling problem is defined by a tuple $\Pi = (N, M, \Psi, f, p, WCT)$, where $N = \{1, \dots, n\}$ is a set of *tasks*, $M = \{1, \dots, m\}$ is a set of heterogeneous *resources* on which the tasks are processed, Ψ is a set of all possible *instances*, $f : \Psi \rightarrow \mathbb{R}^D$ represents a *feature function*, which maps each *instance* $\psi \in \Psi$ to a *feature vector* with dimensionality of $D \in \mathbb{N}_{>0}$, $p : \Psi \times \Phi \rightarrow \mathbb{N}_{>0}$ represents a *normalized processing time function*, and $WCT : \Psi \rightarrow \mathbb{N}_{>0}$ is a *normalized worst case processing time function* (the meaning of the normalization will be explained later in this section). The instances in this context are instances of the optimization problem solved by the computational resources, e.g. *Personnel rostering* in the considered case study. For arbitrary instance $\psi \in \Psi$, it is assumed that $p(\psi, \phi)$ is continuous and increasing function, i.e. $\forall \phi_1, \phi_2 \in \Phi : \phi_1 < \phi_2 \implies p(\psi, \phi_1) < p(\psi, \phi_2)$. The definition allows the normalized processing time function to be non-linear in a general case. A *maximum normalized processing time* of instance ψ is a normalized processing time for quality 1. A *normalized worst case processing time* $WCT(\psi)$ denotes an upper bound of the processing time after which the anytime algorithm, solving the given task, is stopped.

In the whole text, notation $i \in N$ and $j \in M$ is used to denote the tasks and resources, respectively. To highlight that an object is indexed by a task or a resource, we will use superscripts (i) or (j) , respectively.

Each task $i \in N$ is represented by a tuple $(a^{(i)}, rrt^{(i)}, \psi^{(i)})$, where $a^{(i)} \in T$ is an *arrival time* of the task, $rrt^{(i)} \in \mathbb{N}_{>0}$ is a *requested response time* of the task, and $\psi^{(i)} \in \Psi$ is a *task instance*. Roughly, a task is an instance which arrived in the scheduling system. It is assumed that the tasks cannot be preempted. Time $rdd^{(i)} = a^{(i)} + rrt^{(i)}$ represents a *requested due date* of task i , i.e. a soft deadline. The *response time* is defined as the duration between the arrival of the task to the scheduling system and its completion. The requested response time represents the preferred maximum response time set by the client applications.

A parameter *speed* $s^{(j)} \in \mathbb{R}_{>0}$ is defined for each resource j . The speed of resource j can be understood as a constant processing power, i.e. it is the amount of work done by the algorithm running on resource j per time unit. For example, consider an algorithm consisting of one loop which multiplies two numbers in each iteration. A resource with the speed of 1 can make only one iteration per time unit whereas a resource with the speed of 10 can make ten iterations per time unit. By a *normalized processing time* we mean a processing time abstracted from the heterogeneity of the resources, i.e. it represents the processing time on a resource with the speed of 1 and, therefore, it also represents the amount of work to be done to get a solution of the given quality on such a resource.

A *quality function* $q : T \rightarrow \Phi$ assigns a value of the requested quality of solutions to each time $t \in T$. The quality function is not defined by the problem but is part of the decision made by the scheduling system.

If task i is started at time t on resource j for some quality function q , its *completion time* is defined as

$$ct(i, j, t, q) = \min \left\{ t_{min} : t_{min} \in T, t_{min} \geq t, (t_{min} - t)s^{(j)} \geq p(\psi^{(i)}, q(t_{min})) \right\}. \quad (1)$$

The definition can be understood as the shortest time when the amount of work undertaken by an algorithm is greater than or equal to the currently requested amount of work (the requested amount of work equals to the normalized processing time). Note that this definition allows the quality function q to vary over time. If this were not the case, then the completion time could be easily defined as $\left\lceil \frac{p(\psi^{(i)}, \phi)}{s^{(j)}} \right\rceil + t$, where ϕ is a constant quality value.

The *lateness* of task i if started at time t on resource j for some quality function q is defined as

$$L(i, j, t, q) = ct(i, j, t, q) - rdd^{(i)}. \quad (2)$$

The *solution quality* of task i if started at time t on resource j for some quality function q is defined as

$$\varphi(i, j, t, q) = \max \left\{ \phi : \phi \in \Phi, (ct(i, j, t, q) - t)s^{(j)} \geq p(\psi^{(i)}, \phi) \right\}. \quad (3)$$

Again, this definition allows the quality to change over time. The reason for the inequality is to handle the case when the requested amount of work for the maximum quality is less than the actual amount of work done.

A *solution* to problem Π is a tuple $S = (r_S, st_S, q_S)$, where $r_S = (r_S^{(1)}, r_S^{(2)}, \dots, r_S^{(n)})$ is a vector in which $r_S^{(i)} \in M \cup \{\infty\}$ maps task i to some resource or ∞ , $st_S = (st_S^{(1)}, st_S^{(2)}, \dots, st_S^{(n)})$ is a vector in which $st_S^{(i)} \in T \cup \{\infty\}$ maps task i to the start time or ∞ , and $q_S = (q_S^{(1)}, q_S^{(2)}, \dots, q_S^{(n)})$ is a vector of quality functions for each task. Value ∞ represents uninitialized value, i.e. $r_S^{(i)} = \infty$ denotes that task i is not assigned to any resource and $st_S^{(i)} = \infty$ denotes that task i does not have a starting time in solution S . Solution $S = (r_S, st_S, q_S)$ is *feasible* if the following conditions are satisfied for each task $i \in N$:

$$r_S^{(i)} \neq \infty, \quad (4)$$

$$st_S^{(i)} \neq \infty, \quad (5)$$

$$a^{(i)} \leq st_S^{(i)}, \quad (6)$$

$$\forall i, i' \in N : i \neq i' \wedge r_S^{(i)} = r_S^{(i')} \implies ct(i, r_S^{(i)}, st_S^{(i)}, q_S^{(i)}) \leq st_S^{(i')} \vee ct(i', r_S^{(i')}, st_S^{(i')}, q_S^{(i')}) \leq st_S^{(i)} \quad (7)$$

The constraints require that: a task is assigned to some resource (4), (5); a task cannot start before its arrival time (6); and processing of two tasks on the same resource cannot overlap (7). The *set of all feasible solutions* is denoted as \mathcal{S} .

For the feasible solutions the following functions are defined: *average solution quality* (8) and *average normalized lateness* (9)

$$\bar{\varphi}(S) = \frac{1}{n} \sum_{i \in N} \varphi(i, r_S^{(i)}, st_S^{(i)}, q_S^{(i)}), \quad (8)$$

$$\bar{L}(S) = \frac{1}{n} \sum_{i \in N} \frac{L(i, r_S^{(i)}, st_S^{(i)}, q_S^{(i)})}{r_S^{(i)}} \quad (9)$$

The goal in the scheduling problem is to find such a feasible solution which minimizes the average normalized lateness while the average solution quality is maximized, i.e.

$$\begin{aligned} \min \quad & (\bar{L}(S), -\bar{\varphi}(S)) \\ \text{s.t.} \quad & S \in \mathcal{S} \end{aligned} \quad (10)$$

Because the described problem is a multi-objective scheduling problem [13], the outcome is a set of solutions called *Pareto front*. For each solution in the Pareto front holds that it is not dominated by any other solution in the Pareto front. Since the scheduling system has to react in an automatic manner, it needs to select some solution from the Pareto front so that the desired behavior of the system is achieved. Obviously, an increase in the average solution quality leads to increase in the average normalized lateness and vice-versa. Solutions which only optimize the average solution quality or the average normalized lateness are clearly unacceptable since the other objective is ignored. Therefore, solutions which balance the solution quality and the normalized lateness are sought.

One possible way of achieving the balance is to minimize the weighted sum of the objectives. However, due to non-linearity of the processing time functions, such solutions may lead to decreased average solution quality even if the system is not overloaded. Moreover, it is not obvious how to set the weights since the scale of the objectives is different.

We argue that if the tasks are computed within the requested response time, the client applications are sufficiently responsive from the users point of view. Therefore, we propose to maximize the average quality such that, on average, the tasks are processed very close to their requested due date. This requirement can be expressed by pushing the average normalized lateness as close to 0 as possible. This aggregation is correct even if the requested response time is significantly larger than the time needed to find the near optimal solutions since we perform “tail-cutting” on the processing time functions (see Section 5 for more details).

2.1. Extension to online scheduling

The problem described above considers the complete information about tasks and resources. However, in our setting, some quantities are unknown to the scheduler until time t or until some conditions are satisfied (this relates to the online scheduling paradigm where tasks arrive over time [26]):

- The values of function p for instance $\psi^{(i)}$ of task i can be observed only after task i has finished.
- The values $(a^{(i)}, rrt^{(i)}, \psi^{(i)})$ of each task i are unknown until its arrival time $a^{(i)}$.
- The completion time and the solution quality of task i are known only after task i has finished.

On the other hand, the scheduler has full knowledge of the following information: (i) normalized worst case processing time function WCT , (ii) feature functions f , (iii) dimensionality D of the feature vectors, and (iv) set of resources M and speed of each resource (the speed of the resources can be acquired through benchmarking [17]).

3. Scheduling system overview

With respect to the problem statement described in Section 2, we propose a modular architecture of the scheduling system. Each module is implemented as an independent thread. Therefore, the introduced architecture can better utilize the available computational capacity of the server on which the scheduling system is deployed. The proposed architecture is depicted in Figure 3. It consists of three blocks:

1. *Client applications*, which generate tasks and send them to the *scheduling system*.
2. *Scheduling system*, which is responsible for assigning the received tasks from the *client applications* to the available *computational resources*. The *scheduling system* can be further divided into the following *modules*:
 - **Service facade**, which acts as a communication interface between the scheduling system and the client applications. The interface allows the client applications to send tasks, abort tasks and to receive the solution of a previously sent task.

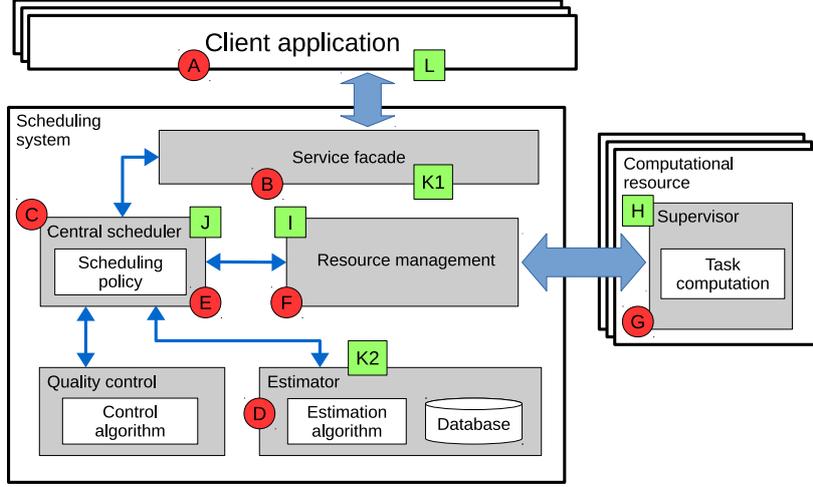


Figure 3: The proposed architecture.

- **Central scheduler**, which stores the received tasks and assigns these tasks to the available resources using a *scheduling policy*. The data structure containing the information about the assignments of the tasks is called a *schedule*. The central scheduler also converts the requested quality of the solutions (given by the **quality control** module) to the actual processing time of each task and sends them to the computational resources through the **resource management**.
 - **Resource management**, which monitors the computational resources and abstracts the communication between the central scheduler and the computational resources, e.g. communication over a socket with remote resources.
 - **Estimator**, which estimates the normalized processing time functions, i.e. it provides $\hat{p} : \Psi \times \Phi \rightarrow \mathbb{N}_{>0}$. These estimations are found by the *estimation algorithm* which uses the *regression model*.
 - **Quality control**, which analyzes the current schedule created by the central scheduler and sets the requested quality of the solutions, i.e. it provides $q_S^{(i)}(t)$. The qualities are found by a *control algorithm*.
3. *Computational resources*, which process the assigned tasks and send their solutions back to the scheduling system. The computational resources can be local threads of the server, remote high-performance resources, etc. On each computational resource, a *supervisor* is executed which communicates with the scheduling system and is responsible for monitoring the computation of a task. We remind the reader that each resource can process only one task at a time.

