Planning and Metaheuristic Optimization in Production Job Scheduler

Dalibor Klusáček and Václav Chlumský

CESNET a.l.e., Brno, Czech Republic
{klusacek,vchlumsky}@cesnet.cz

Abstract. In this work we present our positive experience with a unique advanced job scheduler which we have developed for the widely used TORQUE Resource Manager. Unlike common schedulers using queuing approach and simple heuristics, our solution uses planning (job schedule construction) and schedule optimization by a local search-inspired metaheuristic. Using both complex simulations and practical deployment in a real system, we show that this approach increases predictability, performance and fairness with respect to a common queue-based scheduler. Presented scheduler has been successfully used in the production infrastructure of the Czech Centre for Education, Research and Innovation in ICT (CERIT Scientific Cloud) since July 2014.

Keywords: Scheduling, Planning, Performance, Fairness, Simulation

1 Introduction

The pros and cons of queuing vs. planning have been discussed in the past thoroughly [5]. Classical queuing approaches such as the well known EASY backfilling algorithm provide limited predictability and thus employ additional mechanisms in order to avoid certain unwanted features such as potentially (huge) starvation of particular jobs, etc. Approaches based on full planning represent an opposite approach. Instead of using "ad hoc", aggressive scheduling, they build and manage an execution plan that represents job-to-machine mapping in time. For example, Conservative backfilling establishes reservation for every job that cannot execute immediately, i.e., a guaranteed upper bound of its completion time is always available [13]. This increases predictability but may degrade system performance and requires more computational power. At the same time, the accuracy of such predictions is typically quite low, as provided estimates concerning job execution are typically very imprecise and overestimated.

In theory, the planning-based approach has been often combined with some form of advanced scheduling approach using, e.g., a *metaheuristic* [20] to further optimize the constructed execution plan. These works were often theoretical or used a (simplified) model together with a simulator, while realistic implementations in actual resource managers were not available. Those promising results were then *rarely repeated in the practice* and most of the mainstream production systems like PBS Pro, SLURM or Moab/Maui have adopted the queuing

approach [1, 6], focusing on performance and scalability while limiting the (theoretical) benefits of increased predictability related to the planning systems. Neither metaheuristics nor other advanced optimization techniques are used in current mainstream systems.

In this paper we *bridge the gap between theory and practice* and demonstrate that planning supplied with a schedule-optimizing metaheuristic is a plausible scheduling approach that can improve the performance of an existing computing system. We describe and evaluate newly developed job scheduler (compatible with the production TORQUE Resource Manager) which supports planning and optimization. The presented scheduler has been *successfully used in practice* within a real computing infrastructure since July 2014. It constructs a preliminary job execution plan, such that an expected start time is known for every job prior its execution. This plan is further evaluated in order to identify possible inefficiencies using both *performance* as well as *fairness*-related criteria. A local search-inspired metaheuristic is used to optimize the schedule with respect to considered optimization criteria. The suitability and good performance of our solution is demonstrated in two ways. First, we present *real-life performance results* that are coming from the deployment of our scheduler in the CERIT Scientific Cloud, which is the largest partition of the Czech national grid and cloud infrastructure MetaCentrum, containing $\sim 5,200$ CPUs in 8 clusters. Second, we use several simulations to evaluate the solution against previously used queuing approach. Importantly, we have adopted the novel *workload adaptation* approach of Zakay and Feitelson [21] in order to overcome the shortcomings of "classical" simulations with static workloads.

This paper is a significantly extended and updated version of our short-paper presented at HPDC 2015 [8]. We have included more details concerning the design of the scheduler (Section 3) and significantly extended the evaluation (Section 4), adding more results from practical deployment as well as presenting a set of newly performed detailed simulations using the model of Zakay and Feitelson [21]. Theoretical background of this work has been described in our earlier work [10] which has presented new methods for efficient use of metaheuristic algorithms for multi-criteria job scheduling. However, instead of a real resource manager, this work only used a simulator with static historic workloads while considering simplified problem models [10].

This paper starts with the related work discussed in Section 2. Next, we describe the design and features of our new scheduler. Section 4 presents detailed evaluation of the scheduler's performance, while Section 5 concludes the paper.

2 Related Work

Nowadays, major production resource management systems such as PBS Pro [14], SLURM or Moab/Maui [6] use (priority) queues when scheduling jobs on available resources, applying some form of popular scheduling heuristics — typically FCFS and backfilling. On the other hand, during the past two decades many works have shown that the use of planning represents some advantages [9, 13].

Unlike the traditional "aggressive" queuing approach where scheduling decisions are taken in an ad hoc fashion often disregarding previous and future scheduling decisions, a planning-based approach allows to make plans concerning job execution. The use of such a plan (job schedule) allows to make a partial prediction of job execution, providing information concerning expected start times of jobs to the users of the system, thus improving predictability [13]. Conservative backfilling [13] represents a typical baseline solution for planning systems.

