BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 68, No. 4, 2020
DOI: 10.24425/bpasts.2020.134187

Online scheduling for a Testing-as-a-Service system

J. RUDY* and C. SMUTNICKI

Wrocław University of Science and Technology, Department of Computer Engineering,
Wybrzeże Wyspiańskiego 27, 50-370, Wrocław, Poland

Abstract. The problem of performing software tests using Testing-as-a-Service cloud environment is considered and formulated as an~online cluster scheduling on parallel machines with total flowtime criterion. A mathematical model is proposed. Several properties of the problem, including solution feasibility and connection to the classic scheduling on parallel machines are discussed. A family of algorithms based on a new priority rule called the Smallest Remaining Load (SRL) is proposed. We prove that algorithms from that family are not competitive relative to each other. Computer experiment using real-life data indicated that the SRL algorithm using the longest job sub-strategy is the best in performance. This algorithm is then compared with the Simulated Annealing metaheuristic. Results indicate that the metaheuristic rarely outperforms the SRL algorithm, obtaining worse results most of the time, which is counter-intuitive for a metaheuristic. Finally, we test the accuracy of prediction of processing times of jobs. The results indicate high (91.4%) accuracy for predicting processing times of test cases and even higher (98.7%) for prediction of remaining load of test suites. Results also show that schedules obtained through prediction are stable (coefficient of variation is 0.2–3.7%) and do not affect most of the algorithms (around 1% difference in flowtime), proving the considered problem is semi-clairvoyant. For the Largest Remaining Load rule, the predicted values tend to perform better than the actual values. The use of predicted values affects the SRL algorithm the most (up to 15% flowtime increase), but it still outperforms other algorithms.

Key words: discrete optimization, online scheduling, cloud computing, Testing-as-a-Service.

Notation

Below is the summary of notation used throughout the paper:

n, m	– number of jobs, number of machines
q	– number of families of jobs
$\mathscr{J}$, $\mathscr{M}$	– the set of jobs, the set of machines
$\mathscr{B}$	– the set of families of jobs
$\mathscr{B}^u$	– the u-th family of jobs
r_j	– ready (arrival) time of job j
b_j	– the family to which job j belongs
p_j	– processing time of job j
t_j	– type of job j
z_j	– file transfer time for job j
s_{jk}	– setup time from job j to job k
S_j, C_j	– starting and completion time of job j
A_j	– the machine to which job j was assigned
S, C	– vector of jobs starting and completion times
A	– the vector of job-to-machine assignments
(S, A)	– solution (schedule)
π_i	– vector of jobs to be processed on machine i
$\pi_i(j)$	– the j-th job to be processed on machine i
π	– the order of processing of jobs on machines
$F_j(\pi)$, $F_j(S, A)$	– flowtime of job j in processing order π or schedule (S, A)
$F^u(\pi)$, $F^u(S, A)$	– flowtime of family of jobs u in processing order π or schedule (S, A)
$\hat{p}_j$	– predicted value of p_j
H_t, $H_t(x)$	– vector of processing times of past executions of jobs with type t and x-th most recent execution time in H_t

1. Introduction

The process of software testing is an essential stage in software development process, consuming around 50 to 60% of the total software development costs [1]. Due to the rapid growth of the IT industry and the size of software, the testing process is becoming more complex. At the same time, cloud computing is becoming a rapidly growing paradigm in many areas, including scheduling [2]. As a result, in recent years cloud environments have been employed to facilitate the process of large-scale software testing, leading to the birth of the so-called *Testing-as-a-Service* (or TaaS) cloud computing model [3].

Software tests are often composed of a number of *test suites*, each test suite testing a single functionality of the developed software. Test suites are in turn sets (clusters) of *test cases*, each meant to test the functionality in a different way. Test suites are run on a set of computers (*i.e.* in a cloud environment). Test cases from a given test suite can be executed in any order. Moreover, in some fields, like telecommunications and simulations, the time to perform a single test case can be long, taking tens of seconds on average.

One type of testing that employs test suites is continuous integration (CI) testing. In such a setting the testing process is performed several times every day, yielding a large number of test cases that have to be run in short timespan. One character-

*e-mail: jaroslaw.rudy@pwr.edu.pl

Manuscript submitted 2019-10-17, revised 2020-03-19, initially accepted for publication 2020-04-15, published in August 2020

istic of such a system is the time that passes from when the test suites were ordered for testing to when all test cases of that test suite were completed. This value is called "response time" or flowtime. From a practical point of view, it is thus important to schedule the execution of all test cases on available computers to optimize the flowtime of all test suites. This allows the users to get their response earlier and, in turn, increase productivity.

The idea of a TaaS can be traced back to a paper by Yu *et al.* [3] where a TaaS framework was proposed. Since then several papers concerning such systems have been published. For example, a paper by Yu *et al.* [4] described the architecture of a TaaS system and its complexity. The paper discussed practical issues concerning setting and managing a TaaS and proposed methods of clustering tasks and simple priority-based algorithms to optimize user waiting time. A paper by Sathe and Kulkarni [5] discussed cost-effectiveness and importance of TaaS systems. A paper by Lampe and Rudy analyzed a real-life TaaS system to construct probabilistic models of its workload with good accuracy [6]. Finally, Lee *et al.* considered a TaaSlike system for load and cross-browser testing for website development. The authors proposed a system for delivering heterogeneous web testing tools [7]. The results indicate that the system reduces the effort required for testing compared to conventional methods. Further information about challenges and applications of TaaS systems can be found in [8].

The problem of optimizing testing schedule for TaaS and similar systems was also a topic of several papers. Binder proposed an approach to optimizing the order of test cases inside a test suite through integer programming [9]. Some authors considered multi-objective approach to scheduling software tests as well, considering criteria like average response time, machine utility and test coverage [10, 11]. A paper by Lampe proposed a fuzzy approach, modeling test cases processing times with fuzzy numbers [12]. Alie *et al.* proposed a TaaS system for mobile applications, resulting in reduced testing time and improved resource utilization [13].

Optimization of software testing schedule can be viewed as a specific case of online scheduling for various computing systems like clouds, clusters and grids. A large number of papers on this topic have been published over the last several years. Gog *et al.* [14] formalized the problem as a min-cost maxflow optimization over a graph. The results using Google trace data showed a considerable improvement in placement latency and response time. A paper by Rasley *et al.* [15] discussed advantages and disadvantages of both distributed and centralized schedulers. Delgado *et al.* proposed Hawk scheduler, using both centralized and distributed scheduler for long and short jobs, respectively [16]. Ousterhout *et al.* considered a system with millions of very short jobs and obtained near-optimal schedules. Lastly, Lee *et al.* [17] presented a survey on the online problem of minimizing makespan with machines eligibility.

One of the practical issues in online scheduling is the estimation of job processing times which are usually unknown. Delgado *et al.* [18] proposed a hybrid scheduler that was tested under Google trace dataset and showed high resistance to misestimation of task duration. A number of papers [15, 19] analyze publicly available trace datasets from Google or Microsoft and conclude there is a lot of repetitiveness of tasks (over 60% of tasks are recurring). Moreover, a simple analysis of data of a real-life TaaS system revealed that on average tasks were repeated hundreds of times [10]. This indicates that estimation of processing times is possible for TaaS systems.

A number of papers deal specifically with algorithms for online scheduling. One of the most effective algorithms for minimizing total flowtime of jobs is the Shortest Remaining Processing Time (SRPT) algorithm, for which strong bounds were proven [20]. The SRPT algorithm was thus used in several papers [6, 15, 18]. Research of lower and upper bounds and competitive ratio is common with online algorithms. For example, Dósa *et al.* showed tight lower bounds for twomachine semi-online problem with machine speeds [21]. While fast and simple rule-based algorithms are common, several approaches to online scheduling using metaheuristic methods appeared as well. Iordache *et al.* combined genetic algorithms with lookup services to provide a fault-tolerant, scalable and efficient solution for optimizing task assignment for a decentralized grid scheduling [22]. Similarly, Xhafa *et al.* employed cellular memetic algorithms to schedule jobs for a grid system [23]. The results obtained using known benchmarks showed very high quality. In general, metaheuristics are often viewed as viable methods for many optimization problems ranging from various assignment and distribution problems (see for example paper on rich portfolio optimization by Kizys *et al.* [24]) to job shop scheduling (see for example paper on its multi-criteria by Rudy and ▯elazny [25]). Thus, intuition suggests this approach should be also viable for the problem considered in this paper. More information on the use of metaheuristics for scheduling in cloud computing and similar systems can be found in [26].

In this paper our purpose is to model the problem of reducing the response time of the software testing process as a certain online scheduling problem with the total flowtime goal function. The problem is online because the scheduling algorithm becomes aware of a test suite only when it arrives. Moreover, the time it takes to execute a given test case is not known for certain. However, a single test case is usually executed many times throughout the development process. Even though the content of the test case or the tested code may change between executions, as one of our contributions, we will show that this still allows us to predict the execution time of test cases with high accuracy.

From the above literature overview we conclude that while there exist several approaches to TaaS-like systems, they often focus on practical implementations, taking into account too many factors, making our understanding of the core problem (scheduling of test cases and test suites) largely incomplete. Furthermore, there exist many general theoretical approaches (including lower and upper bounds) to online scheduling for cluster and distribute systems, but we feel those approaches, while sometimes applicable, are too general and cannot take full advantage of the specificity of the considered problem.

Our purpose is to focus on the single aspect of the problem: scheduling. This will allow us to use the theory of scheduling to formulate properties for the problem and some algorithms, increasing our understanding of the problem. Finally, we realize the problem is case-specific, as each software project is different and may require different parameters or processing times prediction strategies. Thus, we base our approach on a real-life TaaS system and use real-life data about test suites and test cases encountered during operation of that system. Therefore, our approach is partially a case study, though we also feel that it could be applied to similar TaaS systems or even more general discrete optimization problems (more on that in Section 8.)