3.1. Processing of a generated task

To explain how the tasks flow through the system, Figure 3 is used. The letters in red circles represent the order of the modules for an unprocessed task, whereas the letters in green rectangles represent the order of modules for a processed task.

When the **client application** (A) generates task i , the **client application** sends that task to the **central scheduler** through the **service facade** (B). The **central scheduler** (C) inserts that task to an array of *tasks to be estimated*, i.e. tasks for which the normalized processing time function needs to be estimated. Then, the **central scheduler** sends that task to the **estimator** (D). If a regression model is already trained, then the regression model is used to estimate the normalized processing time function for task i . If the regression model is not trained, the normalized processing time function is computed from the normalized worst case processing time. The estimated normalized processing time function is provided to the **central scheduler** (E) which then moves task i from the array of tasks to be estimated to an array of *pending tasks*, i.e. tasks which were received by the scheduling system, are not completed yet and for which

the normalized processing time function is already estimated. The **central scheduler** creates a schedule of the pending tasks on the available resources using the scheduling policy. When some resource j becomes free, i.e. it is not processing any tasks, the scheduling system takes the first task from the schedule of the respective resource j and assigns that task to that resource. The task assignment is performed through the **resource management** (F) module which communicates with the computational resources. When the task is received by the **supervisor** (G) of the resource, the **supervisor** spawns a new thread which executes the received task. The processing time of task i is computed from the requested quality and the speed of resource j . Since the processing time is computed from the requested quality which may change over time, resource j is constantly notified about these changes. The anytime algorithms on the computational resources run approximately for the duration of these processing times.

When the task has finished, the **supervisor** (H) sends the solution back to the **central scheduler** through the **resource management** (I). The **central scheduler** (J) deletes task i from the array of pending tasks and sends the solution to the respective **client application** (L) through the **service facade** (K1). If the progress of the solution quality over time was measured by the task algorithm, it is sent to the **estimator** (K2) which uses it to refine the regression model (see Section 5.1).

4. Scheduling policy and quality control algorithms

If we consider the offline version of our problem without the quality control, then the closest scheduling problem can be represented in Graham's notation [3] as $Qm|r_i|\sum w_i L_i$. This problem considers heterogeneous resources with quantifiable speed and that the tasks arrive in the system at *release time* r_i which is equal to the arrival time $a^{(i)}$ from our problem statement in Section 2. The aim is to find a solution which minimizes the weighted lateness where the weight is defined as $w_i = 1/r_{rt^{(i)}}$. The optimal solution to problem $Qm|r_i|\sum w_i L_i$ is equal to the optimal solution to problem $Qm|r_i|\sum w_i C_i$, the only difference is in the constant in the objective functions. Unfortunately, even the simpler problem $1|r_i|\sum w_i C_i$ is \mathcal{NP} -hard [3], therefore, the problem $Qm|r_i|\sum w_i L_i$ is also \mathcal{NP} -hard.

For the online version (again, without the quality control), the situation is even more complicated [26]; this is due to the lack of knowledge of the tasks that will arrive later from the client application. In such a case, the online algorithm might assign a longer task on an available resource, even though it would be more beneficial to wait if a shorter task arrives very soon. Without prediction of the arrival of future tasks, an offline scheduling rule can be employed to create a schedule of all pending tasks.

Our approach to handling the online scheduling with the quality control is to divide this problem into two steps performed by the central scheduler and the quality control module:

1. Central scheduler: For the maximum quality of the solutions, find a schedule of all pending tasks which minimizes the average normalized lateness. The schedule is found by the scheduling policy and is recreated whenever some *event* occurs (see Section 4.1 for a list of events).
2. Quality control module: For a fixed schedule, find the requested quality of the solution for each task such that the average normalized lateness is close to 0. The quality control procedure is started whenever a new schedule is created by the central scheduler.

To describe the algorithms, some additional notation needs to be introduced:

- In algorithms, an *array* data structure is used. An array is an ordered sequence of some objects. Bracket notation is used for accessing the elements of an array, e.g. if $a = (1, 3, 5)$ is an array, then $a[3] = 5$. The length of an array is computed using a *len* function, e.g. the length of the array a is $len(a) = 3$. To append an element to the end of an array, bracket notation is used with $end + 1$ as the index, e.g. after calling $a[end + 1] \leftarrow 7$, the content of a will be $(1, 3, 5, 7)$.
- An *estimated completion time* of task i started at time t on resource j with fixed quality ϕ is defined as

$$\hat{ct}(i, j, t, \phi) = t + \left\lceil \frac{\hat{p}(\psi^{(i)}, \phi)}{s^{(j)}} \right\rceil. \quad (11)$$

- The number of pending tasks is denoted as $n^{(*)}$ and the number of pending tasks assigned to resource j is denoted as $n^{(j)}$.

4.1. Central scheduler

The policy used for scheduling is described in Algorithm 1. It is a combination of an *Earliest Due Date* (EDD) selection policy and a *Minimal Completion Time* (MCT) assignment policy. Although there are other policies, such as *First In First Out* and *Shortest Task First*, the EDD policy is chosen because it considers the requested due date and it does not suffer from starvation of longer tasks.

The global variables of the central scheduler are *pendingTasks* and *currentlyBeingProcessed*. The *pendingTasks* is the array of pending tasks and *currentlyBeingProcessed* is a boolean array indexed by the tasks where *currentlyBeingProcessed*[i] denotes whether task i is currently being processed by some resource. Once a task is assigned to some resource and the resource starts to process it, the task cannot be moved to another resource. The global variables are kept in the memory for the whole running time of the scheduling system.

The scheduling policy receives two input arguments (in addition to the global variables). The first one, denoted as t , is the time when the event leading to rescheduling occurred. The second one, denoted as S , is the current solution to the scheduling problem.

The scheduling policy returns a schedule of the pending tasks which is an array *sch* indexed by the resources. Each element *sch*[j] of this array is an another array which defines the order of the assigned tasks on this resource.

The policy starts by performing the initialization of the *earliestStartTime* and *sch* arrays. The *earliestStartTime* array represents the earliest start time of the tasks on each resource with respect to the previous assignments. Then, in the loop at line 7, the currently processed tasks on the resources are added to the start of the schedule of each resource. The loop at line 11 ensures that the rest of the pending tasks cannot be assigned in the past. The next part sorts the tasks by their requested due dates, i.e. this is the selection policy. The last loop at line 15 finds a resource for each pending task such that it can finish that particular task at the earliest with respect to the previously assigned tasks, i.e. this is the assignment policy.

Since the environment in which the scheduling system operates is not static, the scheduling system needs to adapt to the changes in this environment. These changes are propagated to the scheduling system as *events*. When the environment changes, the scheduling system receives a corresponding event and runs the scheduling policy to adapt to the new state of the environment. The following events are recognized by the scheduling system (if an additional procedure needs to be performed before the scheduling policy is run, the description of that procedure is also provided):

- New task i was received by the scheduling system: task i is added to the array of pending tasks, i.e.

$$\begin{aligned} \text{currentlyBeingProcessed}[i] &\leftarrow \text{false}; \\ \text{pendingTasks} &\leftarrow \text{pendingTasks} \cup \{i\}; \end{aligned}$$

- The solution for some task i was received by the scheduling system: task i is removed from the array of pending tasks, i.e.

$$\begin{aligned} \text{currentlyBeingProcessed}[i] &\leftarrow \text{false}; \\ \text{pendingTasks} &\leftarrow \text{pendingTasks} \setminus \{i\}; \end{aligned}$$

The complexity of the presented policy is $\mathcal{O}(n^{(*)} \log n^{(*)} + n^{(*)}m)$. The term $n^{(*)} \log n^{(*)}$ is due to sorting of the pending tasks. The term $n^{(*)}m$ is added because it is needed to iterate over all the resources to find a resource which completes the given task at the earliest.

4.2. Quality control

We propose two quality control algorithms: (i) *bisection control* and (ii) *individual control*. Both algorithms are able to handle the overload situations. However, they have different behavior and assumptions.

```

1 Function EDDMCT( $t, pendingTasks, S, currentlyBeingProcessed$ )
2   /* Initialization */
3   foreach  $j \in M$  do
4      $earliestStartTime[j] \leftarrow 0$ ;
5      $sch[j] \leftarrow$  empty array;
6   /* Add the already assigned tasks to the schedule */
7   foreach  $i \in pendingTasks$  do
8     if  $currentlyBeingProcessed[i]$  then
9        $earliestStartTime[r_S^{(i)}] \leftarrow \widehat{ct}(i, r_S^{(i)}, st_S^{(i)}, 1)$ ;
10       $sch[r_S^{(i)}][end + 1] \leftarrow i$ ;
11  foreach  $j \in M$  do
12     $earliestStartTime[j] \leftarrow \max(earliestStartTime[j], t)$ ;
13   $sorted \leftarrow$  array of sorted tasks from  $pendingTasks$  in ascending order by (i)  $rdd^{(i)}$ , if equal then
14    (ii)  $\widehat{p}(\psi^{(i)}, q_S^{(i)})$ , if equal then (iii)  $a^{(i)}$ , if equal then (iv) randomly;
15  /* Assignment policy */
16  foreach  $k = 1, \dots, len(sorted)$  do
17     $i \leftarrow sorted[k]$ ;
18    if  $currentlyBeingProcessed[i] = false$  then
19       $r_S^{(i)} \leftarrow \arg \min_{j \in M} \widehat{ct}(i, j, earliestStartTime[j], 1)$ ;
20       $st_S^{(i)} \leftarrow earliestStartTime[r_S^{(i)}]$ ;
21       $earliestStartTime[r_S^{(i)}] \leftarrow \widehat{ct}(i, r_S^{(i)}, st_S^{(i)}, 1)$ ;
22      if  $len(sch[r_S^{(i)}]) = 0$  then
23        /* The resource starts processing the task immediately */
24         $currentlyBeingProcessed[i] \leftarrow true$ ;
25         $sch[r_S^{(i)}][end + 1] \leftarrow i$ ;
26  return  $sch$ ;

```

Algorithm 1: EDDMCT (Earliest Due Date + Minimal Completion Time) scheduling policy

4.2.1. Bisection control

The bisection control algorithm is described in Algorithm 2. The algorithm tries to find one global quality ϕ for all the tasks over all the resources such that the average normalized lateness of the tasks in the current schedule is close to 0. Equivalently, the algorithm tries to find a root of the average normalized lateness function for the fixed assignment and order of the tasks defined in sch , i.e. output of Algorithm 1. For efficiency and ease of implementation, a well-known *bisection method* is used as the root-finding method. A bisection method iteratively divides the quality interval Φ into two halves. The search continues only in the half where the root lies. Algorithm 2 also uses Algorithm 3 which recomputes the duration of the tasks in the current schedule for the given fixed quality ϕ and returns the average normalized lateness of the tasks in the current schedule.

In addition to the already introduced input arguments for the scheduling policy, the control algorithm has three new input arguments. The first one, denoted as sch , is the schedule created by Algorithm 1. The second one, denoted as $maxIters$, is the maximum number of iterations of the control algorithm. The third one, denoted as ϕ , is the minimum quality of the solutions. Both arguments $maxIters$ and ϕ are set by the administrator of the scheduling system depending on the desired behavior. The $maxIters$ argument controls the performance of the control algorithm: the higher number of iterations will result in a more accurate quality, but the running time of the control algorithm will be higher. Based on our preliminary

```

1 Function BisectionControl( $t, S, sch, maxIters, \underline{\phi}$ )
2    $averageNormalizedLateness \leftarrow ScheduleRecomputation(t, sch, 1)$ ;
3   if  $averageNormalizedLateness \leq 0$  then return;
4    $averageNormalizedLateness \leftarrow ScheduleRecomputation(t, sch, \underline{\phi})$ ;
5   if  $averageNormalizedLateness \geq 0$  then return;
6    $iter \leftarrow 0$ ;  $lb \leftarrow \underline{\phi}$ ;  $ub \leftarrow 1$ ;  $\phi \leftarrow 1$ ;
7    $averageNormalizedLateness \leftarrow 1$ ;
8   while  $iters < maxIters$  do
9     if  $averageNormalizedLateness > 0$  then
10      |  $ub \leftarrow \phi$ ;
11      else
12      |  $lb \leftarrow \phi$ ;
13       $\phi \leftarrow \frac{lb+ub}{2}$ ;
14       $averageNormalizedLateness \leftarrow ScheduleRecomputation(t, sch, \phi)$ ;
15       $iter \leftarrow iter + 1$ ;

```

Algorithm 2: Bisection control algorithm

experiments, for the majority of the cases, the sufficient number of iterations is 30.