As far as we know, fully planning-based schedulers using, e.g., the Conservative backfilling algorithm, are less popular in practice than "classical" queue-based solutions [12]. Typically, only few waiting jobs are usually planned ahead, while no predictions are made for the remaining jobs. For example, PBS Professional Administrator's Guide recommends that at most 100 jobs should be planned ahead (`backfill_depth` parameter in [14]). Notable exceptions represent fully planning-based systems such as Computing Center Software (CCS) [7] and its successor Open CCS[1], that use planning and are used in practice.

Within the CERIT system, a form of Conservative backfilling has been used prior our new scheduler has been adopted. The scheduler maintained several queues with different maximal walltime limits (1h, 2h, 1d, 2d, 4d, 1w, 2w and 2m). Job queues were periodically reordered by fair-sharing algorithm. To reflect aging, resource usage records (fair-share usage) were subject to decay [6] by the aging factor of 0.5 which was applied each 72 hours, i.e., each 3 days the current user's fair-share usage was dived by 2. To avoid excessive job starvation, waiting jobs obtained a reservation when their waiting time reached a given threshold. However, by default this threshold was equal to 0 seconds implying that every waiting job obtained a reservation.

In many (theoretical) works, planning-based approach has been also used in conjunction with further optimization. Simply put, a prepared execution plan has been *evaluated* with respect to selected optimization criteria and further *optimized* using some form of a metaheuristic [20, 11, 18, 15]. As far as we know, evaluation and/or metaheuristics are not applied in nowadays production systems. In the past, Global Optimising Resource Broker (GORBA) [17] represented an experimental planning-based system designed for scheduling sequential and parallel jobs as well as workflows. It was using Hybrid General Learning and Evolutionary Algorithm and Method (HyGLEAM) optimization procedure, combining local search with the GLEAM algorithm [16] which is based on the principles of evolutionary and genetic algorithms. Sadly, this system was a proprietary solution and it seems that it is no longer operational.

3 Planning and Optimizing Job Scheduler

This section describes the new planning-based job scheduler that uses a schedule-optimizing metaheuristic. The scheduler is compatible with the TORQUE Resource Manager system, which is used in the Czech National Grid Infrastructure

[1] `https://www.openccs.eu/core/`

MetaCentrum. TORQUE is an advanced open-source product providing control over batch jobs and distributed computing resources [1]. It consists of three main entities — the server (`pbs_server`), the node daemons (`pbs_mom`) and the job scheduler (`pbs_sched`). The scheduler interacts with the server in order to allocate jobs onto available nodes. While the server and the node daemons are mostly unchanged, the default simple queue-based FCFS scheduler [1] has been replaced in this case. When using TORQUE, it is a common practice to use other than the default scheduler [1].

The new scheduler contains four major parts. The first part is the data structure that represents the *job schedule*. The schedule is *built and maintained* using schedule construction and maintenance routines that represent the default scheduling algorithm, working similarly to the well known *Conservative backfilling* [13]. Maintenance routines are used to adjust the schedule in time subject to dynamic events such as (early) job completions, machine failures, etc. Remaining parts perform the *evaluation* and the *schedule optimization*. We now closely describe these major parts in detail.

3.1 Data Representation of Job Schedule

The *schedule* is represented by a rather complex data structure that keeps all important information regarding planned job execution. In fact it consists of three separate structures. First, there is the linear list of jobs (*job_list*), where each job in the list stores information regarding its allocation, e.g., CPU/GPU IDs, the amount of RAM memory per node, amount of disk space, etc. Also the planned start time and completion time is stored for each job. Second structure (*gap_list*) stores "gaps", i.e., "unused" parts of the schedule. It is used to speed up the scheduling algorithm during the backfilling phase. Each gap is associated with its start time, duration and a list of available resources (CPUs, GPUs, RAM, HDD, etc.). Both jobs and gaps are ordered according to their expected start times. The third part of the schedule is called *limits_list* and is used to guard appropriate usage of resources. For example, it is used to guarantee per-user limits concerning maximum CPU usage. Similarly, it is used to check that a given class of jobs does not exceeds its maximum allowed allocation at any moment in the planned future (e.g., jobs running for more than a week cannot use more than 70% of all system resources).

All these structures and their parameters are kept up-to-date as the system runs using methods described in Section 3.2. For practical reasons, independent instances of *job_list*, *gap_list* and *limits_list* are created for every cluster in the system. First, such solution speeds up the computation of schedule as changes and updates are often localized to a given cluster while it also allows to simplify management of heterogeneous infrastructures, where different clusters may have different properties and usage constraints. Although the job schedule in fact consists of three major parts (*job_list*, *gap_list* and *limits_list*), for simplicity we will mostly use the term *schedule* in the following text when describing the pseudo codes of the scheduler.