The remainder of this paper is organized as follows. Section 2 contains problem formulation. Sections 3 and 4 discuss properties of the problem and several online algorithms, respectively. Section 5 contains online algorithms comparison based on data from a real-life TaaS system. Section 6 discusses metaheuristic algorithm approach, including a computer experiment. In Section 7 we show the results of predicting processing times of test cases and test suites. Section 8 contains discussion of several aspects and application of our approach and results. Finally, Section 9 contains the conclusions.

2. Problem formulation

In this section we formulate the optimization problem under consideration. All values are integers unless otherwise stated.

Let $\mathscr{J} = \{1, 2, \ldots, n\}$ be a set of n jobs. Each job j has ready (arrival) time, processing time and type denoted r_j, p_j and t_j, respectively. The set $\mathscr{J}$ is partitioned into q sets $\mathscr{B}^1, \mathscr{B}^2, \ldots, \mathscr{B}^q$, called families of jobs, with $\mathscr{B}^u$ denoting the u-th family of jobs. For each job j let b_j denote to which family j belongs:

$$\forall\, j \in \mathscr{J} \;:\; b_j = u \iff j \in \mathscr{B}^u. \tag{1}$$

Next, let $\mathscr{M} = \{1, 2, \ldots, m\}$ be a set of m identical (parallel) machines, meaning that any machine can process any job. By s_{jk} we will denote the setup time a machine has to undergo before processing job k if the last job processed on that machine was job j. Also, let s_{0k} denote a special situation, when no job was processed on the given machine prior to the processing of job k. The setup time s_{jk} is defined as follows:

$$s_{jk} = \begin{cases} z_k & \text{if } j = 0 \vee b_j \neq b_k, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where z_j is a job-specific value called file transfer time.

Finally, we assume that jobs belonging to the same family of jobs have the same type and file transfer size:

$$\forall\, j,k \in \mathscr{J} \;:\; b_j = b_k \implies t_j = t_k\,, \tag{3}$$

$$\forall\, j,k \in \mathscr{J} \;:\; b_j = b_k \implies z_j = z_k\,. \tag{4}$$

Let us consider how the introduced elements match the concepts in a real-life software testing system. Families of jobs are meant to represent test suites. Jobs, in turn, are representing *executions* of test cases. The type of job is meant to identify the particular test case. If two jobs have the same type then they are different executions of the same test case. This represents a situation when the same test case is executed more than once (common in software testing). Also, two jobs of the same type might have different processing times, representing a situation when the test case was modified between executions. The set of machines represents the computers on which the test cases are executed. Finally, setup times represent the time needed to transfer the files necessary for a given test case to the target machine. However, all test cases in a given test suite need the same set of files. Thus, if $j \neq 0$ and $b_j = b_k$, then there is no need to copy any files and $s_{jk} = 0$. Otherwise, the setup time depends on how long it takes to transfer the files for that particular test suite, thus $s_{jk} = z_k$. To sum it up, in general the setup time depends on both j and k, but if those jobs are from the same family of jobs, then setup time depends only on k.

The task is to assign to each job its starting time and machine on which it should be processed to minimize a certain goal function. Let $S = (S_1, S_2, \ldots, S_n)$ and $A = (A_1, A_2, \ldots, A_n)$ be vectors of job starting times and job-to-machine assignments, respectively. Thus, job j will be processed on machine A_j starting at time S_j. The pair (S, A) will be called a schedule. For convenience we can also define vector of job completion times $C = (C_1, C_2, \ldots, C_n)$. Below we present an example instance and schedule for the considered problem.

Example 1. Let us consider a problem instance with 7 jobs $\mathscr{J} = \{1,2,3,4,5,6,7\}$, 4 machines $\mathscr{M} = \{1,2,3,4\}$ and 3 families of jobs: $\mathscr{B}^1 = \{1,2\}$, $\mathscr{B}^2 = \{3,4,5\}$, $\mathscr{B}^3 = \{6,7\}$. The ready times of jobs are: $r_1 = r_2 = 0$, $r_3 = r_4 = r_5 = 4$, $r_6 = r_7 = 9$. The processing times of jobs are $p_1 = p_6 = p_7 = 2$, $p_2 = p_3 = 6$, $p_4 = 7$, $p_5 = 8$. For simplicity let us ignore setup times and job types. For this instance a possible schedule (S, A) could be:

$$S = (0,0,4,4,4,9,10), \tag{5}$$

$$A = (1,2,1,3,4,2,1). \tag{6}$$

The schedule for this instance is presented in Fig. 1 with different colors denoting jobs from different families of jobs.

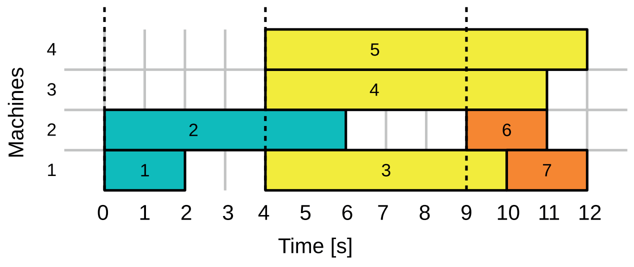

Fig. 1. Example schedule with 7 jobs, 4 machines and 3 families of jobs

Not all schedules are feasible. Schedule is feasible if the following constraints are met: (1) a job can be processed by at most one machine at a time, (2) a machine can process at most one job at a time, (3) job j needs to be processed for time p_j without interruption (implying that $C_j = S_j + p_j$), (4) processing of a job cannot start before its ready time, (5) setup time has to be performed before processing of each job. Let us notice that the schedule presented in Example 1 is feasible.

We want to consider only feasible schedules. To this end, let $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$ be a vector, where π_i is the (possibly empty) vector of jobs to be processed on machine i. The jobs in π_i will be processed in the order they appear in the vector, i.e. $\pi_i(j)$ is the j-th job to be processed on the i-th machine. Vector π must meet the following conditions:

$$\pi_i(j) \in \mathscr{J}, \tag{7}$$

$$i \neq l \vee j \neq k \implies \pi_i(j) \neq \pi_l(k), \tag{8}$$

$$\sum_{i=1}^{m} |\pi_i| = n. \tag{9}$$

This ensures that π contains only jobs and each job appears in π exactly once. Also, note that vectors π_i can have different lengths, thus π is not a matrix. π will be called a processing order and will serve as the decision variable.

The schedule (S,A) can be derived from the processing order π using the equations below:

$$S_{\pi_i(j)} = \max\{r_{\pi_i(j)}, C_{\pi_i(j-1)}\} + s_{\pi_i(j-1)\pi_i(j)}, \quad j = 1,2,\ldots,|\pi_i|, \; i \in \mathscr{M}, \tag{10}$$

$$C_{\pi_i(j)} = S_{\pi_i(j)} + p_{\pi_i(j)}, \quad j = 1,2,\ldots,|\pi_i|, \; i \in \mathscr{M}, \tag{11}$$

$$A_{\pi_i(j)} = i, \quad j = 1,2\ldots,|\pi_i|, \; i \in \mathscr{M}, \tag{12}$$

with additional starting conditions $\pi_i(0) = 0$ and $C_0 = 0$.

A possible processing order for schedule from Example 1 would be $\pi = (\pi_1, \pi_2, \pi_3, \pi_4)$, where $\pi_1 = (1,3,7)$, $\pi_2 = (2,6)$, $\pi_3 = (4)$ and $\pi_4 = (5)$. Further in the paper we will provide a proof that each π represents a single feasible schedule (S,A).

Let us now define the goal function to optimize. By $F_j(\pi)$ we denote the flowtime of job j according to some processing order π. $F_j(\pi)$ is defined as the difference between the completion and ready time of j:

$$F_j(\pi) = C_j - r_j. \tag{13}$$

Similarly, we can define $F_j(S,A) = F_j(\pi)$ if processing order π represents schedule (S,A).

Next, by $F^u(\pi)$ we denote flowtime of family of jobs $\mathscr{B}^u$, which is the maximum of flowtimes of jobs belonging to $\mathscr{B}^u$:

$$F^u(\pi) = \max_{j \in \mathscr{B}^u} F_j(\pi). \tag{14}$$

Similarly, we can define $F^u(S,A) = F^u(\pi)$.

Finally, let $\Sigma F(\pi)$ be a total flowtime according to π and defined as the sum of flowtimes over all families of jobs:

$$\Sigma F(\pi) = \sum_{u=1}^{q} F^u(\pi). \tag{15}$$

As before, $\Sigma F(\pi) = \Sigma F(S,A)$ if π represents (S,A).

The goal is to find such processing order π^* as to minimize the total flowtime:

$$\Sigma F(\pi^*) = \min_{\pi} \Sigma F(\pi). \tag{16}$$

Once again, $\Sigma F(S^*,A^*) = \Sigma F(\pi^*)$. The processing order π^* (schedule (S^*,A^*)) is called the optimal processing order (optimal schedule). In our Example 1 the total flowtime $\Sigma F(\pi) = F^1(\pi) + F^2(\pi) + F^3(\pi) = 6 + 8 + 3 = 17$. This is, in fact, the optimal schedule for this instance.

The above problem will be called *cluster scheduling problem* on parallel machines with ready times and setup times constraint and total flowtime goal function. We will denote this problem by $\mathrm{C}_m|r_j,s_{jk}|\Sigma F$ in the Graham's notation (m can be omitted if it is obvious). "Cluster" here means that jobs are grouped into clusters (families). Unlike operations in problems like the job shop problem, any job can be processed on any machine, there are no precedence constraints and many jobs of the same family can be processed at the same time (on different machines). Thus, it is not a typical multi-stage scheduling or scheduling on identical parallel machines.