The complexity of Algorithm 2 is $\mathcal{O}(maxIters \cdot (m + n^{(*)}))$, where $m + n^{(*)}$ is due to schedule recomputation algorithm.

4.2.2. Individual control

The individual control algorithm is described in Algorithm 4. The quality is computed for each task and resource independently, and the algorithm assumes that the normalized processing time of each task i can be approximated by a linear function $q_S^{(i)}(t) \cdot \widehat{p}(\psi^{(i)}, 1)$. To illustrate the concept behind the algorithm, we can assume that: (i) all tasks from N are processed on one resource j , (ii) the order of assignments is given by the value of $i \in N$, and (iii) resource j has no idle times between processing each task. First, expand Equation (9) for the average normalized lateness using the estimated completion time (11)

$$\begin{aligned}
\frac{1}{n} \sum_{i \in N} \frac{\widehat{ct}(i, j, st_S^{(i)}, q_S^{(i)}(t)) - rdd^{(i)}}{rrt^{(i)}} &= \frac{1}{n} \sum_{i \in N} \frac{st_S^{(i)} + q_S^{(i)}(t) \frac{\widehat{p}(\psi^{(i)}, 1)}{s^{(j)}} - rdd^{(i)}}{rrt^{(i)}} \\
&= \frac{1}{n} \sum_{i \in N} \frac{st_S^{(1)} + \left(\sum_{k=1}^i q_S^{(k)}(t) \frac{\widehat{p}(\psi^{(k)}, 1)}{s^{(j)}} \right) - rdd^{(i)}}{rrt^{(i)}} \tag{12} \\
&= \left(\sum_{i \in N} \sum_{k=1}^i q_S^{(k)}(t) \frac{\widehat{p}(\psi^{(k)}, 1)}{s^{(j)} \cdot n} \frac{1}{rrt^{(i)}} \right) + \left(\sum_{i \in N} \frac{st_S^{(1)} - rdd^{(i)}}{n \cdot rrt^{(i)}} \right).
\end{aligned}$$

The second term of Equation (12) on the right-hand side is a constant, i.e. its value does not depend on the values $q_S^{(i)}(t)$. This term is denoted as *cons* in Algorithm 4. To further simplify the first term, the sums for each i are expanded and the multipliers of each $q_S^{(i)}(t)$ are collected. By doing so, a *weight* of each task is obtained

$$w^{(i)} = \frac{\widehat{p}(\psi^{(i)}, 1)}{s^{(j)} \cdot n} \sum_{k=i}^n \frac{1}{rrt^{(k)}}, \tag{13}$$

with which the average normalized lateness from Equation (12) can be rewritten as

$$\left(\sum_{i \in N} q_S^{(i)}(t) \cdot w^{(i)} \right) + \left(\sum_{i \in N} \frac{st_S^{(1)} - rdd^{(i)}}{n \cdot rrt^{(i)}} \right). \tag{14}$$

```

1 Function ScheduleRecomputation( $t, sch, \phi$ )
2    $sumNormalizedLateness \leftarrow 0$ ;  $numTasks \leftarrow 0$ ;
3   foreach  $j \in M$  do
4      $earliestStartTime[j] \leftarrow 0$ ;
5     foreach  $k = 1, \dots, len(sch[j])$  do
6        $i \leftarrow sch[j][k]$ ;
7        $q_S^{(i)}(t) \leftarrow \phi$ ;
8       if  $k \neq 1$  then
9          $st_S^{(i)} \leftarrow earliestStartTime[j]$ ;
10       $earliestStartTime[j] \leftarrow \hat{ct}(i, j, st_S^{(i)}, \phi)$ ;
11      if  $k = 1$  then
12         $earliestStartTime[j] \leftarrow \max\{t, earliestStartTime[j]\}$ ;
13       $sumNormalizedLateness \leftarrow sumNormalizedLateness + \frac{earliestStartTime[j] - rdd^{(i)}}{rrt^{(i)}}$ ;
14     $numTasks \leftarrow numTasks + len(sch[j])$ ;
15  if  $numTasks = 0$  then
16    return 0;
17  else
18    return  $\frac{sumNormalizedLateness}{numTasks}$ ;

```

Algorithm 3: Schedule recomputation

From this equation, it can be seen that the tasks that contribute the most to the average normalized lateness are those with the highest weights. The idea of the individual control algorithm is to compress greedily the tasks with the highest weights until the average normalized lateness is close to zero.

The algorithm starts by computing the weight of each task and the constant part $cons$ (see line 10). Then, the tasks are sorted non-increasingly by their weights (see line 15) and the algorithm proceeds by greedy compression of the tasks with the highest weights (see line 17). The compression is at line 31 where the qualities for all tasks except i are fixed and the algorithm is trying to find the root of Equation (14). The last part of the algorithm recomputes the start time of each task using the computed qualities (see line 34).

There are two complications which need to be considered when implementing this idea. The first complication is that if some task is currently being processed by a resource, generally it cannot be compressed to ϕ because the solution found until time t can already be of better quality than ϕ , i.e. the quality of the current solution is a new lower bound for the quality compression. Therefore, the quality of the current solution must be reflected in the algorithm when compressing the first task (see line 28). The second complication arises when the requested quality for the first task is set to one and the solution is not received within the estimated completion time, e.g. the solution is being sent from a resource to the scheduling system. In such a case, there is an idle time between the estimated completion time of the first task and time t called the *phantom time*, which needs to be considered when computing the average normalized lateness (see lines 14

and 23).

```

1 Function IndividualControl( $t, S, sch, j, \phi$ )
2   if  $len(sch[j]) = 0$  then return;
3    $usePhantom \leftarrow false; phantomTime \leftarrow 0;$ 
4   if  $t \geq \widehat{ct}(sch[j][1], j, st_S^{(sch[j][1])}, 1)$  then
5      $usePhantom \leftarrow true;$ 
6      $phantomTime \leftarrow t - \widehat{ct}(sch[j][1], j, st_S^{(sch[j][1])}, 1);$ 
7   /* The constant part of the average normalized lateness */
8    $cons \leftarrow 0;$ 
9   /* Compute the weights and the constant part */
10  foreach  $k = 1, \dots, len(sch[j])$  do
11     $i \leftarrow sch[j][k];$ 
12     $q_S^{(i)}(t) \leftarrow 1;$ 
13     $w[i] = \frac{\widehat{p}(\psi^{(i)}, 1)}{s^{(j)} \cdot len(sch[j])} \sum_{l=k}^{len(sch[j])} \frac{1}{rrt^{(sch[j][l])}};$ 
14     $cons \leftarrow cons + \frac{st_S^{(sch[j][1])} + phantomTime - rdd^{(i)}}{len(sch[j]) \cdot rrt^{(i)}};$ 
15   $sorted \leftarrow$  array of tasks from  $sch[j]$  sorted decreasingly by  $w$ ;
16  /* Compress the tasks in order given by the sorted array */
17  foreach  $k = 1, \dots, len(sorted)$  do
18    if  $\left( \sum_{l=1}^{len(sch[j])} q_S^{(sch[j][l])}(t) \cdot w[sch[j][l]] \right) + cons$  is negative or is near to zero then
19      /* Stop if the average normalized lateness is */
20      /* negative or is near to zero */
21      break;
22     $i \leftarrow sorted[k];$ 
23    if  $i = sch[j][1] \wedge usePhantom$  then
24      /* The first task cannot be compressed further */
25      continue;
26    if  $i = sch[j][1]$  then
27      /* The current progress of the task should not be ‘lost’ */
28       $qualityLowerBound \leftarrow \max \left\{ \phi, \frac{(t - st_S^{(i)})s^{(j)}}{\widehat{p}(\psi^{(i)}, 1)} \right\};$ 
29    else
30       $qualityLowerBound \leftarrow \phi;$ 
31     $q_S^{(i)}(t) \leftarrow \max \left\{ qualityLowerBound, \frac{-cons - \sum_{l=1, l \neq i}^{len(sch[j])} q_S^{(sch[j][l])}(t) \cdot w[sch[j][l]]}{w[i]} \right\};$ 
32   $earliestStartTime \leftarrow 0;$ 
33  /* Computation of the start times using the found qualities */
34  foreach  $k = 1, \dots, len(sch[j])$  do
35     $i \leftarrow sch[j][k];$ 
36    if  $k \neq 1$  then
37       $st_S^{(i)} \leftarrow earliestStartTime;$ 
38     $earliestStartTime \leftarrow \widehat{ct}(i, j, st_S^{(i)}, q_S^{(i)}(t));$ 
39    if  $k = 1$  then
40       $earliestStartTime \leftarrow \max\{t, earliestStartTime\};$ 

```

Algorithm 4: Individual control algorithm

In addition to the already introduced input arguments for the bisection control, the independent control algorithm has one new input argument j which denotes the resource for which the qualities are currently computed.

The complexity of the algorithm is $\mathcal{O}(n^{(j)} \log n^{(j)})$ because the complexity is dominated by the sorting of the tasks. From Theorem 1, the total complexity of computing Algorithm 4 for all resources is $\mathcal{O}(n^{(*)} \log n^{(*)})$.

Theorem 1. *The total worst case complexity of computing Algorithm 4 for all resources is $\mathcal{O}(n^{(*)} \log n^{(*)})$.*

PROOF. See Appendix A.

5. Estimation of the normalized processing time functions

Since normalized processing time function p is not known *a priori*, the scheduling system needs to estimate it². This estimation can be used by the scheduler instead of the worst-case estimates to find better assignments.

Estimation is based on the assumption that p , for the given instance, can be approximated by a *piecewise linear function* which is parameterized by *approximated parameters* $\tilde{\mathbf{y}}$. Each parameter \tilde{y}_k represents the normalized processing time of the endpoint of k -th segment. An endpoint is a point where two neighboring segments of a piecewise linear function meet. The approximation is made due to performance reasons - a function approximation with a few parameters can be more efficiently estimated than the whole p . However, since $\tilde{\mathbf{y}}^{(i)}$ are not known *a priori* (because p is also not known *a priori*), the question remains how to estimate them when task i arrives in the scheduling system. To solve this problem, a *regression analysis* is used. Using a set of previously collected parameters $\tilde{\mathbf{y}}$, a *regression model* is *trained*. That regression model is then able to estimate $\tilde{\mathbf{y}}^{(i)}$ for the given task i , where the *estimated parameters* for task i are denoted as $\hat{\mathbf{y}}^{(i)}$. Figure 4 summarizes the flow of the normalized processing time estimation.

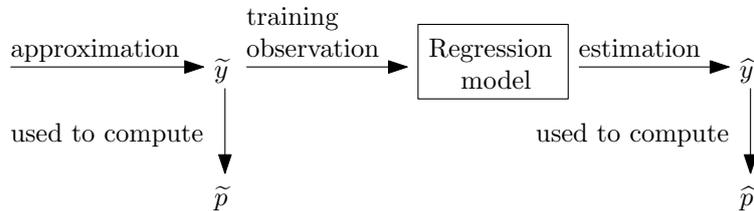


Figure 4: The flow of the normalized processing time function estimation.

The training of the regression model can be performed:

- *offline*, i.e. before the scheduling system is used in production.
- *online*, i.e. after enough observations are collected at runtime.
- using a combined approach.

In the combined approach, the estimator starts with an initial regression model which is iteratively refined using information acquired online. The offline model can be built by sampling the observation space, i.e. the space of tuples representing the features and the approximated parameters of the instances. However, this method assumes that we are provided with such samples. The online model does not need such samples

²For simplicity, we assumed in this work that the instances are only of one optimization problem, i.e. personnel rostering. However, it is possible that the scheduling system would have to process more optimization problems. In such a case, the processing time functions would be indexed by the problems and the estimation would be performed for each problem individually.

beforehand. However, its disadvantage is that until enough training observations are collected, the estimator must use the default parameters such as the worst case normalized processing time. All approaches also fail to provide satisfactory estimations if the observation space is insufficiently sampled, e.g. observations used for training the regression model are sampled from a different region of the observation space than the observations to estimate.