Finally, we will add two additional constraints, reflecting the issues encountered in practice. First, the information about each job j is unavailable to solving algorithms until time r_j, making this problem online. Second, job processing times p_j are uncertain. Further in the paper we will show that values p_j can be predicted with high accuracy and little effect on the resulting total flowtime, making this problem semi-clairvoyant. The problem with those additional constraints will be denoted by $\mathrm{C}_m|online\text{-}time\text{-}sclv, r_j, s_{jk}|\Sigma F$.

3. Problem properties

In this section we will formulate several properties for the considered problem (or its variants). This will include properties concerning processing orders π and schedules (S,A) they represent. For now we assume that the processing times of jobs are known (*i.e.* the cluster scheduling problem is clairvoyant).

We start by defining another class of schedules. A feasible schedule (S,A) is called *semi-active* if there does not exist another feasible schedule (S',A') satisfying the conditions below:

1. Jobs in (S',A') are processed on the same machines as in (S,A), *i.e* $A' = A$.
2. Jobs assigned to machine i in schedule (S',A') are processed in the same order as in schedule (S,A).
3. No job in (S',A') is processed later than the corresponding job in (S,A), *i.e.* $\forall j \in \mathscr{J} : S'_j \leq S_j$.
4. At least one job in (S',A') is processed earlier than the corresponding job in (S,A), *i.e.* $\exists j \in \mathscr{J} : S'_j < S_j$.

We will now show that all non-semi-actives schedules can be ignored when solving the considered problem.

Property 1. If (S,A) is a feasible schedule that is not semi-active, then there exists a semi-active schedule (S',A') such that:

$$\Sigma F(S',A') \leq \Sigma f(S,A). \tag{17}$$

Proof. Since (S,A) is feasible, but not semi-active, then we can construct another feasible schedule $(\hat{S},\hat{A})$ with the same job-to-machine assignments and the same job orders on each machine, such that:

$$\forall j \in \mathscr{J}: \; \hat{S}_j \leq S_j, \tag{18}$$

$$\exists j \in \mathscr{J}: \ \hat{S}_j < S_j. \tag{19}$$

From this it follows that:

$$\forall j \in \mathscr{J}: \ \hat{C}_j \leq C_j, \tag{20}$$

$$\exists j \in \mathscr{J}: \ \hat{C}_j < C_j. \tag{21}$$

Job flowtime depends only on its completion time and ready time. However, since ready times are the same for both $(\hat{S},\hat{A})$ and (S,A) then:

$$\forall j \in \mathscr{J}: \ F_j(\hat{S},\hat{A}) \leq F_j(S,A), \tag{22}$$

similarly, we have:

$$\forall u \in \mathscr{B}: \ F^u(\hat{S},\hat{A}) \leq F^u(S,A), \tag{23}$$

and consequently:

$$\Sigma F(\hat{S},\hat{A}) \leq \Sigma F(S,A). \tag{24}$$

Now, if schedule $(\hat{S},\hat{A})$ is semi-active then $(\hat{S},\hat{A}) = (S',A')$ and the proof ends. If $(\hat{S},\hat{A})$ is not semi-active, then the above steps can be repeated. Each time we can construct a new schedule from the current not semi-active one, such that the completion time of at least one job decreases, while not increasing for any job. As a result, the total flowtime does not increase. However, due to job starting times, job ready times and the number of jobs being all integers, this process will eventually stop and we will arrive at a feasible schedule that is semi-active, which is the sought schedule (S',A'). □

This property leads to the following immediate conclusion:

Corollary 1. If (S,A) is a feasible schedule that is not semi-active, then (S,A) is not optimal.

This also means that if (S,A) is optimal, then it is semi-active.

Thus we do not want to consider schedules that are not semi-active. Now we will show that schedule represented by any processing order π is semi-active.

Property 2. If π is a processing order and (S,A) is a schedule that π represents, then (S,A) is semi-active.

Proof. First, let us show that (S,A) is feasible. For this (S,A) has to meet the 5 conditions mentioned in Section 2 for all jobs in $\mathscr{J}$. Let us recall that π contains each job from $\mathscr{J}$ exactly once. We also know that (S,A) can be derived from π using Eqs. (10)–(11). We will now show that those equations guarantee that the 5 mentioned feasibility conditions are met.

Eq. (10) ensures that $S_{\pi_i(j)} \geq r_{\pi_i(j)}$ and thus no job is processed before its ready time. Moreover, it guarantees that $S_{\pi_i(j)} \geq S_{\pi_i(j-1)}$ and thus all jobs from π_i will be processed in order of their appearance in π_i without overlapping with each other. This equation also ensures that setup time is performed before each job. Eq. (11) guarantees that $C_{\pi_i(j)} = S_{\pi_i(j)} + p_{\pi_i(j)}$ and thus each job will be processed for the exact required time $p_{\pi_i(j)}$ without interruption. Finally, Eq. (12) ensures that each job is assigned to some machine and processed on that machine only. Since the above applies to every job in $\mathscr{J}$, the schedule (S,A) is feasible.

Now, let us assume (S,A) is feasible but not semi-active. That would mean we can create a new feasible schedule (S',A') from (S,A) by shifting some job j to start earlier without: (1) changing job-to-machine assignments, (2) changing processing order on any machine and (3) shifting any other job to start later. Job starting times in (S,A) are determined by Eq. (10). Thus, shifting j to earlier time on the machine without changing the job order on that machine would mean one or more of the following:

1. Setup time before j was not performed fully.
2. j started before its ready time.
3. j started before the previous job on the same machine completed.
4. The job k which is processed on the same machine, but before j was shifted left by at least the same amount as the shift of job j.

The first 3 options would make the new schedule (S',A') infeasible. The last option would mean we would have to apply the same reasoning to job k as well, going back to the starting point. Eventually, one of the jobs we have shifted that way would fall under one of the first 3 options, making (S',A') infeasible. Thus, we cannot make job j start earlier on the same machine without changing the job order on that machine. The only options are to move job j to another machine or shift some of the other jobs to a later time. Thus, (S',A') either does not hold at least one of the conditions we assumed or is infeasible, leading to contradiction in both cases. Thus, schedule (S,A) has to be semi-active. □

Thus, any processing order π will only ever generate schedules that are not only feasible, but also semi-active. We will now show that similar “converse” property is also true.

Property 3. If (S,A) is a semi-active schedule then there is exactly one processing order π that represents (S,A).

Proof. We know (S,A) is feasible, so jobs on each machine are processed in certain order and each job is assigned to exactly one machine. From that it is trivial to construct candidate processing order π that can represent (S,A). We need to prove that no other processing order π' can represent (S,A) and that π does not represent any other semi-active schedule (S',A').

The former case is trivial. Indeed, if $\pi' \neq \pi$ then either some pair of jobs on some machine is processed in a different order (making at least one jobs starting time not match what S requires) or some job is performed on a different machine than assignments A require. Thus, $\pi' \neq \pi$ does not represent (S,A).

The latter case is more complex. If $(S',A') \neq (S,A)$ then either $A' \neq A$ or $S' \neq S$. In the first case some job is performed on different machine in (S',A') than in (S,A), so π cannot represent (S',A'). In the second case some job j is started at different time in (S',A') than in (S,A), but j is still processed on the same machine in both schedules. If this is done by changing the order of jobs on the considered machines in (S',A') compared to (S,A) then π cannot represent (S',A'). Thus, if (S',A') is represented by π then the shift of the job j has to happen without changing the job order on its machine.

Thus, two more possibilities remain. If j started in (S',A') earlier than in (S,A) then (S,A) would not be semi-active or (S',A') would not be feasible and, thus, semi-active (either leading to contradiction). Lastly, if j started in (S',A') later than in (S,A) then the same result would happen with (S',A') and (S,A) swapping roles. Thus, π cannot represent $(S',A') \neq (S,A)$. $\square$

The above results justify the use of processing orders instead of dealing with schedules directly, as processing orders generate only feasible and semi-active solutions. Moreover, every semi-active schedule can be considered this way and there is no redundancy, *i.e.* function from the set of all processing orders into all semi-active schedules is a bijection.

Let us now consider the relation between the cluster scheduling problem on parallel machines and the classic scheduling problem on parallel machines. Let $\mathrm{C}_m^k|r_j,s_{jk}|\Sigma F$ be a $\mathrm{C}_m|r_j,s_{jk}|\Sigma F$ problem with an additional constraint that the number of jobs in a family of jobs is bound by k:

$$\forall u \in \mathscr{B}^u: \ |\mathscr{B}^u| \leq k. \tag{25}$$

Similarly, we define $\mathrm{C}_m^k|online\text{-}time\text{-}sclv, r_j, s_{jk}|\Sigma F$. Then the following property holds:

Property 4. Problem $\mathrm{C}_m|r_j,s_{jk}|\Sigma F$ is a generalization of problem $\mathrm{P}_m|r_j,s_{jk}|\Sigma F$.

Proof. We will start by showing that problems $\mathrm{P}_m|r_j,s_{jk}|\Sigma F$ and $\mathrm{C}_m^1|r_j,s_{jk}|\Sigma F$ (or P and C^1 for brevity) are equivalent. We will do this by showing that any instance of the first problem can be transformed to an instance of the second and that those two problems have the same sets of possible schedules.

The set of machines is identical in both problems. For every job j in P we create a family of jobs $\mathscr{B}^u$ in C^1, such that $\mathscr{B}^u$ has a single job. Thus, job j in P corresponds to job j in C^1. We then set the processing and ready times of the corresponding jobs to be the same in both problems. Next step, the setup times, is similar: for P we define setup time between jobs j and k as s_{jk}, which is the setup time between corresponding jobs in C^1. Thus, we can transform any instance of problem P into an instance of problem C^1.

Next, for problem P we define values S_j, A_j and C_j to match those values in problem C^1 for corresponding jobs. Thus, we have constructed schedule (S,A) for P. It is easy to see that every schedule feasible for problem P is also feasible for corresponding problem C^1.

The last step is equivalence of the goal functions for both problems. For problem P the total flowtime is defined as:

$$\Sigma F(S,A) = \sum_{j\in\mathscr{J}} C_j - r_j. \tag{26}$$

For the problem C^1 the total flowtime is:

$$\Sigma F(S,A) = \sum_{u=1}^{q} F^u(S,A) = \max_{j\in\mathscr{B}^u}(C_j - r_j). \tag{27}$$

However, since every family of jobs in C^1 has only one job this is simplified to:

$$\Sigma F(S,A) = \sum_{j\in\mathscr{J}} C_j - r_j. \tag{28}$$

Thus, the values of $f(S,A)$ are the same for both problems. This proves the problems P and C^1 are equivalent.

Finally, we notice that every instance of problem $\mathrm{C}_m^k|r_j,s_{jk}|\Sigma F$ is also an instance for problem $\mathrm{C}_m|r_j,s_{jk}|\Sigma F$, hence the latter is generalization of the former. This, with the equivalence of $\mathrm{C}_m^1|r_j,s_{jk}|\Sigma F$ and $\mathrm{P}_m|r_j,s_{jk}|\Sigma F$ means that problem $\mathrm{C}_m|r_j,s_{jk}|\Sigma F$ is a generalization of problem $\mathrm{P}_m|r_j,s_{jk}|\Sigma F$. $\square$

Similar properties hold with the online and semi-clairvoyant versions of the problem.

Property 5. Problem $\mathrm{C}_m|online\text{-}time\text{-}sclv, r_j, s_{jk}|\Sigma F$ is a generalization of problem $\mathrm{P}_m|online\text{-}time\text{-}sclv, r_j, s_{jk}|\Sigma F$.

Proof. The proof is similar to the proof of Property 4. $\square$

The consequences of the above properties will be useful in the next section.

4. Algorithm properties

In this section we will discuss a family of online algorithms for the cluster scheduling problem or some of its variants.

One of the most-well known algorithms for the classic problem of scheduling on parallel machines is the Shortest Remaining Processing Time algorithm or SRPT. If there are uncompleted jobs, the algorithm schedules them on idle machines starting with the job with the least processing time remaining. For the preemptive version of the problem, *i.e.* $\mathrm{P}_m|online\text{-}time, pmnt, r_j|\Sigma F$, the SRPT algorithm is an $O\left(\log\left(\min\left\{\frac{n}{m}, P\right\}\right)\right)$-approximation algorithm, where $P = \dfrac{\max p_j}{\min p_j}$ [20]. SRPT is also the best, up to a constant factor, online algorithm for this problem. For the non-preemptive case, *i.e.* $\mathrm{P}_m|online\text{-}time, r_j|\Sigma F$, the SRPT is an $O\left(\sqrt{\frac{n}{m}}\log\left(\min\left\{\frac{n}{m}, P\right\}\right)\right)$-approximation algorithm.

We will now propose a new rule-based online algorithm, serving as the SRPT counterpart for the considered cluster scheduling problem. For this we first define the remaining load of family of jobs $\mathscr{B}^u$ as the sum of processing times of all unstarted jobs in $\mathscr{B}^u$. Now, we define the Smallest Remaining Load (or SRL) algorithm as follows.

Whenever a machine i becomes idle (including at time 0) we determine the set of all families of jobs that have at least one unstarted job. If the set is not empty, then we choose from it the family of jobs with the smallest remaining load. From that family of jobs we choose an arbitrary job and schedule it on machine i. The following property holds.

Property 6. The lower bound on the competitive ratio of the SRL algorithm for the problem $\mathrm{C}_m|online\text{-}time, r_j|\Sigma F$ is $O\left(\sqrt{\frac{n}{m}}\log\left(\min\left\{\frac{n}{m}, P\right\}\right)\right)$.

Proof. Let us consider the $\mathrm{C}_m^1|online\text{-}time, r_j|\Sigma F$ problem. We know this problem is equivalent to the $P_m|online\text{-}time, r_j|\Sigma F$ problem. It is also easy to see that the SRL algorithm for $\mathrm{C}_m^1|online\text{-}time, r_j|\Sigma F$ behaves exactly like the SRPT algorithm behaves for the $P_m|online\text{-}time, r_j|\Sigma F$ problem. Thus, the lower bound for the SRL algorithm for the $\mathrm{C}_m^1|online\text{-}time, r_j|\Sigma F$ is $O\left(\sqrt{\frac{n}{m}}\log\left(\min\left\{\frac{n}{m}, P\right\}\right)\right)$.

Moreover, it is well-known that if an online problem π_G is generalization of an online problem π then the competitive ratio of π_G is at least as large as that of problem π. Thus, the competitive ratio of the SRL algorithm for the $\mathrm{C}_m|online\text{-}time, r_j|\Sigma F$ problem will be at least as large as it is for the $\mathrm{C}_m^1|online\text{-}time, r_j|\Sigma F$ problem. □

We also can extend this property to the variant of the problem with setup times:

Corollary 2. The lower bound on the competitive ratio of the SRL algorithm for the problem $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ is $O\left(\sqrt{\frac{n}{m}}\log\left(\min\left\{\frac{n}{m}, P\right\}\right)\right)$.

Proof. This follows because every instance of problem $\mathrm{C}_m|online\text{-}time, r_j|\Sigma F$ is an instance of problem $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ with $s_{jk} = 0$. □

The SRL algorithm always schedules an unstarted job from the incomplete family of jobs u with the Smallest Remaining Load. However, u usually has more than one unstarted job. Thus, the SRL algorithm for the $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ is, in fact, a two-level algorithm. On the upper level we choose the family of jobs u. On the lower level we choose an unstarted job from that family. In essence, one can construct multiple versions of the SRL algorithm. Let SRL_w denote the version of the SRL algorithm that chooses an unstarted job according to some strategy w. We will now show that many of such SRL_w algorithms are not competitive against each other. For that we will need to define a class of algorithms under consideration.

Definition 1. The pair $(\mathrm{SRL}_v, \mathrm{SRL}_w)$ of two algorithms for the $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem is called *divergent* if and only if there exist instances I_A and I_B such that:

1. I_A has one family of jobs with two jobs with ready times 0 and processing times A_1 and A_2.
2. I_B has one family of jobs with two jobs with ready times 0 and processing times B_1 and B_2.
3. $A_1 > 1$ and $B_1 > 1$.
4. $A_2 > A_1$ and $B_2 > B_1$.
5. $A_2 - A_1$ and B_1 are arbitrary large.
6. When run on instance I_A (I_B) the SRL_v algorithm will schedule the job with processing time A_1 (B_1) first, while SRL_w will schedule job with processing time A_2 (B_2) first.

One example of two SRL_w algorithms that are divergent is the one which schedules shortest remaining job first (SRL_{SJ}) and the one which schedules longest remaining job first (SRL_{LJ}) when $A_1 = 2$, $A_2 = c$, $B_1 = c$, $B_2 = c + 1$, where c is some arbitrary large integer.

Now we can formulate the following theorem for the relative competitiveness of divergent algorithms.

Theorem 1. Let $(\mathrm{SRL}_1, \mathrm{SRL}_2)$ be a pair of divergent algorithms for the $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem. Let $\mathrm{SRL}_1(I)$ and $\mathrm{SRL}_2(I)$ be the solution (total flowtime) obtained for the given problem instance I by both algorithms, respectively. Then for any $r \in \mathbb{R}_+$ there exist instances I_1 and I_2 such that:

$$\frac{\mathrm{SRL}_1(I_1)}{\mathrm{SRL}_2(I_1)} \geq r, \tag{29}$$

$$\frac{\mathrm{SRL}_2(I_2)}{\mathrm{SRL}_1(I_2)} \geq r. \tag{30}$$

Proof. It is sufficient to prove the theorem for specific case of the $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem. Thus, we assume $m = 1$ and $s_{jk} = 0$.

Let us start with the first case. We construct instance I_1 to have $q + 1$ families of jobs as follows. The first family is made like the family from instance I_B from Definition 1 (this is possible as SRL_1 and SRL_2 are divergent). The remaining q families have one job each. Those jobs have ready times $B_2, B_2 + 1, \ldots, B_2 + q - 1$ and processing time 1.

For instance, I_1 algorithm SRL_1 will schedule job B_1 first. Since $B_1 < B_2$ the job will complete before family 2 is ready and thus job B_2 will be scheduled. Thus, the first family will have the flowtime of $B_1 + B_2$. However, the second family will be started at time $B_1 + B_2$ and will complete at time $B_1 + B_2 + 1$. Its flowtime will thus be $B_1 + 1$. Similar situation will happen for all $q - 1$ remaining families and they will have the same flowtime. The total flowtime $\mathrm{SRL}_1(I_1)$ will thus be:

$$\mathrm{SRL}_1(I_1) = q(B_1 + 1) + B_1 + B_2\,. \tag{31}$$

Let us now consider algorithm SRL_2. The algorithm will first schedule job B_2 which will complete at time B_2. The second family will then arrive with one job with processing time 1. The algorithm will choose to schedule that new family, since the only other choice is to schedule the remaining job B_1 of the first family. But since both algorithms operate on the general SRL principle and $B_1 > 1$, that option is forbidden. Thus, the job of the new family will start at time B_2, complete at time $B_2 + 1$ and will have flowtime of 1. When that family completes, similar will happen for subsequent $q - 1$ families, all will end up having flowtime of 1. After they all have been completed, finally the job B_1 of the first family will be scheduled. It will start at time $B_2 + q$ and complete at time $B_1 + B_2 + q$. Since the ready time of the first family was 0, its flowtime will be $B_1 + B_2 + q$. The total flowtime $\mathrm{SRL}_2(I_1)$ will thus be:

$$\mathrm{SRL}_2(I_1) = q + B_1 + B_2 + q. \tag{32}$$

We can now compute the ratio of flowtimes from expressions (31) and (32) when the number of families q is large:

$$\lim_{q\to\infty}\frac{\mathrm{SRL}_1(I_1)}{\mathrm{SRL}_2(I_1)} = \lim_{q\to\infty}\frac{q(B_1+1)+B_1+B_2}{2q+B_1+B_2} = \frac{B_1+1}{2}\,. \tag{33}$$

It is easy to see that the above expression approaches infinity when $B_1 \to \infty$. Thus, for any $r \in \mathbb{R}_+$ we can find such values q and B_1 that:

$$\frac{\mathrm{SRL}_1(I_1)}{\mathrm{SRL}_2(I_1)} \geq r. \tag{34}$$

Let us now consider the second case. We construct instance I_2 similarly to instance I_1, except the first family is like family I_A from Definition 1 and the jobs of the remaining families have ready times A_1, A_1+1, $\ldots$, A_1+q-1.