5.1. Collecting training observations

This Subsection describes how the training observations are collected and processed. When the online learning is employed, the steps are performed during the runtime of the scheduling system. If the offline learning is used, the steps are performed on the sampled observations before the scheduling system is used in production. A new training observation from task i is created in the following steps:

1. The scheduling system assigns task i to some resource j with normalized processing time equal to the normalized worst case processing time of that task. The normalized worst case processing time represents some fixed time after which the algorithm is stopped even though the solution could still be improved. However, it is our assumption that the improvements after the normalized worst case processing time are insignificant.
2. The algorithm solving the task must record how the value of the objective for the best-known solution evolves through time. This progress is recorded in a *objective curve* $Z^{(i)}$ of task i which is a sequence of tuples $\left((t_1^{(i)}, z_1^{(i)}), (t_2^{(i)}, z_2^{(i)}), \dots, (t_{h_i}^{(i)}, z_{h_i}^{(i)}) \right)$, where $z_k^{(i)}$ is a objective value of the best known solution until time $t_k^{(i)}$ and h_i is a number of collected tuples. Both $z_k^{(i)}$ and $t_k^{(i)}$ are assumed to be increasing sequences, i.e. it is assumed that the objective function is maximized. The objective function can also be minimized, but then the algorithm solving the task is responsible for transforming the decreasing objective curve to an increasing one.
3. When the task is finished, resource j sends the solution and objective curve $Z^{(i)}$ back to the central scheduler which sends the objective curve to the estimator.
4. The estimator rescales times $t_k^{(i)}$ in the objective curve by the estimated speed of resource j to the normalized processing time $t_k^{(i)} s^{(j)}$.
5. The objective curve is trimmed to retain only the times $t_k^{(i)} \leq WCT(\psi^{(i)})$.
6. The objective curve is trimmed to retain only the part where the approximated slope of the objective value is higher than some threshold, i.e. this step performs the “tail cutting”. The slope for each time $t_k^{(i)}$ is computed as

$$slope_k = \frac{z_{k+1}^{(i)} - z_k^{(i)}}{t_{k+1}^{(i)} - t_k^{(i)}} \quad (15)$$

and the threshold is computed as $0.05 \cdot \max_k slope_k$. This is the last step where the objective curve is trimmed; let us denote the index of the last remaining tuple from the objective curve as k_{max} .

The reason for this step is that even if the normalized worst case processing time represents the stopping time of the algorithm, it does not mean that the solution of the maximum quality cannot be found much earlier or that all solutions found after some threshold time are significantly better than the solution found by the threshold time. Consider Figure 5 which illustrates this idea. The threshold time represents the time after which all measurement are trimmed because the improvements of the solution are not significant. The best-known solution until this threshold time is then declared as the solution of the maximum quality.

7. objective values $z_k^{(i)}$ are linearly scaled to $[0, 1]$ using min-max normalization. The normalized values are denoted as $\phi_k^{(i)}$.
8. The curve from the previous step represents the progress of the normalized processing time on the solution quality. The number of tuples in this curve can be large and, therefore, a compact representation by the approximated curve is used. For our purposes, a piecewise linear function with ten segments is used. The endpoints of the qualities in this piecewise linear function are fixed to $0.1, 0.2, \dots, 0.9, 1$. To

acquire the approximated parameters $\tilde{\mathbf{y}}^{(i)}$, a curve from the previous step is sampled along those fixed endpoints, e.g. $\hat{y}_3^{(i)}$ represents the normalized processing time needed to find the solution of quality 0.3. A linear interpolation between the neighboring qualities is used if the value of the given fixed quality does not exist in the curve.

9. A new training observation $(\mathbf{x}^{(i)}, \tilde{\mathbf{y}}^{(i)})$ is added to the database of the training observations, where $\mathbf{x}^{(i)} = f(\psi^{(i)})$ is the feature vector of task i .

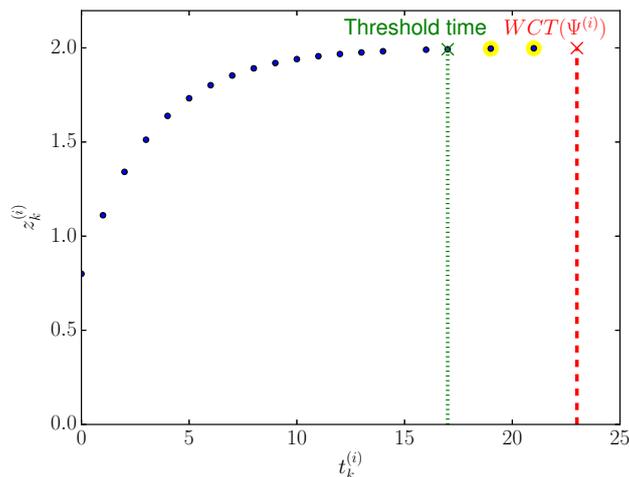


Figure 5: An example of a measured objective curve with highlighted worst case processing time (WCT) and the threshold time denoting the processing time for solution quality of 1. The trimmed points are highlighted by larger yellow circles.

5.2. Regression model training and estimation of unseen tasks

When a sufficient number of training observations is collected, a regression model is trained. The regression model g maps the feature vectors $\mathbf{x}^{(i)} = f(\psi^{(i)})$ to the estimated parameters, i.e. $\hat{\mathbf{y}}^{(i)} = g(\mathbf{x}^{(i)})$. The found parameters are then used to construct the estimated normalized processing time function. Since a piecewise linear function is used for approximating the objective curve, the same function type is also used for the estimation. Therefore, the coordinates of the endpoints in the estimated normalized processing time function are $(0.1 \cdot k, \hat{y}_k^{(i)})$, where $k = 1, \dots, 10$.

Since most regression methods learn only one output, an independent regression model is trained for each quality endpoint $0.1 \cdot k$ and outputs of those models are then combined into one function. However, this approach may result in a non-monotonic function; this can be corrected by taking a running maximum of the processing time values.

5.2.1. Estimation methods

In the previous Subsection a procedure for collecting training observations using an approximation by a piecewise linear function is described. This approach is called *full estimation*. For the experiments in Section 6, the following alternative estimation methods are also considered:

1. *measured processing time*: No estimation is employed, the scheduling system has full knowledge of the processing time function (the function is acquired by running all instances before the scheduling system is run and measuring their processing time). Since in most cases this knowledge is unavailable, such scenario is unrealistic in practice. However, in experiments such scenario allows us to determine how the performance of the scheduling system differs from a more realistic scenario where the estimation is used.
2. *linear estimation*: Instead of using a piecewise linear function to approximate the measured curve, a simple linear function can be used. In such case, only one point needs to be estimated to obtain the estimated processing time function.

All approaches are illustrated in Figure 6.

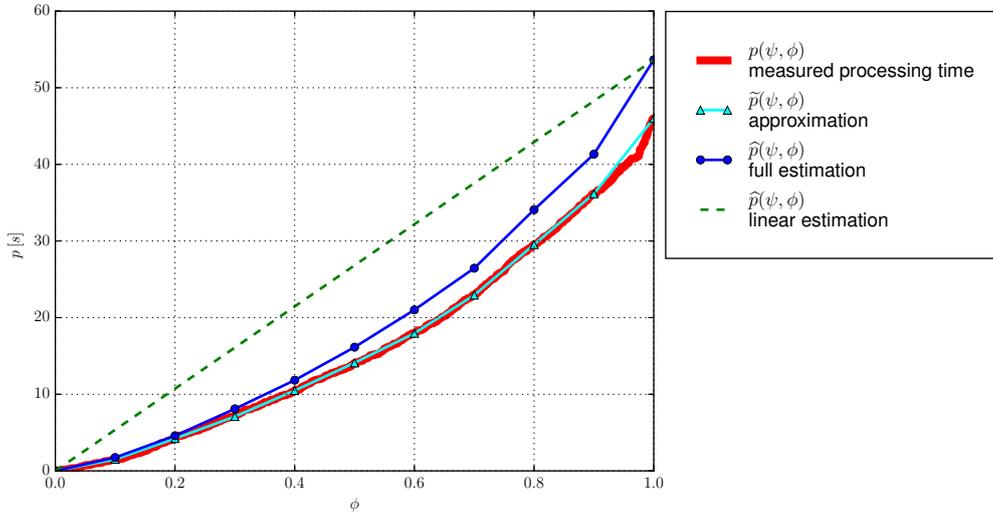


Figure 6: An example of the measured processing time (drawn in red, solid line), its approximation (drawn in cyan, solid line with triangles), its full estimation (drawn in blue, solid line with points) and its linear estimation (drawn in green, dashed line). The triangles and points in approximation and full estimation, respectively, represent the endpoints of the piecewise linear function. The estimation is found using the k -nn method.

6. Experiments

In this section, it is verified that our proposed scheduling system can keep the average normalized lateness near 0. It is shown that if the quality control is disabled, the lateness could be 60 times larger than the requested lateness which is unacceptable.

First, the experimental results for the offline estimation of the processing time of real-world instances from a domain of personnel rostering are provided. Then, the instances from the previous experiment and their estimations are used for the experiments with the scheduling system. The computer, on which the scheduling system run, has an Intel Core i7-3520M @ 2.90GHz processor and 8 GB of RAM.

6.1. Processing time estimation

6.1.1. Experimental setup

As a task type for experiments, an *automatic roster design* from the domain of personnel rostering is used. Initially, 13 features of the rostering instances were considered (some similar to ones described in [23]), but after performing the sequential forward selection, 8 features were obtained from which the following features are the most influential: the number of employees, the number of days, the number of required shifts to assign and roster size (the number of employees multiplied by the number of days). Since each feature has a different scale of values, the features are normalized using *z-score normalization* so that each feature has the same weight.

To generate the dataset, 500 observations were generated which differed in number of the employees (5-30, sampled uniformly), number of days in a roster (7, 14 and 31, sampled uniformly), weekly workload of the employees (20 and 40 hours, sampled uniformly) and workload coverage (ratio between the number of required shifts to the number of shifts computed from the workload of the employees). All observations had three types of shifts to assign: early, late and night (each shift was 8 hours long). As a *training set*, 300 observations were randomly selected from the original set of 500 observations; the rest was used as a *testing set*. The training set was used for feature selection, parameter tuning of the regression methods and

training the estimation model. The testing set was used to determine the performance of the learned model on unknown data.

Two methods were considered for the estimation: (i) *k-nearest neighbors* (abbreviated as *k-nn*) and (ii) *regression trees*. For the *k-nearest neighbors* method, the Euclidean distance was chosen as a distance metric and the number of neighbors was set to 7. For the regression tree method, the minimum number of observations in the branch nodes was set to 10 and the minimum number of observations in the leaf nodes was set to 1.

6.1.2. Results

The results of the estimation experiment are reported in Table 1 and in Figure 7. As an error metric, an *absolute percentage error* is used and is defined as

$$ape(p, \hat{p}, \phi, \psi) = \left| \frac{\hat{p}(\psi, \phi) - p(\psi, \phi)}{p(\psi, \phi)} \right| \cdot 100\%. \quad (16)$$

The absolute percentage error for the given solution quality ϕ and instance ψ is an error ratio between the normalized processing time of instance ψ for quality ϕ and the estimated normalized processing time of instance ψ for quality ϕ . The qualities in Table 1 correspond to the endpoint coordinates of a piecewise linear function. Figure 7 shows the absolute percentage error of each testing observation for maximum quality.

Notice that for higher qualities the absolute percentage error of the majority of observations (0.75%) is around 20% for both methods. For this quantile, the *k-nn* method gives better estimations with the exception of the 0.2 quality. The higher estimation error is achieved when the processing time is low, as can be seen in Figure 7. This is not a considerable issue, because the absolute error, i.e. the absolute value of the difference between y and \hat{y} , of these short instances is small. Therefore, the overall effect on the scheduling system is also small if the system processes both short and long tasks. To overcome the problem of underestimation of short tasks, the minimum processing time can be set to some fixed lower bound, e.g. 1 s. Because the *k-nn* method achieves lower estimation error on the majority of the observations, the *k-nn* method is used in the following experiments of the scheduling system.

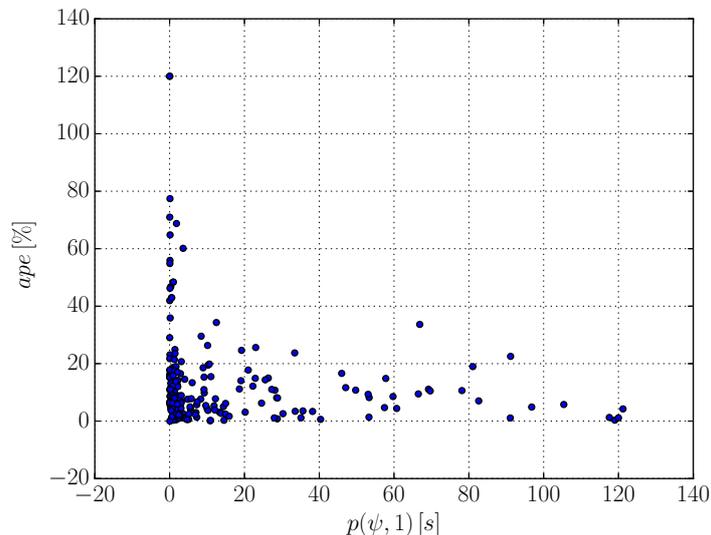


Figure 7: The relationship between the absolute percentage error and the maximum normalized processing time. Each point represents one observation estimated using the *k-nn* method. The total number of observations is 200, i.e. the whole testing set.

| Quality | Method | Absolute percentage error as quantiles [%] | | | | | |
|---------|-----------------|--------------------------------------------|------------|-------------|-------------|-------------|--------------|
| | | 0.05 | 0.25 | 0.5 | 0.75 | 0.95 | 1.00 |
| 0.1 | regression tree | 2.0 | 7.9 | 19.8 | 34.6 | 78.3 | 172.0 |
| | knn | 1.4 | 7.2 | 18.1 | 32.1 | 67.3 | 186.3 |
| 0.2 | regression tree | 1.2 | 6.2 | 13.7 | 24.5 | 49.1 | 146.1 |
| | knn | 1.3 | 6.6 | 13.1 | 28.6 | 60.5 | 170.2 |
| 0.3 | regression tree | 1.4 | 7.1 | 14.8 | 25.6 | 52.7 | 140.3 |
| | knn | 1.0 | 6.2 | 13.1 | 24.8 | 63.1 | 132.4 |
| 0.4 | regression tree | 0.8 | 5.4 | 11.0 | 22.3 | 46.8 | 141.0 |
| | knn | 1.4 | 5.7 | 10.8 | 20.3 | 66.2 | 117.8 |
| 0.5 | regression tree | 0.9 | 4.7 | 9.8 | 21.4 | 51.5 | 136.4 |
| | knn | 0.6 | 4.9 | 10.2 | 18.0 | 68.6 | 117.5 |
| 0.6 | regression tree | 0.9 | 4.5 | 9.9 | 22.0 | 49.8 | 131.6 |
| | knn | 1.2 | 4.7 | 9.9 | 17.3 | 66.7 | 121.6 |
| 0.7 | regression tree | 0.8 | 5.4 | 9.6 | 21.6 | 47.1 | 133.6 |
| | knn | 1.1 | 4.5 | 9.2 | 16.9 | 65.2 | 108.9 |
| 0.8 | regression tree | 1.0 | 4.2 | 10.9 | 22.1 | 51.1 | 131.8 |
| | knn | 0.7 | 3.8 | 9.7 | 16.7 | 68.7 | 92.4 |
| 0.9 | regression tree | 0.8 | 4.5 | 10.8 | 21.1 | 61.1 | 131.6 |
| | knn | 1.1 | 4.3 | 8.5 | 16.6 | 66.1 | 90.4 |
| 1.0 | regression tree | 1.0 | 5.6 | 11.9 | 23.1 | 69.1 | 137.0 |
| | knn | 0.6 | 3.9 | 8.0 | 15.1 | 48.3 | 120.0 |

Table 1: Absolute percentage error of the learned models on the testing set for different qualities. The values in bold represent the best value among all regression methods. The most interesting values are in the column for the 0.75 quantile which represents the majority of the observations.

Figure 6 shows an example of one observation with its measured processing time, approximated processing time function and estimated processing time functions (full and linear estimation). The estimated processing time overestimates the measured processing time function which is better than if the processing time functions were underestimated. To explain this, assume that there are two tasks i, i' and the maximum normalized processing time of both tasks is 100 ms . The estimated maximum normalized processing time of task i is 120 ms (i.e. overestimation) and the estimated maximum normalized processing time of task i' is 80 ms (i.e. underestimation). Next, assume that for task i' the normalized processing time of 80 ms relates to quality 0.8. When the scheduling system is not overloaded, i.e. the requested quality of the solution is 1, the actual quality of the solution for task i would be 1 because the processing time is equal to 120 ms which means that the algorithm is processing task i for more time than is required to get the solution of quality 1. On the other hand, the actual quality of the solution for task i would be 0.8 because the processing time equals to 80 ms which means that the algorithm is processing task i' for less time than is required to get the solution of quality 1. Therefore, underestimation is undesirable if, most of the time, the scheduling system is not overloaded.

6.2. Experiments with the scheduling system

In the following experiments, the ability of the scheduling system to keep the average normalized lateness near 0 is analyzed. The instances from the previous experiment are used.

In the experiments, the proposed bisection and individual control algorithms (see Section 4.2) are compared to alternative control algorithms:

1. *Max quality*: The quality control is disabled, i.e. the requested quality is set to 1 for all tasks. However, due to imprecision in the execution time estimation, the actual quality of the returned solutions does not have always to be 1.
2. *Min quality*: The requested quality is set to some minimum quality which is denoted as $\underline{\phi}$.

3. *Random control*: Before a task is assigned to some resource, the requested quality of that task is fixed to some randomly sampled value from the uniform distribution $\mathcal{U}(\underline{\phi}, 1)$.
4. *Naive*: When the overload is detected, the algorithm iteratively sets the lowest possible quality to tasks which are currently being processed and which has been processed for the longest time until the average normalized lateness is less or equal to 0.

6.2.1. Experimental setup

In the experiments, 20 heterogeneous resources are available to the scheduling system. The speed of each resource was sampled randomly from the uniform distribution $\mathcal{U}(1, 3)$. The minimum quality $\underline{\phi}$ was set to 0.2.

To generate the requested response time for each task, the instances of the tasks were split to disjoint bins with the width of 1 second by their maximum normalized processing time. For each instance in each bin_k , the requested response time was randomly sampled from uniform distribution $\mathcal{U}(1.5k, 3k)$. The advantage of this approach is that the requested response time is not directly dependent on the maximum processing time of the tasks.

To create a workload, the instances were split into three groups by their maximum processing time:

1. the maximum processing time is at most 2 s.
2. the maximum processing time is greater than 2 s and less than or equal to 20 s.
3. the maximum processing time is greater than 20 s.

For each group, six client applications (i.e. 18 client applications in total) were created which generated tasks at some rate and sent them to the scheduling system. The inter-arrival time of the tasks, i.e. the difference between the arrival time of consecutive tasks, was sampled from the exponential distribution with a mean of 100 ms. In total, 600 tasks were generated from the first group, 510 from the second group and 420 from the third group. Therefore, the total number of generated tasks was $n = 1430$.

6.2.2. Evaluation of the quality control algorithms

The results of each scenario, i.e. a combination of a control algorithm and an estimation method, are depicted in a figure consisting of three subfigures ordered from top to bottom: (i) the number of pending tasks in the system in each time unit, (ii) the normalized lateness of the solutions, and (iii) the quality of the solutions.

The results for the situation when the quality control is disabled (i.e. *Max quality* control algorithm) and the scheduler is using the measured processing time are shown in Figure 8. It can be clearly seen that the scheduling system starts to be overloaded around 40 s which results in a higher normalized lateness. For some tasks, the lateness is 60 times larger than the requested lateness which is unacceptable.

The situation when the bisection control is enabled and the scheduler is using the measured processing time is demonstrated in Figure 9 (notice that the scale of the normalized lateness is different from Figure 8). The control algorithm detects the overload and decreases the requested quality of the solutions so that the normalized lateness is around 0. The requested quality of the solutions gradually increases as the overload decreases. In Figure 10, the results for the same control method with the full estimation are shown. Notice that the overall shape of the requested quality of the solutions is similar to Figure 9. Due to the inexact estimation, the spread of the quality is larger, however even in such a situation the response time of the tasks is still maintained at an acceptable level.

The results for the individual control with the measured processing time are shown in Figure 11 (the individual control with the full estimation is not included because the effect of the estimation is not clearly visible as with the bisection control). The individual control algorithm is also able to keep the normalized lateness around 0. From the quality of the solutions, it can be seen that the bisection and individual control algorithms behave differently. Whereas the bisection control sets the same quality for all the tasks, the independent control can specifically address those tasks whose contribution to the average normalized lateness is the highest. If the minimum quality were set to 0, the independent control would behave similarly to the admission control, i.e. it would drop the tasks with the highest contribution to the average normalized lateness.

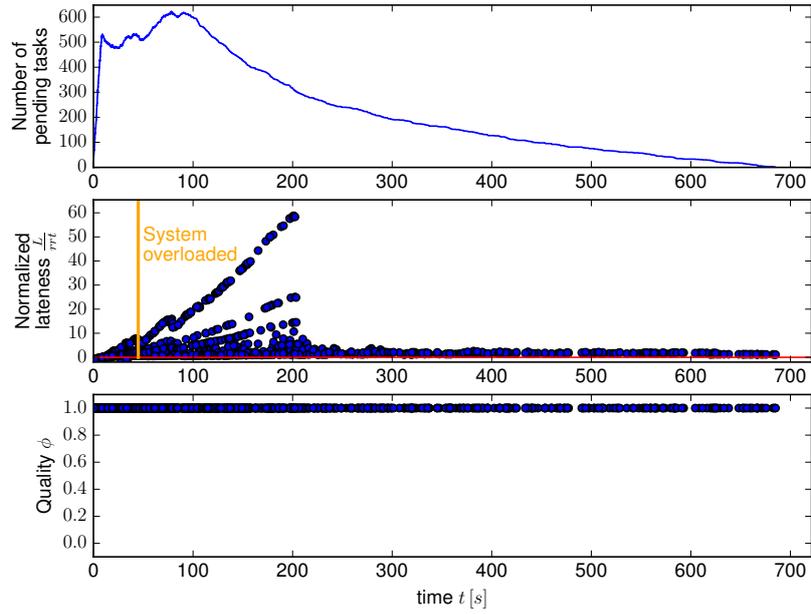


Figure 8: The dependence of the normalized lateness and the solution quality of tasks on the workload when the quality control is disabled (i.e. *Max quality* control algorithm) and measured processing time is used.

The last Figure 12 illustrates the situation when the naive control with the measured processing time is used. Similarly to the individual control, the naive control sets the quality for each task and is also able to keep the response time around the acceptable level.

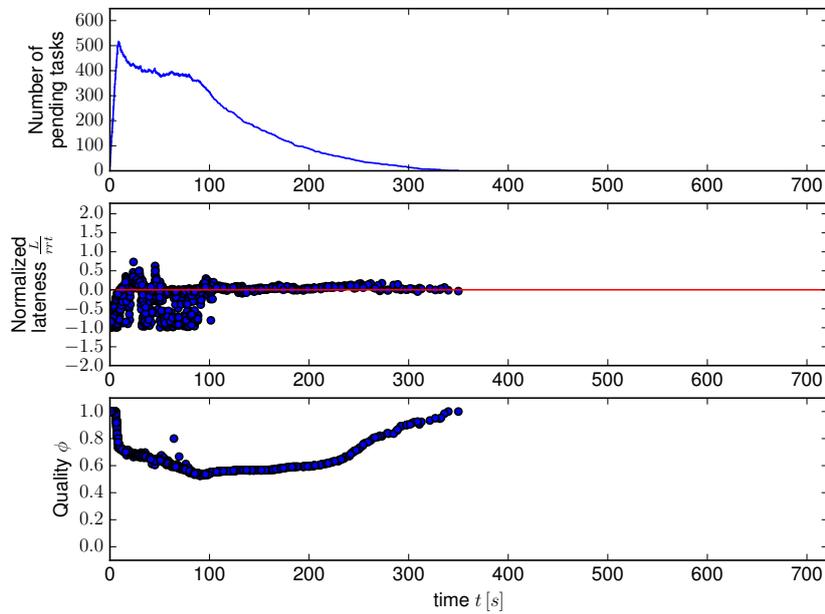


Figure 9: The dependence of the normalized lateness and the solution quality of the tasks on the workload when the bisection control is enabled and measured processing time is used.