This time SRL$_1$ algorithm will schedule A_1 and when that job is completed, the second family will arrive. Since $A_2 > 1$ the job of that new family will be scheduled first. Thus, that and subsequent families will end up having flowtime of 1. When those q families complete, the job A_2 of the first family will be scheduled at time A_1+q and will complete at time A_1+A_2+q for the flowtime of A_1+A_2+q. Thus, the total flowtime SRL$_1(I_2)$ for this case is:

$$\mathrm{SRL}_1(I_2) = 2q + A_1 + A_2. \tag{35}$$

Finally, SRL$_2$ algorithm will start by scheduling job A_2 first. When that job is completed at time A_2, then the algorithm will start to schedule the job from the newly arrived family, which will thus start at A_2 and complete at A_2+1 for a flowtime of A_2-A_1+1. Similar will happen for the remaining families. At last, the job A_1 will be scheduled at time A_2+q and completed at A_1+A_2+q for a flowtime of A_1+A_2+q. The total flowtime SRL$_2(I_2)$ will be:

$$\mathrm{SRL}_2(I_2) = q(A_2 - A_1 + 1) + A_1 + A_2. \tag{36}$$

As before, we compute the ratio of flowtimes for large q:

$$\lim_{q\to\infty} \frac{\mathrm{SRL}_2(I_2)}{\mathrm{SRL}_1(I_2)} = \lim_{q\to\infty} \frac{q(A_2 - A_1 + 1) + A_1 + A_2}{2q + A_1 + A_2} = \frac{A_2 - A_1 + 1}{2}. \tag{37}$$

Once again, the expression approaches infinity when $A_2 - A_1 \to \infty$. Thus, for any $r \in \mathbb{R}_+$ we can find such values q, A_1 and A_2 that:

$$\frac{\mathrm{SRL}_2(I_2)}{\mathrm{SRL}_2(I_2)} \geq r. \tag{38}$$

□

The above theorem proves divergent algorithms are not r-competitive relative to each other for any r. This also means that divergent algorithms are not r-competitive for any r:

Corollary 3. Let $(\mathrm{SRL}_1, \mathrm{SRL}_2)$ be a pair of divergent algorithms for the $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem. Let SRL$_1(I)$ and SRL$_2(I)$ be the solution (total flowtime) obtained for the given problem instance I by both algorithms, respectively. Also let OPT(I) be the optimal total flowtime for instance I. Then for any $r \in \mathbb{R}_+$ there exist instances I_1 and I_2 such that:

$$\frac{\mathrm{SRL}_1(I_1)}{\mathrm{OPT}(I_1)} \geq r, \tag{39}$$

$$\frac{\mathrm{SRL}_2(I_2)}{\mathrm{OPT}(I_2)} \geq r. \tag{40}$$

Proof. The first inequality follows because from Theorem 1 we know SRL$_1(I_1)$ can be r times greater than SRL$_2(I_1)$ and SRL$_2(I_1) \geq$ OPT(I_1). The second inequality is similar. □

5. Algorithms comparison

In the previous section we discuss theoretical properties and competitiveness of some online algorithms for the $\mathrm{C}_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem and its variants. In this section we perform a set of computer experiments in order to support the previously stated properties and show the average performance of online algorithms in a more practical setting.

All experiments were performed as simulations on a machine with Intel Core i708650U 1.9GHz CPU, 16 GiB of RAM working under 64-bit Linux (5.3.0 kernel). Algorithms were implemented in C++ and compiled with gcc/g++. The testing environment for running experiment was prepared using a mix of C++ and Linux scripts (written in Bash).

The first experiment considers five different variants of the SRL algorithm. Thus, all five algorithms choose some unstarted job from the family of jobs with the smallest sum of processing times of unstarted jobs. However, they differ in which job from that family is chosen:

1. SRL$_{\mathrm{LJ}}$ – (Longest Job) chooses the job with the longest processing time.
2. SRL$_{\mathrm{SJ}}$ – (Shortest Job) chooses the job with the shortest processing time.
3. SRL$_{\mathrm{AVG}}$ – (Average) chooses the job with processing time closest to the average processing time of remaining jobs.
4. SRL$_{\mathrm{RAVG}}$ – (Reverse Average) the reverse of SRL$_{\mathrm{AVG}}$. Chooses the job that SRL$_{\mathrm{AVG}}$ would choose last.
5. SRL$_{\mathrm{RAND}}$ – (Random) chooses random job.

In order to observe the differences between the algorithms better, we restricted the setting to a single family of jobs, two machines and assumed no setup times. The number of jobs in a family was drawn from a discrete uniform distribution from 2 to 20 ($\mathscr{U}\{2,20\}$), while job processing times were drawn from a $\mathscr{U}\{1,100\}$ distribution. 100 000 such random instances were tested. Each total flowtime was normalized compared to the worst algorithm for the given instance. Thus, value of 0.5 for instance I means the algorithm provided twice as good total flowtime as the worst algorithm for I. The average and maximum values over 100 000 instances for each algorithm are shown in Table 1. The values of maximum confirm that each of the con-

Table 1
Average and maximum normalized flowtimes for several variants of the SRL algorithm

Algorithm	Average	Maximum
SRL$_{\mathrm{LO}}$	0.863	1.000
SRL$_{\mathrm{SO}}$	0.984	1.000
SRL$_{\mathrm{AVG}}$	0.900	1.000
SRL$_{\mathrm{RAVG}}$	0.918	1.000
SRL$_{\mathrm{RAND}}$	0.910	1.000

sidered algorithms has a "bad" instance for which it performs badly. As for average values, we observe that the "shortest job" strategy is the worst of all considered strategies almost all the time. On the other hand, the "longest job" strategy turned to be the best one, being 14% better than "shortest job" and 5% better than random strategy.

The second computer experiment considered comparing SRL_{LJ} against several other online algorithms:

1. LRL (Largest Remaining Load) – incomplete family of jobs with the largest remaining load is chosen first.
2. SJ (Shortest Job) – shortest unstarted remaining job is chosen first (regardless of which family it belongs to).
3. LJ (Longest Job) – longest unstarted job is chosen, similar to SJ.
4. FIFO (First In First Out) – incomplete families of jobs are chosen in the order of their ready times (First-In First-Out).
5. RAND (Random) – random incomplete family of jobs is chosen.
6. MIN – first a family of jobs is chosen from all incomplete families in a cyclic, round-robin-like fashion. Then the shortest job of that family is chosen.
7. MAX – similar to MAX, but the longest job is chosen.

The MIN and MAX algorithms were considered based on the algorithms implemented in a certain real-life Testing-as-a-Service system. The data from that system was used to construct 50 problem instances of roughly 1 800 jobs each. The setup time s_{jk} was set to 30, modeling the time needed to transfer necessary files in the referenced real-life system. The algorithms were ran for several number of machines. The most important results, for 30 and 400 machines, are shown in Figs. 2 and 3 in the form of boxplots.

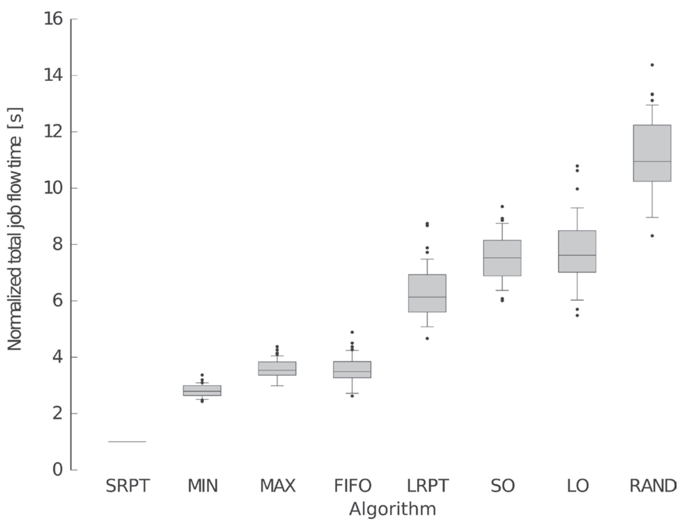

Fig. 2. Normalized flowtime bloxplots for 30 machines

Let us start with the case of 30 machines. The SRL is clearly the best algorithm, as all values are very close to 1. Even the second closest algorithm, MIN, is outperformed by a large margin, its total flowtime being from 2.5 to nearly 4 times worse than for SRL (3 times on average). Each subsequent algorithm generally performs worse than the previous one. Finally, in the case of the RAND algorithm (which serves as a reference point of sorts) we get 9 to 14 times worse values of the total flowtime. Thus, the SRL provides the best and most stable (very small variance) results in this case.