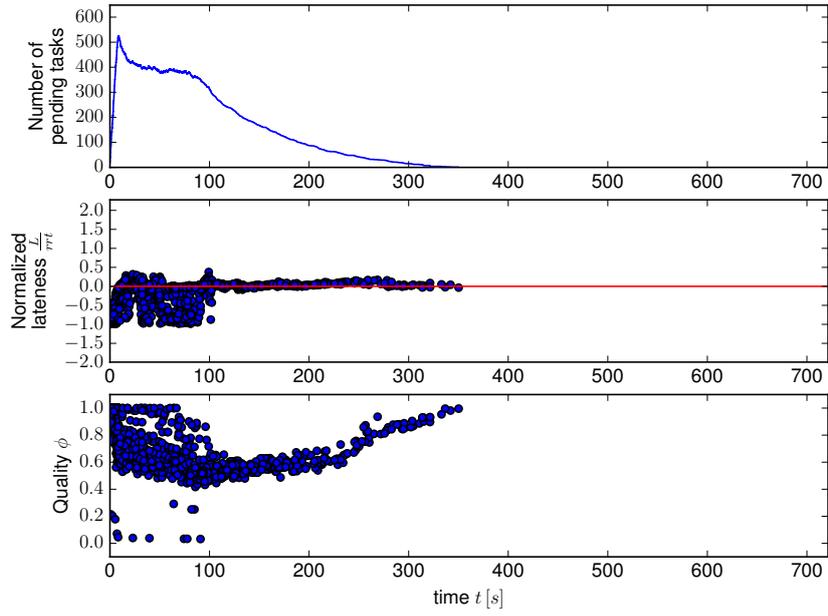


Figure 10: The dependence of the normalized lateness and the solution quality of the tasks on the workload when the bisection control is enabled and full estimation is used.

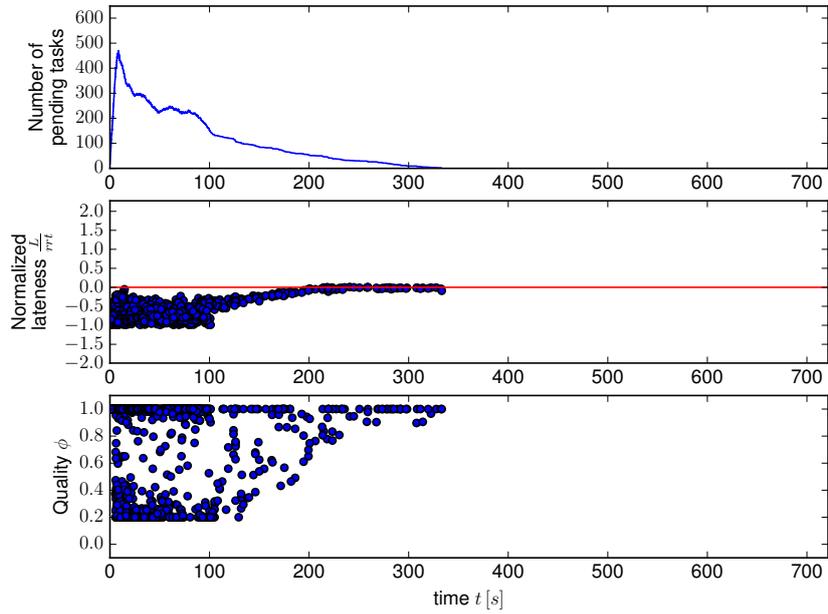


Figure 11: The dependence of the normalized lateness and the solution quality of the tasks on the workload when the individual control is enabled and measured processing time is used.

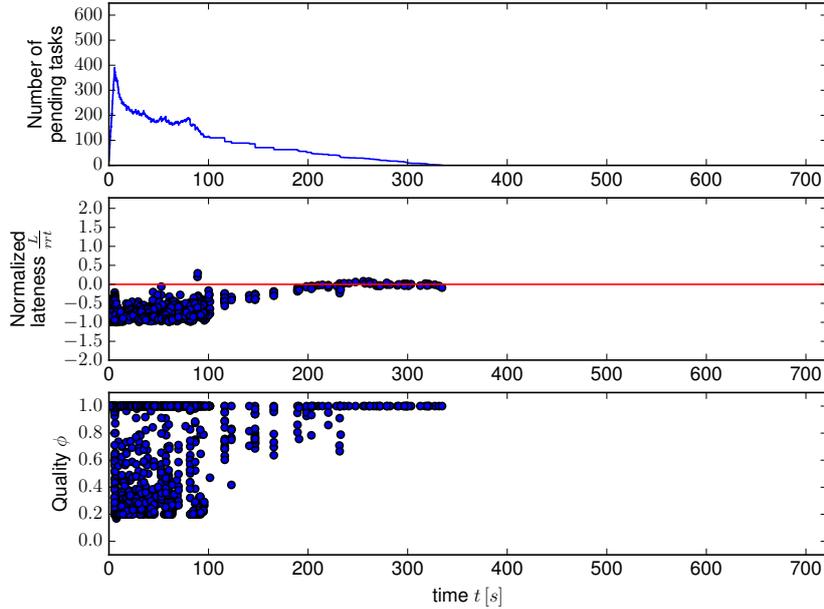


Figure 12: The dependence of the normalized lateness and the solution quality of the tasks on the workload when the naive control is enabled and measured processing time is used.

6.2.3. Overall comparison

Figures 13 and 14 show the spread of the normalized lateness and the quality of the solutions, respectively. Observe that the maximum normalized lateness for the bisection control, when the full estimation is used, is less than 0.38 and in 75% of the cases it is less than 0.051. Although these values are slightly higher than 0, it is much better than the maximum normalized lateness of the *Max quality*. However, the accuracy of the estimation affects the spread of the quality. The minimum solution quality for the bisection control when the full estimation is used is considerably lower than when the scheduling system is using the measured processing time. However, in most cases (i.e. 75%), the solution quality is sufficiently high.

For the individual control with the full estimation, the vast majority of the tasks have a lateness of less than 0.03 which is a great result considering that this result was not obtained at the cost of significantly decreasing the quality of the solutions. The quality of the solutions is also affected by the estimation but, as with the bisection control, the difference is small in comparison with the situation when the measured processing time is used. In comparison with the naive control, the individual control has much higher median of the solution quality. In the case of the full estimation, the difference in the median of the solution quality is almost 0.3. We note that even though the normalized lateness of the naive control is smaller compared to the individual control, the difference is not relevant because the normalized lateness of both control methods is negative - we aim for non-positive average normalized lateness.

The reason why the quality for the *Min quality* with the measured processing time has a large nonzero spread is because the smallest time unit is set to 1 ms which influences the instances with a small processing time. For example, consider a task instance with a linear processing time function and with the maximum normalized processing time of 100 ms. If the requested quality is set to 0.2, the normalized processing time equals to 20 ms. If the scheduling system decides to assign that task to a resource j with a speed of $s^{(j)} = 100$, it computes that the estimated processing time of that task is

$$\left\lceil \frac{p(\psi, 0.2)}{s^{(j)}} \right\rceil = \left\lceil \frac{20}{100} \right\rceil = \lceil 0.2 \rceil = 1, \quad (17)$$

i.e. resource j should process the task for 1 ms. After converting that processing time to the normalized processing time, i.e. 100 ms, it is obvious that the quality of the solution is 1 even though the requested quality is set to 0.2.

The overall comparison of various quality control algorithms and estimation methods is shown in Figure 15. The methods are compared by plotting the average quality of the solutions and the average normalized lateness. It can be seen that the full estimation does not significantly influence the average values, i.e. the scheduling system is robust against small errors in the estimation. When the bisection control is used, the average quality of the solutions for the full estimation achieves the highest quality, whereas the linear estimation is the worst. The full estimation is only slightly worse than using the measured processing time. When the individual control is used, the difference between the full and linear estimation is small because the individual control, according to Equation 12, uses only the estimation of the maximum normalized processing time. The difference arises from the fact that the scheduling policy uses the linear and full estimation in their original form. The result of the random control is not surprising because the mean of the uniform distribution $\mathcal{U}(\underline{\phi}, 1) = \mathcal{U}(0.2, 1)$ is $\mu = 0.6$.

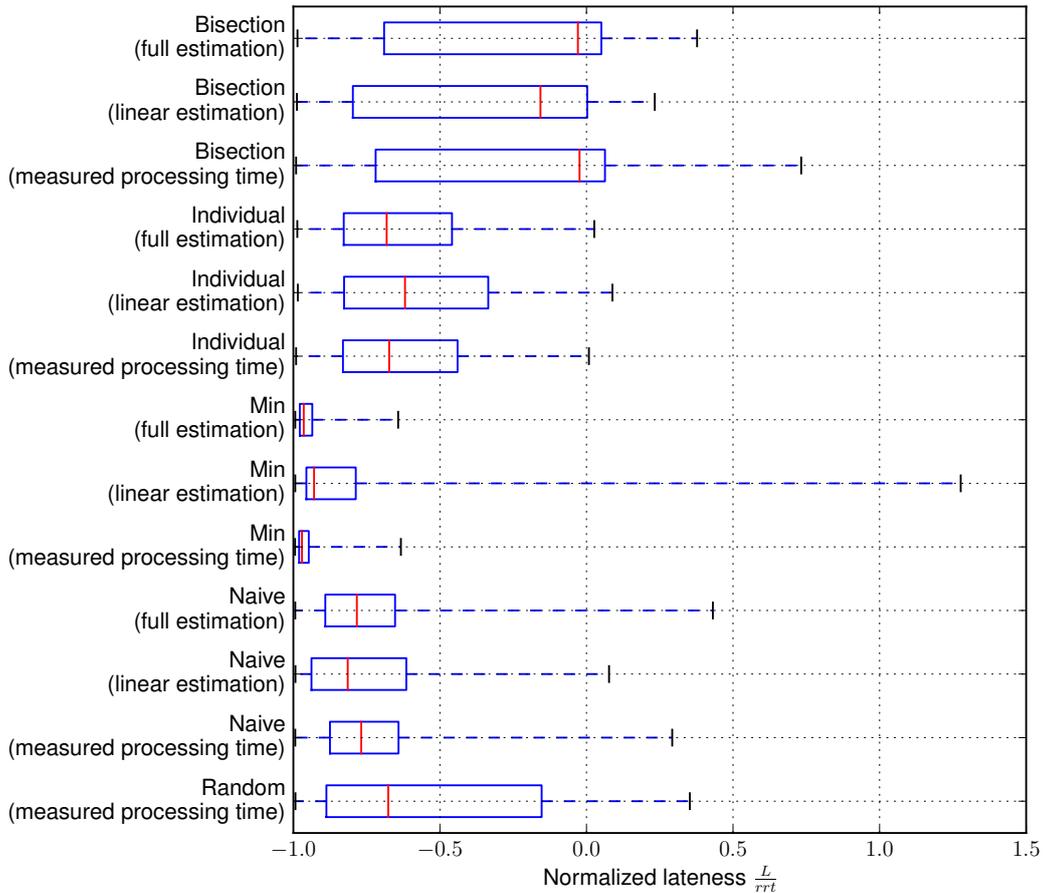


Figure 13: Comparison of various quality control and estimation methods by normalized lateness. The edges of the boxes represent the lower and upper quartile, the red vertical line inside each box is median and the whiskers extend from the minimum to maximum value.

Figure 15 also shows that the total solution quality is approximately the same for the individual and bisection control. The difference of these methods is how the quality is distributed among the tasks. Whereas the bisection control distributes the quality uniformly among the pending tasks, the individual control can compress the processing time of those tasks that have a large contribution to the average normalized lateness. The effect of this is that more solutions have a quality of 1 at the cost of having a few solutions of minimum quality. If the minimum quality were set to 0, the individual control would compress some tasks completely

and, therefore, the individual control would behave similarly as an admission control. Such behavior may not be desirable in some applications and, therefore we cannot declare that the individual control is better than the bisection control even though the numeric results are in favor of the individual control.

To compare the naive and the individual control, we note that the difference in the average solution quality between the corresponding estimation methods is at least 10% in favor of the individual control. Since the objective functions on the application layer (i.e. the objective functions of the task instances) relate to the quality of the solutions, such difference can be significant, e.g. consider an objective function denoting the saved money achieved by a solution. Moreover, in other scenarios the naive control may perform much worse, e.g. consider a situation when short task 1 and two much longer $\{2, 3\}$ arrive in the system at the same time. Assume that the scheduler creates schedule $(1, 2, 3)$ on one resource and the last task 3 is missing its requested due date. The naive control decreases the quality of the short task 1 even though it has a negligible effect on the completion time of the longer tasks and, therefore, the requested due date of task 3 is still being missed. After the solution of task 1 is received, the quality is recomputed. The naive control decreases the quality of task 2, which considerably affects the completion time of task 3 and the requested due date of task 3 is satisfied. On the other hand, the individual control detects that the congestion is caused by task 2 leaving the quality of task 1 intact.

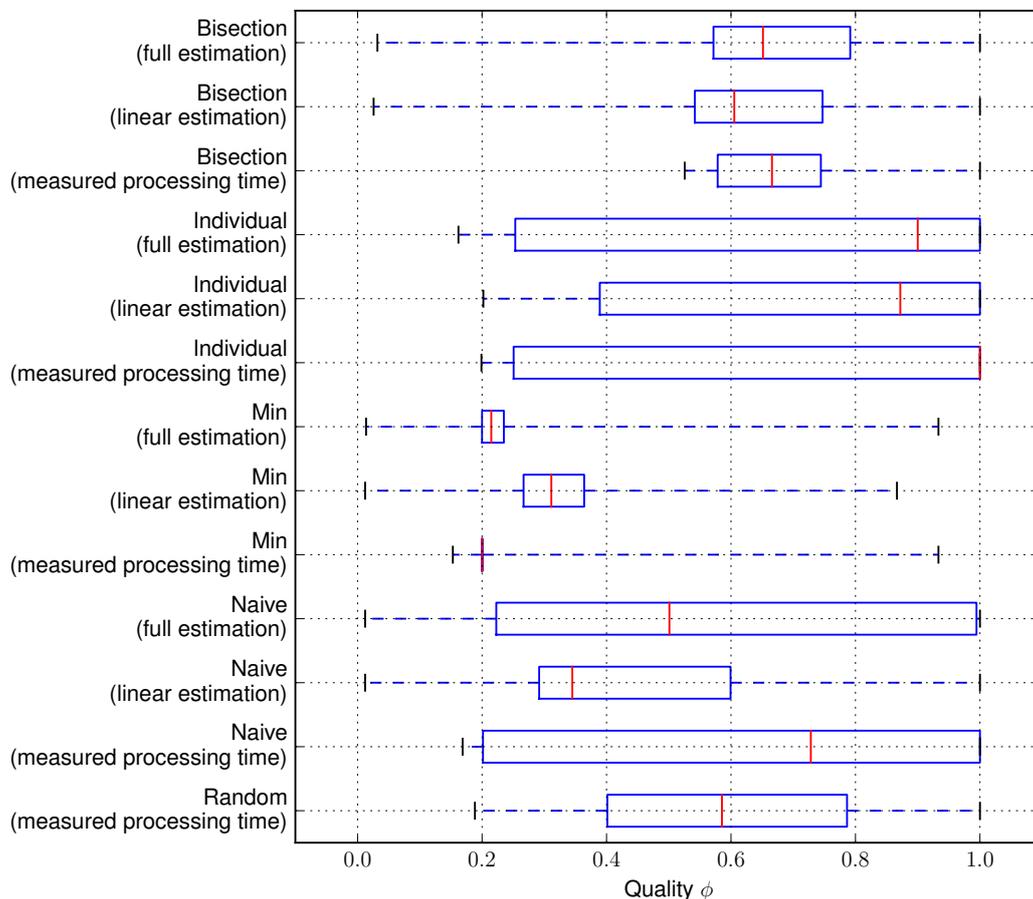


Figure 14: Comparison of various quality control and estimation methods by solution quality. The edges of the boxes represent the lower and upper quartile, the red vertical line inside each box is median and the whiskers extend from the minimum to maximum value.

The question may arise about the low normalized lateness of the individual control: could be the quality of solution increased so that the normalized lateness is closer to zero? Recall that the average normalized

lateness in the individual control is computed per resource. Consider an example with two resources $\{1, 2\}$ of speed 1 and three tasks $\{1, 2, 3\}$ with parameters from Table 2. If the average normalized lateness is computed over all resources for these tasks, then it is equal to -0.1296 . However, if the average normalized lateness is computed for each resource independently, then it is equal to 0.056 for resource 1 and for resource 2 it is equal to -0.5 . Since all tasks are running to the maximum quality and the overall average normalized lateness is less than 0, the bisection control would do nothing whereas the individual control would compress the tasks on resource 2 so that the average normalized lateness is closer to zero. By compressing the tasks, the solutions are received earlier and, therefore, the average normalized lateness of all the tasks in the experiments for the individual control is less than for the bisection control. The average quality of both control algorithm is, in the end, similar because the individual control is able to compress a single bottleneck task without affecting the rest of the tasks.

| i | $p(\psi^{(i)}, 1)$ | $rrt^{(i)}$ | $a^{(i)}$ | $r_S^{(i)}$ | $st_S^{(i)}$ | $ct(i, r_S^{(i)}, st_S^{(i)}, 1)$ |
|-----|--------------------|-------------|-----------|-------------|--------------|-----------------------------------|
| 1 | 10 | 10 | 0 | 1 | 0 | 10 |
| 2 | 90 | 90 | 0 | 1 | 10 | 100 |
| 3 | 10 | 20 | 0 | 2 | 0 | 10 |

Table 2: Parameters of the tasks for example explaining why the average normalized lateness of the individual control is less than 0.

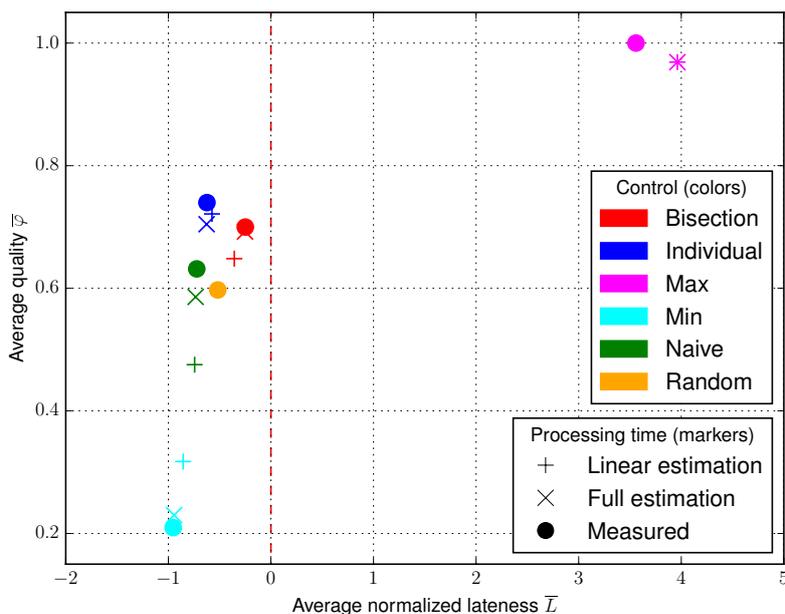


Figure 15: Comparison of various quality control and estimation methods by the average quality of the solutions and the average normalized lateness. The markers for random control are overlapping.

6.2.4. Performance of the algorithms

The measured dependence between the number of pending tasks and the running time of the proposed control algorithms is presented in Figure 16. The values for each number of pending tasks are obtained by choosing a maximum running time among all estimation methods. The results of all methods show that the running time of the quality computation is quite low even for a high number of pending tasks and, therefore, negligible for the whole system. Similarly, Figure 17 shows the dependency between the number of pending tasks and the running time of the scheduling policy. Again, the running time is negligible.

The reason for such small running times of the individual control algorithm is that the case when all pending tasks are assigned to one resource is not common in typical problem instances. The number of pending tasks on each resource are rather balanced which leads to a lower running time.

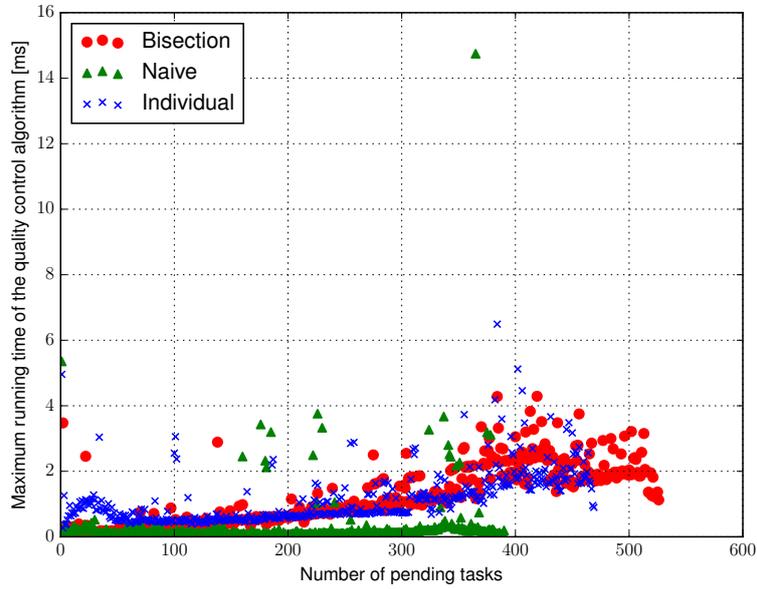


Figure 16: Comparison of the control algorithms by maximum running time. For each number of pending tasks there is, at most, one point (this is not obvious due to the density of the points).

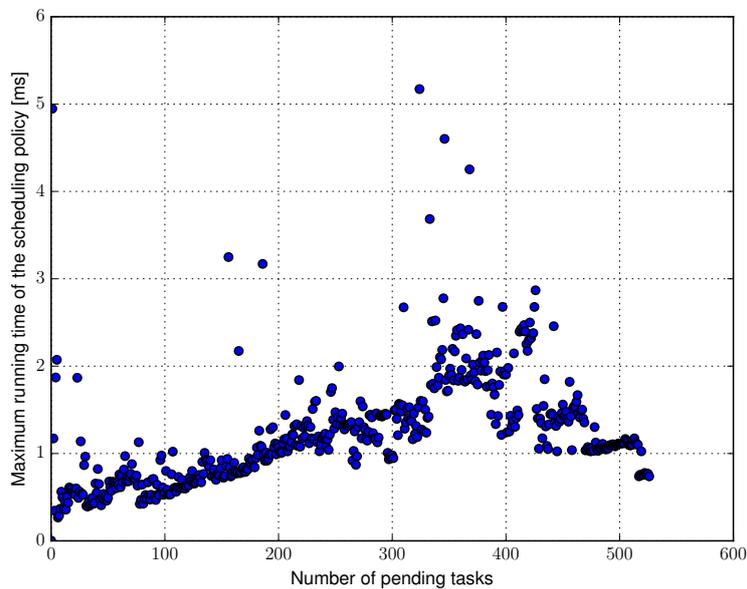


Figure 17: Maximum running time of the scheduling policy (since all control algorithms are using the same scheduling policy, the points are not distinguished by the control algorithms). For each number of pending tasks there is, at most, one point (this is not obvious due to the density of the points).

7. Conclusion

In this work, a problem of scheduling tasks in a heterogeneous environment is considered, i.e. the scheduling system has to determine the assignments of the tasks to the available computational resources. The goal of the scheduling system is to handle overload situations in which many computationally intensive tasks arrive in the system. In such a situation, the response time of the server increases considerably and this leads to user dissatisfaction.