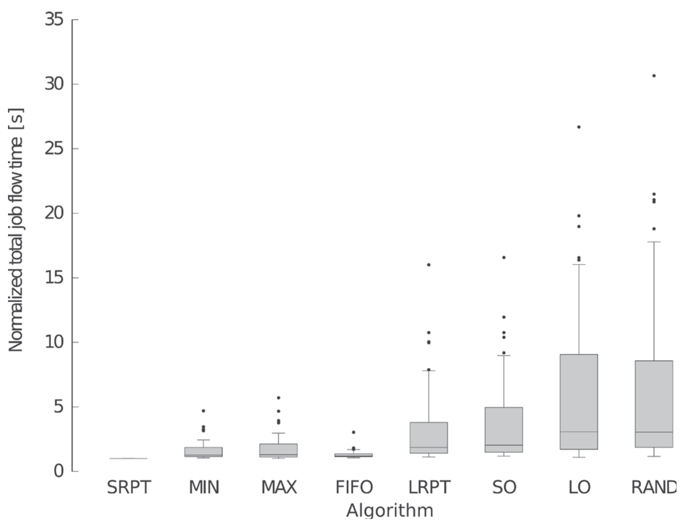

Fig. 3. Normalized flowtime bloxplots for 400 machines

When the number of machines is increased to 400, the situation changes, slightly, but the general conclusion remains the same. The SRL algorithm provides the smallest total flowtime with little variance. Due to the increased number of machines the average values for the other algorithms have improved greatly (staying close to 1, even for the RAND algorithm). However, it is at the cost of the maximum values.

To summarize the above research, the SRL algorithm is the best choice from all priority-rules online algorithms for the $C_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem that we considered. The SRL_{LJ} subvariant of SRL provides the most promising results.

6. Simulated annealing approach

The results presented in Sections 4 and 5 showed that the SRL algorithm and its variants generally outclass many other online scheduling algorithms for the $C_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem. However, that does not answer a more general question, namely: how much better algorithm for the considered problem can we get? Let us note that the algorithms we considered so far were so-called constructive algorithms, choosing the next job based on priority rules. Such algorithms create only one candidate solution and do not even evaluate it, thus essentially not performing any search of the solution space.

There exists a completely different category of algorithms called metaheuristics. Metaheuristics are also inexact methods that are faster than exact methods, but do not guarantee obtaining the optimal solution. They also have higher computational cost when compared to the constructive algorithms like SRL. However, metaheuristics are able to evaluate many candidate solutions and compare them, performing a controlled search of the solution space. Metaheuristics are a very diverse group of methods, but most of them work in an iterative manner. In each iteration, the method tries to obtain a better solution than in the last iteration. This means a metaheuristic has to start from some initial solution, which is usually supplied by some other heuristic method like the SRL algorithm. That also means that, in theory, a metaheuristic will not obtain the final results that are worse than the initial solution. Metaheuristics are commonly used for solving many offline scheduling problems and some were used for online problems as well, as shown in Section 1. However, another study [10] for a similar TaaS system indicated that the performance of metaheuristics was not better than that of the SRL algorithm. We will now present a more detailed study of this phenomenon.

6.1. Solving algorithm. We start by describing the solving method used in this study. The method works by applying a certain metaheuristic every time a new family of jobs arrives[1]. This schedules all unstarted jobs, including the newly arrived jobs and the ones already scheduled but not yet started. Thus, previously scheduled jobs might be rescheduled.

We decided to employ Simulated Annealing (SA) method, which is one of the most well-known metaheuristics used in optimization problems, including scheduling [27]. Our implementation of SA follows the classical scheme well-known in the literature [28], with a few adjustments. The method used the processing order π introduced in Section 2. This allows employing a swap-or-move neighborhood. Thus, a transition between solutions happens either by swapping jobs inside a specific vector π_i or by moving a job between different vectors. The cooling scheme and solution acceptance probability are defined as follows:

$$\alpha(i) = t_0 - i/I, \tag{41}$$

$$A_{xy}(t) = \begin{cases} 1 & \text{when } f(y) \le f(x), \\ \exp\left(\dfrac{f(x)-f(y)}{t}\right) & \text{otherwise}, \end{cases} \tag{42}$$

where $\alpha(i)$ is the temperature at iteration i, $A_{xy}(t)$ is acceptance probability of transition from solution x to y given temperature t, t_0 is starting temperature, I is the total number of iterations and f is cost function (total flowtime for a given solution).

Lastly, a metaheuristic requires an initial solution and two different variants were prepared. The first variant uses the solution provided by the $\mathrm{SRL_{LJ}}$ algorithm as the initial solution. The second variant runs the $\mathrm{SRL_{LJ}}$ algorithm first and then worsens the result by making 25 random swap transitions.

6.2. Computer experiment. To provide details on the phenomenon of metaheuristic approach to scheduling for the considered TaaS system, we conducted a computer experiment. In the experiment, we compared the performance of both SA variants mentioned earlier with the performance of the $\mathrm{SRL_{LJ}}$ algorithm. The instances for this experiment were generated using probability distributions derived from the workload characteristics of a real-life TaaS system [6].

In the experiment 50 instances were used for 4 numbers of machines (30, 50, 100 and 200), resulting in 200 subexperiments in total. The $\mathrm{SRL_{LJ}}$ is deterministic, so it is run once, while both SA-based method variants were run 7 times for each subexperiment, with the reported values being the averages. Also, the SA-based method was run for different number of iterations. The results of each subexperiment were normalized with the value of 1 being the result provided by $\mathrm{SRL_{LJ}}$. The software and hardware used in the experiment are the same as the ones described in the experiment from Section 5.

In Fig. 4 we show the results when the SA-based method starts from the worsened $\mathrm{SRL_{LJ}}$ algorithm. We notice that with the growing number of iterations, the effectiveness of the SA method gets closer to that of the $\mathrm{SRL_{LJ}}$ algorithm. In particular, the range of values obtained by the SA method decreases: at 1 iteration SA is from around 1.4 to 5.09 times worse than $\mathrm{SRL_{LJ}}$. However, at 100 000 iterations the values are from 0.98 to 1.4. Let us note that the SA method manages to outperform the $\mathrm{SRL_{LJ}}$ algorithm, but very rarely and unreliably. It also takes a large number of iterations for it to get close to the results provided by the $\mathrm{SRL_{LJ}}$ algorithm. However, we can conclude that as the number of iterations grows, the SA method tends to get closer to the result provided by the $\mathrm{SRL_{LJ}}$ method, which is an expected behavior.

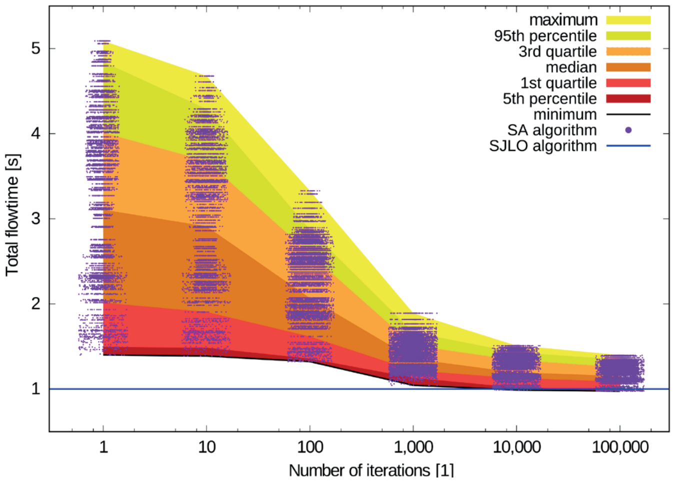

Fig. 4. Overall total flowtime results for 8400 runs of the SA-based method in function of the number of iterations allowed for SA. Initial solution was worsened $\mathrm{SRL_{LJ}}$ algorithm. The X axis is in log scale

For the next step, we made the SA method start directly from the $\mathrm{SRL_{LJ}}$ algorithm. Since the initial solution is now of higher quality, we expect the SA method to perform better. The results of this experiment are shown in Fig. 5. The outcome is largely surprising. First, we notice that the SA method on average provides worse results than its starting solution. In fact, up to 1 000 iteration the SA method almost always generates results worse than the $\mathrm{SRL_{LJ}}$ algorithm. When the number of iterations increases further, the best values become better, but the results do

[1]If several families arrive at the same time, the method is run only once.

not go below 0.97 of what the SRL$_{\text{LJ}}$ can provide. Moreover, as for the maximal values and most other ordinals we observe a surprising behavior. Beyond 1 000 iterations, the average results obtained by the SA method get worse as the number of iterations increases. While the values never get as bad as when the SA method was starting from a worsened solution, the effect is still counter-intuitive.

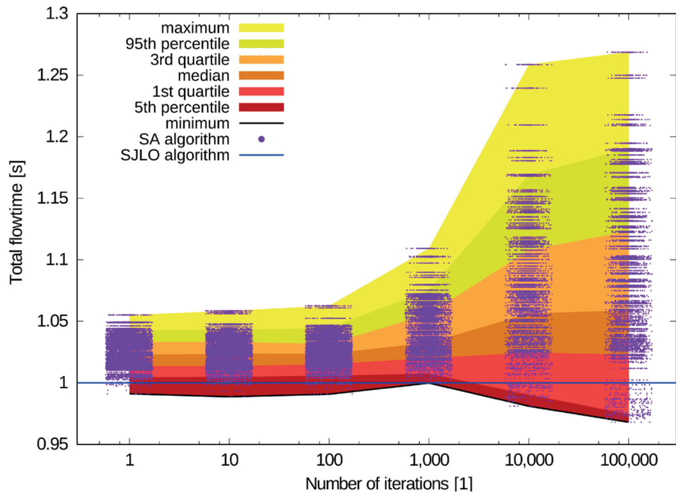

Fig. 5. Overall total flowtime results for 8400 runs of the SA-based method in function of the number of iterations allowed for SA. Initial solution was the SRL$_{\text{LJ}}$ algorithm. The X axis is in log scale

If we were to look at Figs. 4 and 5 when joined together, we would notice that the shape of curves pertaining to most of the ordinals (at least from the first quartile to the maximum) resemble a bathtub curve. Of course, it would be best to portray this effect on a single figure; however, due to the time needed to run the SA-based method 1 400 times for a number of iterations greater than 100 000, it was difficult to obtain the bathtub curve with a single setting. This is why two different variants for the SA-based method were used.