The first characteristic of the tasks is that they are solved by anytime algorithms, i.e. the quality of the solutions depends on the processing time of the tasks. The second characteristic is that the relationship between the solution quality and the processing time of the tasks is not known *a priori*. Therefore, to make intelligent decisions the scheduling system has to estimate it.

Since the problem in its entirety is not already addressed in the existing literature, we proposed a modular system architecture, two efficient quality control algorithms and a procedure for estimating the processing time functions of the tasks. Both quality control algorithms exploit the anytime property, i.e. when the overload is detected, the algorithms decrease the requested quality of the solutions, thus, decreasing the response time of the server. The algorithms differ in how the requested quality is controlled; the *bisection control* sets one global quality for all tasks, whereas the *individual control* controls the quality of each task independently. The algorithmic complexities of the bisection and individual control algorithms are $\mathcal{O}(\text{maxIters} \cdot (m + n^{(*)}))$ and $\mathcal{O}(n^{(*)} \log n^{(*)})$, respectively (m is a number of resources and $n^{(*)}$ is a number of tasks in the system). Thanks to the low algorithmic complexity, the system can be used in online environments and is able to handle even hundreds of tasks.

To estimate the relationship between the solution quality and the processing time we introduced a procedure based on regression analysis. The measured relationships from the previously completed tasks are approximated by a piecewise linear function with ten segments whose endpoints are used to train the regression methods. The regression methods then estimate the processing time functions of the incoming tasks.

The design of the system is evaluated on a real client-server application from a domain of *personnel rostering*. The experiments show a huge decrease in the response time when either of the proposed quality control algorithms is used. The proposed algorithms also outperformed a simple control approach which always stop a task that was running for the longest time. The experiments also show that both control algorithms are robust against up to 20% error in the estimation, i.e. the results in the average solution quality and the average normalized lateness are not significantly different if the estimation is used. The positive results verify the validity of our approach.

Although the scheduling and quality control algorithms can be used in any domain where the anytime tasks are scheduled, future research could focus on generalizing the estimation methods to domains other than personnel rostering. Currently, we require that the relevant features of the task instances are identified manually so that the estimation can be performed. However, the scheduling system could perform automatic *feature learning* based on the description of the task instances. Moreover, the estimation error could be further decreased by using different regression methods such as Deep Learning.

Future research could also focus on extending the individual control algorithm with non-linear processing time functions. We propose two ideas how to extend this algorithm. The first method splits each task along the linear segments of the piecewise linear function (see Section 5). The original task is replaced by the newly created tasks. If each task consists of ten segments, then the complexity is $\mathcal{O}(10n^{(*)} \log(10n^{(*)}))$, however, to ensure the correct behavior of the algorithm, the slopes of each segment of one task has to be non-decreasing. The second method does not split the tasks but uses only the last segments of the piecewise linear functions to compute the weights. The task with the highest weight is compressed only up to the starting endpoint of the last segment, then the last segment of that task is thrown away and the weight of that task is recomputed using the one before the last segment. The advantage of this method is that there is no assumption of the shape of the piecewise linear functions as with the previous method. However the complexity of this method is $\mathcal{O}(n^{(*)} \cdot n^{(*)})$.

Acknowledgment

This work was supported by the ARTEMIS initiative funded by the European Commission under the project DEMANES 295372.

References

References

- [1] Amazon.com, How elastic load balancing works: Request routing. From Amazon Web Services documentation, <http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/how-elb-works.html#request-routing>, accessed August 3, 2015.
- [2] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, R. F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, *Journal of Parallel and Distributed Computing* 61 (6) (2001) 810 – 837.
- [3] P. Brucker, *Scheduling Algorithms*, 5th ed., Springer-Verlag, Berlin, 2007.
- [4] Z. Bäuml, P. Šücha, Z. Hanzálek, A multistage approach for an employee timetabling problem with a high diversity of shifts as a solution for a strongly varying workforce demand, *Computers & Operations Research* 49 (2014) 117 – 129.
- [5] F. Y. L. Chin, S. P. Y. Fung, Online scheduling with partial job values: Does timesharing or randomization help?, *Algorithmica* 37 (3) (2003) 149–164.
- [6] J. K. Dey, J. Kurose, D. Towsley, On-line scheduling policies for a class of IRIS (increasing reward with increasing service) real-time tasks, *IEEE Transactions on Computers* 45 (7) (1996) 802–813.
- [7] F. Dong, S. G. Akl, Technical report no. 2006-504 scheduling algorithms for grid computing: State of the art and open problems (2006).
- [8] A. Ernst, H. Jiang, M. Krishnamoorthy, D. Sier, Staff scheduling and rostering: A review of applications, methods and models, *European Journal of Operational Research* 153 (1) (2004) 3 – 27.
- [9] I. Foster, Y. Zhao, I. Raicu, S. Lu, Cloud computing and grid computing 360-degree compared, in: *Grid Computing Environments Workshop, 2008. GCE '08, 2008*, pp. 1–10.
- [10] K. Gilly, C. Juiz, N. Thomas, R. Puigjaner, Adaptive admission control algorithm in a QoS-aware web system, *Information Sciences* 199 (0) (2012) 58–77.
- [11] Y. He, S. Elnikety, Scheduling for data center interactive services, in: *Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on, 2011*, pp. 1170–1181.
- [12] J. L. Hellerstein, Y. Diao, S. Parekh, D. M. Tilbury, *Feedback Control of Computing Systems*, Wiley-IEEE Press, New Jersey, 2004.
- [13] H. Hoogeveen, Multicriteria scheduling, *European Journal of Operational Research* 167 (3) (2005) 592–623.
- [14] J. Hoogeveen, A. Vestjens, Optimal on-line algorithms for single-machine scheduling, in: W. H. Cunningham, S. T. McCormick, M. Queyranne (eds.), *Integer Programming and Combinatorial Optimization*, vol. 1084 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 1996, pp. 404–414.
- [15] F. Hutter, L. Xu, H. H. Hoos, K. Leyton-Brown, Algorithm runtime prediction: Methods & evaluation, *Artificial Intelligence* 206 (0) (2014) 79–111.
- [16] O. H. Ibarra, C. E. Kim, Heuristic algorithms for scheduling independent tasks on nonidentical processors, *J. ACM* 24 (2) (1977) 280–289.
- [17] M. Iverson, F. Ozguner, L. Potter, Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment, in: *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth, 1999*, pp. 99–111.
- [18] M. Karlsson, M. Covell, Dynamic black-box performance model estimation for self-tuning regulators, in: *Proceedings of the Second International Conference on Automatic Computing*, IEEE Computer Society, Washington, DC, USA, 2005, pp. 172–182.
- [19] D. Klusáček, H. Rudová, Multi-resource aware fairsharing for heterogeneous systems, in: *Job Scheduling Strategies for Parallel Processing*, 1st ed., Springer, 2014.
- [20] J. W. S. Liu, K.-J. Lin, W.-K. Shih, A. C.-s. Yu, J.-Y. Chung, W. Zhao, Algorithms for scheduling imprecise computations, *Computer* 24 (5) (1991) 58–68.
- [21] C. Lu, J. A. Stankovic, G. Tao, S. H. Son, Design and evaluation of a feedback control EDF scheduling algorithm, in: *Proceedings of the 20th IEEE Real-Time Systems Symposium*, IEEE Computer Society, Washington, DC, USA, 1999, pp. 56–67.
- [22] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, R. Freund, Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems, in: *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth, 1999*, pp. 30–44.
- [23] T. Messelis, P. De Causmaecker, G. Vanden Berghe, Algorithm performance prediction for nurse rostering, in: *Proceedings of the 6th Multidisciplinary International Scheduling Conference: Theory and Applications, 2013*, pp. 21–38.
- [24] A. Mittal, G. Manimaran, C. Murthy, Integrated dynamic scheduling of hard and qos degradable real-time tasks in multiprocessor systems, in: *In Proceedings on Fifth International Conference Real-Time Computing Systems and Applications*, IEEE Computer Society, Washington, DC, USA, 1998, pp. 127–136.

- [25] A. J. Page, T. M. Keane, T. J. Naughton, Scheduling in a dynamic heterogeneous distributed system using estimation error, *Journal of Parallel and Distributed Computing* 68 (11) (2008) 1452–1462.
- [26] J. Sgall, On-line scheduling, in: A. Fiat, G. J. Woeginger (eds.), *Online Algorithms: The State of the Art*, Springer-Verlag, Berlin, Heidelberg, 1998, pp. 196–231.
- [27] D. Shabtay, G. Steiner, A survey of scheduling with controllable processing times, *Discrete Applied Mathematics* 155 (13) (2007) 1643 – 1666.
- [28] C.-T. Tseng, C.-J. Liao, K.-L. Huang, Minimizing total tardiness on a single machine with controllable processing times, *Computers & Operations Research* 36 (6) (2009) 1852–1858.
- [29] M. Welsh, D. Culler, Adaptive overload control for busy internet servers, in: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems*, USENIX Association, Berkeley, CA, USA, 2003, pp. 43–57.
- [30] F. Xhafa, A. Abraham, Computational models and heuristic methods for grid scheduling problems, *Future Generation Computer Systems* 26 (4) (2010) 608–621.

Appendix A. Total complexity of computing Algorithm 4 for all resources

Lemma 1. *Let $x_1, x_2, \dots, x_n \in \mathbb{N}_{>0}, n \geq 2$. Then it holds that*

$$\left(\sum_{i=1}^n x_i \right) \log \left(\sum_{i=1}^n x_i \right) > \sum_{i=1}^n x_i \log x_i. \quad (\text{A.1})$$

PROOF. We start by noting that

$$\frac{\sum_{j=1}^n x_j}{x_i} > 1, \quad i = 1, \dots, n \quad (\text{A.2})$$

We use this result to prove Lemma 1

$$\begin{aligned} & \left(\sum_{i=1}^n x_i \right) \log \left(\sum_{i=1}^n x_i \right) > \sum_{i=1}^n x_i \log x_i \\ & \log \left(\sum_{i=1}^n x_i \right)^{\left(\sum_{i=1}^n x_i \right)} > \log \left(\prod_{i=1}^n x_i^{x_i} \right) \\ & \left(\sum_{i=1}^n x_i \right)^{\left(\sum_{i=1}^n x_i \right)} > \prod_{i=1}^n x_i^{x_i} \\ & \prod_{i=1}^n \left(\sum_{j=1}^n x_j \right)^{x_i} > \prod_{i=1}^n x_i^{x_i} \\ & \prod_{i=1}^n \left(\frac{\sum_{j=1}^n x_j}{x_i} \right)^{x_i} > 1 \end{aligned} \quad (\text{A.3})$$

□

Theorem 1. *The total worst case complexity of computing Algorithm 4 for all resources is $\mathcal{O}(n^{(*)} \log n^{(*)})$.*

PROOF. In Section 4.2.2, the complexity of computing Algorithm 4 for one resource has been discussed already. To prove Theorem 1, it has to be shown that the worst case is when all pending tasks are assigned to only one resource.

Assume that not all pending tasks are assigned to one resource, i.e.

$$\exists j', j'' \in M : j' \neq j'' \wedge n^{(j')} \geq 1 \wedge n^{(j'')} \geq 1. \quad (\text{A.4})$$

This also implies that there are at least two pending tasks, i.e. $n^{(*)} \geq 2$. The total complexity of computing Algorithm 4 for all resources is

$$\sum_{j \in M : n^{(j)} \geq 1} n^{(j)} \log n^{(j)}. \quad (\text{A.5})$$

From Lemma 1 it holds that

$$\left(\sum_{j \in M: n^{(j)} \geq 1} n^{(j)} \right) \log \left(\sum_{j \in M: n^{(j)} \geq 1} n^{(j)} \right) > \sum_{j \in M: n^{(j)} \geq 1} n^{(j)} \log n^{(j)} \quad (\text{A.6})$$

Because all tasks have to be assigned to some resource, it holds that

$$n^{(*)} = \sum_{j \in M: n^{(j)} \geq 1} n^{(j)}. \quad (\text{A.7})$$

By substituting $n^{(*)}$ into inequality (A.6) we get

$$n^{(*)} \log n^{(*)} > \sum_{j \in M: n^{(j)} \geq 1} n^{(j)} \log n^{(j)}. \quad (\text{A.8})$$

This inequality proves that the worst case is when all pending tasks are assigned to only one resource.

□