A possible explanation of the observed phenomenon might be the greedy-like way the solving method is applied for the considered problem. The method tries to solve an instance with n families of jobs by applying the SA-based method $O(n)$-times. Thus, each execution of the SA method acts on the limited part of the instance (with limited knowledge) and tries to find the best solution. However, that best solution is not the solution to the entire problem, so it might not be a global optimum. This might be likened to a game of chess, where a greedy algorithm is run for every move. However, a game constructed from moves that are locally optimal might not be globally optimal. Thus, for our solving method, increasing the number of iterations enabled the SA method to get closer to the local optimum, which might make it return worse solution globally. This also makes it possible for a metaheuristic like SA to return a result that is worse than its initial solution, which is also counter-intuitive for a metaheuristic.

To summarize, in this section we presented the results of applying an exemplary metaheuristic to the problem of scheduling tasks for the considered real-life TaaS system. Results indicate that, at the current time, metaheuristics are not a viable choice for this problem. Even if they can provide solutions better than SRL-like algorithms, they have much higher computational complexity and the possible gain is not significant (around 3%). The most important conclusion, however, is that the metaheuristics do not seem to be reliable. Even if one out of 100 runs of the proposed solving method were to be successful, we do not know which one it will be. Coupled with the fact that we have to create the schedule on-the-fly, it makes the metaheuristics impractical at the current time.

7. Processing time prediction

In the previous sections we have considered the $C_m|online\text{-}time, r_j, s_{jk}|\Sigma F$ problem with the assumption that the processing times of jobs are known in advance and the problem is clairvoyant. However, in practice the processing times are not known for certain. The prediction of such uncertain parameters is always difficult, even for single-machine scheduling problems [29]. If we were not able to approximate the processing times of jobs, then the considered cluster scheduling problem would be reduced to typical load-balancing problem.

This issue is especially important as we are trying to model a Testing-as-a-Service cloud system. In such a system changing a single line of code can affect many test cases (jobs) and drastically change their processing time. However, a single test case is executed many times on average. Thus, previous executions of a test case can be used to predict the time it would take to execute it again. In this section we will show that the processing times of jobs can be predicted well and thus the considered online scheduling problem is in fact semi-clairvoyant: $C_m|online\text{-}time\text{-}sclv, r_j, s_{jk}|\Sigma F$.

For prediction we employed the k-recent moving average technique. Thus, to predict processing time p_j of job j, we compute the average of the k most recent executions of jobs of type t_j. Such predicted value will be denoted as $\hat{p}_j$. Let H_t be the vector containing past processing times of jobs with type t and $H_t(x)$ be the x-th most recent processing time in that vector. Then the predicted value $\hat{p}_j$ is defined as follows:

$$\hat{p}_j = \begin{cases} \dfrac{1}{k}\displaystyle\sum_{x=1}^{k} H_{t_j}(x) & \text{if } |H_{t_j}| > k \wedge k > 0, \\ \dfrac{1}{|H_{t_j}|}\displaystyle\sum_{x=1}^{|H_{t_j}|} H_{t_j}(x) & \text{if } |H_{t_j}| \leq k \wedge |H_{t_j}| > 0, \\ 50 & \text{if } |H_{t_j}| = 0 \vee k = 0, \end{cases} \tag{43}$$

where 50 is the default value used when the history is empty. This value was chosen based on the average processing time in the testing instances mentioned earlier in the paper.

It is also possible to include weights in the prediction mechanism, with weight decreasing as x increases. Such a system would help take freshness of data into account, leading to the following formula:

$$\hat{p}_j = \begin{cases} \dfrac{1}{W(k)} \displaystyle\sum_{x=1}^{k} w_x H_{t_j}(x) & \text{if } |H_{t_j}| > k \wedge k > 0, \\ \dfrac{1}{W(|H_{t_j}|)} \displaystyle\sum_{x=1}^{|H_{t_j}|} w_x H_{t_j}(x) & \text{if } |H_{t_j}| \leq k \wedge |H_{t_j}| > 0, \\ 50 & \text{if } |H_{t_j}| = 0 \vee k = 0, \end{cases} \tag{44}$$

where $W(k)$ and $W(|H_{t_j}|)$ are the sum of the weights of the k and $|H_{t_j}|$ most recent processing times, respectively.

We can now use the value $\hat{p}_j$ to predict the remaining load L_u of family of job u (which is the sum of processing times of its jobs). The resulting value is denoted $\hat{L}_u$:

$$\hat{L}_u = \sum_{j \in \mathscr{B}^u} \hat{p}_j \,. \tag{45}$$

Through preliminary research we concluded that the best value for k is 3. After that we performed two series of computer experiments. The purpose of the first experiment was to research the Person correlation coefficient between values $\hat{p}_j$ and p_j as well as between $\hat{L}_u$ and L_u. The results for the 50 testing instances mentioned earlier are shown in Table 2. The software and hardware used in the experiment are the same as the ones described in experiment from Section 5.

Table 2
Pearson correlation coefficient for prediction of job processing times and family of job loads for the k-recent prediction method

Instance	$\hat{L}_u$	$\hat{p}_j$	Instance	$\hat{L}_u$	$\hat{p}_j$
inst01	0.989	0.912	inst26	0.996	0.922
inst02	0.971	0.829	inst27	0.996	0.928
inst03	0.920	0.839	inst28	0.999	0.943
inst04	0.941	0.863	inst29	0.997	0.927
inst05	0.994	0.891	inst30	0.997	0.916
inst06	0.936	0.855	inst31	0.996	0.890
inst07	0.978	0.857	inst32	0.997	0.918
inst08	0.991	0.917	inst33	0.997	0.914
inst09	0.964	0.915	inst34	0.997	0.918
inst10	0.995	0.908	inst35	0.994	0.908
inst11	0.995	0.926	inst36	0.998	0.932
inst12	0.987	0.957	inst37	0.996	0.911
inst13	0.996	0.949	inst38	0.997	0.940
inst14	0.997	0.945	inst39	0.945	0.898
inst15	0.998	0.898	inst40	0.998	0.951
inst16	0.995	0.931	inst41	0.998	0.937
inst17	0.996	0.937	inst42	0.997	0.906
inst18	0.996	0.915	inst43	0.998	0.937
inst19	0.995	0.937	inst44	0.936	0.876
inst20	0.995	0.929	inst45	0.996	0.902
inst21	0.960	0.923	inst46	0.997	0.913
inst22	0.996	0.910	inst47	0.997	0.906
inst23	0.995	0.897	inst48	0.998	0.932
inst24	0.997	0.922	inst49	0.950	0.911
inst25	0.997	0.947	inst50	0.998	0.936

We observe that the quality of the $\hat{p}_j$ predictor is good with average correlation coefficient at 0.914 and the minimum value was 0.829. The coefficient of variation (CV), which is standard deviation divided by the average, was 0.031. The prediction is not perfect though, so it is natural to think that using that value to predict the remaining load of family of jobs would provide worse results as the errors of prediction for each job would accumulate inside the family. The results from the table, however, show a different outcome. The average, minimum and CV values for the $\hat{L}_u$ predictor are 0.987, 0.920 and 0.020. Thus, the correlation between $\hat{L}_u$ and L_u was for all instances higher than the average correlation between $\hat{p}_j$ and p_j. We assume this effect is caused by prediction errors averaging and canceling each other over all jobs in a family.

For the second experiment we wanted to test the practical quality of prediction. Even if there is a high correlation between values, it is still possible that algorithm using the predictor would obtain a vastly different total flowtime than the "ideal" (clairvoyant) algorithm which uses the actual processing times. Thus, in our experiment we ran both variants (the one employing prediction and the one using the actual processing times) for 5 previously discussed online algorithms. We normalized the results (except CV) by dividing the total flowtime obtained with prediction by the one obtained using actual processing times. The summary of results is shown in Table 3.

Table 3
Normalized quality of the k-recent prediction method. All averages, minima, maxima, and CVs are computed over 50 tested instances

Algorithm	Average	Minimum	Maximum	CV
30 machines				
LRL	0.973	0.875	0.993	0.023
MAX	0.975	0.951	0.990	0.009
MIN	1.017	1.006	1.033	0.006
RAND	1.000	0.997	1.005	0.002
SRL	1.060	1.014	1.142	0.025
100 machines				
LRL	0.986	0.839	1.016	0.027
MAX	0.984	0.968	0.996	0.006
MIN	1.008	0.993	1.031	0.007
RAND	1.000	0.998	1.003	0.001
SRL	1.046	0.998	1.101	0.026
400 machines				
LRL	1.003	0.892	1.106	0.037
MAX	1.001	0.985	1.009	0.005
MIN	0.999	0.994	1.008	0.003
RAND	1.000	0.994	1.007	0.003
SRL	1.004	0.992	1.038	0.007

In general, we observe that all of the tested algorithms are fairly robust: the prediction not worsening the results by more than a few percent. The MIN, MAX and RAND algorithms also are very stable with small CV values: the prediction barely affects them. The LRL algorithm is very interesting as using the predicted value seems to work better than using the actual values. The SRL algorithm suffers the biggest prediction error, but still remains the best of the considered algorithms when it comes to optimizing the total flowtime for Testing-as-a-Service cloud system under consideration.

8. Discussion

In this section we will discuss the proposed approach, including its features, advantages, disadvantages and results presented in previous sections, as well as the possibility of applying the approach to different kinds of problems.

We will start by discussing the advantages of the approach. While the problem of running large-scale software test is related to the problem of distributing or balancing load in computer clusters, it is also very specific. This problem was considered before, but most authors either focus on practical issues, often considering many additional aspects (including test cases generation, flow of data in the company, aggregation of test case results, etc.). However, there is little focus on the theoretical properties of the problem. Our approach is partially theoretical and focuses on a single aspect of the problem: the scheduling. Thus, the approach using theory of scheduling allowed to better define and understand the problem, its limitation, difficulties and properties.

Such an approach allowed to extend the existing SRPT algorithm to this problem. We also notice that the resulting SRL algorithm is two-level and conducted research concerning entire class of SRL algorithms, including the fact that many of them are not competitive for any r. More importantly, the SRL is a very fast algorithm, which is important for large-scale software testing, where tens and hundreds of thousands of test cases are run every day.

The SRL is also effective, outclassing other tested algorithms. In particular, SRL outclasses the MIN algorithm, which was originally the one used in the company. The effectiveness of the SRL algorithm also allows to decrease the number of machines used for software testing. This can reduce costs (by buying and maintaining fewer resources) or allow to employ (now idle) machines for other tasks.

Finally, we proved that the considered problem is not typical load-balancing, because the processing times of test cases, though generally unknown, can be predicted with high accuracy using simple k-recent moving average predictor. As such, while our research are case study-specific, the proposed approach should be applicable for high-scale software testing in other companies as well.

We will now discuss some issues and disadvantages associated with considered approaches. While applicable for different software-testing workloads, the numerical experiments presented in this paper are very case-specific. The use of our approach in a different software testing system would still require prior analysis. Moreover, such analysis requires a considerable history of past executions of test cases to be available. While we expect companies to keep track of past data, that still means the approach cannot be fully applied from the start, but only after some time has passed.

Our research also indicated that while there are some promising results, the metaheuristics are generally inferior to simple rule-based algorithms, at least in our approach (also due to the large size of the problem). However, the research proved that metaheuristics can, in some contexts, provide results worse than their initial solutions. This counter-intuitive result means metaheuristics should be used carefully.

Finally, while the results, especially for the $\mathrm{SRL_{LJ}}$ algorithm are promising, the actual implementation in a real-life software system still would encounter issues, including delay caused by computer network, failure of machines and additional constraints (for example, different types of machines).

Lastly, we discuss the possibility of using our approach to other problems. First, the approach can be used for distributing workload in IT systems for purposes other than software testing. In fact, most computation-heavy tasks would suffice. For example, one can imagine a system for running programs for scientific experiments. Each user would generally submit a task (family of jobs in our context) composed of multiple subtasks (e.g. many versions of the same program run with different parameters). Thus, the owner of the system would minimize the total (equivalently average) time of completing the entire task, which conforms to our definition of the goal function. Otherwise, as a generalization of the classic scheduling jobs on parallel machines problem, our approach could be used for variety of systems, where jobs are packed into families. This could include password retrieval, where a single task (submitted by a single court expert) is composed of thousands of password masks to be checked.

9. Conclusions

In this paper we have considered the problem of scheduling test suites and test cases in a case study Testing-as-a-System cloud environment. The problem was modeled as an online clustered scheduling problem on parallel machines with the total flowtime goal function. We proved that this problem is a generalization of the classic problem of scheduling on parallel machines. We proposed a processing order representation that allows to consider only semi-active feasible schedules. We proposed the Smallest Remaining Load (SRL) online algorithm, based on the well-known Shortest Remaining Processing Time algorithm, and a number of its variants.

We showed that SRL algorithm variants belonging to a certain class are not competitive, not even relative to each other. Through computer experiments on synthetic and real-life data, we have shown that the $\mathrm{SRL_{LJ}}$ is the best of of the considered SRL variants in practice, outclassing other SRL variants and non-SRL algorithms. We also compared the SRL approach to the metaheuristic approach and showed that, at the current time, such approach leads to generally worse results, which is counter-intuitive for a metaheuristic.

Finally, we proposed an approach to prediction of the processing times of test cases using th k-recent method.

The method showed very high correlation between predicted and actual values for both test case processing times and test suite remaining load. Results from a computer experiment with real-life data confirmed that this approach is robust, with the prediction having little effect on the computed total flowtime value.

References

[1] D. Kumar and K. Mishra, "The impacts of test automation on software's cost, quality and time to market", *Procedia Comput. Sci.* 79, 8–15 (2016).

[2] J. Musial, M. Guzek, P. Bouvry, and J. Blazewicz, "A note on the complexity of scheduling of communicationaware directed acyclic graph", textitBull. Pol. Ac.: Tech. 66 (2), 187–191 (2018).

[3] L. Yu, L. Zhang, H. Xiang, Y. Su, W. Zhao, and J. Zhu, "A framework of testing as a service", in *2009 International Conference on Management and Service Science*, 2009, pp. 1–4.

[4] L. Yu, W. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a service over cloud", in *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, 2010, pp. 181–188.

[5] A. Sathe and D.R. Kulkarni, "Study of testing as a service (taas)–cost effective framework for taas in cloud environment", *International Journal of Application or Innovation in Engineering and Management (IJAIEM)* 2 (5), 239–243, (2013).

[6] P. Lampe and J. Rudy, "Models and scheduling algorithms for a software testing system over cloud", in *Contemporary Complex Systems and Their Dependability*, pp. 326–337, Eds. W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, and J. Kacprzyk, Cham: Springer International Publishing, 2019.

[7] S.-J. Lee, Y.-C. Lin, K.-H. Lin, and J.-L. You, "A system for composing and delivering heterogeneous web testing software as a composite web testing service", *J. Inf. Sci. Eng.* 34 (3), 631–648 (2018).

[8] J. Gao, X. Bai, and W. Tsai, "Cloud testing issues, challenges, needs and practice", *Software Engineering: An International Journal* 1(1), 9–23 (2011). [Online]. Available: http://www.seij.dce.edu/Paper%5Cn1.pdf

[9] R. V. Binder, "Optimal scheduling for combinatorial software testing and design of experiments", in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, 2018, pp. 295–301.

[10] J. Rudy, "Online multi-criteria scheduling for testing as a service cloud platform", in *Smart Innovations in Engineering and Technology*, pp. 34–52, Eds. R. Klempous and J. Nikodem, Cham: Springer International Publishing, 2020.

[11] S. Tahvili, "Multi-criteria optimization of system integration testing", Ph.D. dissertation, RISE SICS Västerås, 2018.

[12] P. Lampe, "Fuzzy job scheduling for testing as a service platform", in *Smart Innovations in Engineering and Technology*, pp. 25–33, Eds. R. Klempous and J. Nikodem, Cham: Springer International Publishing, 2020.

[13] A. Ali, H.A. Maghawry, and N. Badr, "Automated parallel gui testing as a service for mobile applications", *J. Software: Evol. Process* 30 (10), e1963 (2018), [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1963

[14] I. Gog, M. Schwarzkopf, A. Gleave, R.N.M. Watson, and S. Hand, "Firmament: Fast, Centralized Cluster Scheduling at Scale", in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 99–115.

[15] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao, "Efficient queue management for cluster scheduling", in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, New York, USA, 2016, pp. 36:1–36:15.

[16] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling", in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conferece*, Berkeley: USENIX Association, 2015, pp. 499–510. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813767.2813804.

[17] K. Lee, J.Y.-T. Leung, and M.L. Pinedo, "Makespan minimization in online scheduling with machine eligibility", *Ann. Oper. Res.* 204(1), 189–222 (2013).

[18] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel, "Job-aware scheduling in eagle: Divide and stick to your probes", in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, USA, 2016, pp. 497–509. [Online]. Available: http://doi.acm.org/10.1145/2987550.2987563

[19] C. Reiss, A. Tumanov, G.R. Ganger, R.H. Katz, and M.A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis", in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, New York, USA, 2012, pp. 7:1–7:13.

[20] S. Leonardi and D. Raz, "Approximating total flow time on parallel machines", *J. Comput. Syst. Sci. Int.* 73 (6), 875–891 (2007).

[21] G. Dósa, A. Fügenschuh, Z. Tan, Z. Tuza, and K. Węsek, "Tight lower bounds for semi-online scheduling on two uniform machines with known optimum", *Cent. Eur. J. Oper. Res.* 27(4), 1107–1130 (2019).

[22] G. Iordache, M. Boboila, F. Pop, C. Stratan, and V. Cristea, "Decentralized grid scheduling using genetic algorithms", in *Metaheuristics for Scheduling in Distributed Computing Environments*, Eds. F. Xhafa and A. Abraham, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 215–246.

[23] F. Xhafa, E. Alba, and B. Dorronsoro, "Efficient batch job scheduling in grids using cellular memetic algorithms", in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[24] R. Kizys, A. Juan, B. Sawik, and L. Calvet, "A biasedrandomized iterated local search algorithm for rich portfolio optimization", *Appl. Sci.* 9, 3509 (2019).

[25] Å. Dominik *et al.*, "Solving multi-objective job shop problem using nature-based algorithms: new pareto approximation features", *Int. J. Optim. Control, Theor. Appl.* 5 (1), 1–11 (2014).

[26] M. Kalra and S. Singh, "A review of metaheuristic scheduling techniques in cloud computing", *Egypt. Inform. J.* 16 (3), 275–295 (2015).

[27] M. Klimek and P. Łebkowski, "Financial optimisation of the scheduling for the multi-stage project", *Bull. Pol. Ac.: Tech.* 65 (6), 899– 908 (2017).

[28] K.A. Dowsland and J.M. Thompson, "Simulated annealing", in *Handbook of natural computing*. pp. 1623–1655, Springer, 2012.

[29] W. Bożejko, P. Rajba, and M. Wodecki, "Stable scheduling of single machine with probabilistic parameters", *Bull. Pol. Ac.: Tech.* 65 (2), 219–231 (2017).