3.2 Scheduling Algorithms

The job schedule is built, maintained and used according to the dynamically arriving events from the `pbs_server` using the core method called SchedulingCycle which is shown in Algorithm 1. SchedulingCycle invokes all necessary actions and auxiliary methods in order to update the schedule and perform scheduling decisions. At first, all jobs that have been completed since the previous check are removed from the *schedule* (Line 1). Next, the *schedule* is updated (Line 2) using the Update function which is described in Algorithm 2.

During the update it is checked whether existing (planned) job start times are still relevant (see Lines 1–6 in Algorithm 2). If not, those are adjusted according to the current known status. There are several reasons why planned start times may change. Commonly, jobs are finishing earlier than expected as the schedule is built using processing time estimates which are typically overestimated. Also, in some situations jobs are shifted into later time slots, e.g., due to fairness-related constraints and/or via the optimization algorithm. In both cases, jobs are checked one by one and a start time of each job is adjusted (see Line 3), i.e., it is moved into the earliest possible time slot with respect to previously adjusted jobs while respecting existing usage limits. Next, *limits_list* and *gap_list* structures are updates accordingly (see Lines 4–5 in Algorithm 2). During the update process, the relative ordering of job start times is kept, i.e., a later job cannot start earlier than some previous job. This approach is a runtime-optimized version of the *schedule compression method* used in Conservative backfilling [13].

Once the schedule is updated, all newly arrived jobs are inserted into the existing schedule (Lines 3–7 in Algorithm 1) using the Conservative backfilling-like approach. It founds the *earliest gap* in the initial *schedule* which is suitable for the new *job*. This approach is identical with the method used in Conservative backfilling for establishing job reservations [13]. It significantly increases system utilization while respecting the start times of previously added jobs. In this case, the applied data representation represents major benefit as all gaps in the current schedule are stored in a separate list (*gap_list*) which speeds up the whole search procedure. When the suitable gap is found and the job is placed into it (Line 5

Algorithm 1 SchedulingCycle

```
 1: remove finished jobs from schedule;
 2: schedule := Update(schedule);
 3: while new jobs are available do
 4:     job := get new job from pbs_server;
 5:     schedule := backfill job into the earliest suitable gap in schedule;
 6:     schedule := Update(schedule);
 7: end while
 8: notify pbs_server to run ready jobs according to schedule;
 9: if (time_current − time_previous) ≥ 60 seconds then
10:     schedule := ScheduleOptimization(schedule);
11:     time_previous := time_current;
12: end if
```

Algorithm 2 UPDATE(*schedule*)

```
1: for i := 1 to number of jobs in schedule do
2:     job := i-th job from schedule;
3:     schedule := adjust job's start time subject to limits_list;
4:     update limits_list;
5:     update gap_list;
6: end for
7: return schedule;
```

in Algorithm 1) the *schedule* is appropriately updated and another incoming job is processed.

Once all new jobs are placed into the *schedule*, the scheduler checks whether some jobs are prepared to start their execution. Those jobs are immediately scheduled for execution as depicted on Line 8 in Algorithm 1. Finally, the *schedule* is periodically optimized (see Line 10 in Algorithm 1) by a metaheuristic which we describe in the following section.

3.3 Evaluation and Metaheuristic

The real contribution of our scheduler is related to its ability to "control" itself and adjust its behavior in order to better meet optimization criteria. This is done by the periodically invoked metaheuristic optimization algorithm which is guided by the schedule evaluation. We use a simple local search-inspired metaheuristic called *Random Search (RS)* [10] and focus both on the performance- and the fairness-related criteria. We minimize the *avg. wait time* and the *avg. bounded slowdown* to improve the overall performance [4]. *User-to-user fairness* is optimized using the *Normalized User Wait Time (NUWT)* metric [10]. For a given user, NUWT is the total user wait time divided by the amount of previously consumed system resources by that user. Then, the user-to-user fairness is optimized by minimizing the mean and the standard deviation of all NUWT values. It follows the classical fair-share principles, i.e., a user with lower resource usage and/or higher total wait time gets higher priority over more active users and vice versa [6]. The calculation of NUWT reflects consumptions of multiple resources (CPU and RAM utilization), representing a solution suitable for systems having heterogeneous workloads and/or infrastructures [6].

The *Random Search (RS)* optimization algorithm is implemented in the SCHEDULEOPTIMIZATION function (see Algorithm 3) that uses one input — the schedule that will be optimized. In each iteration, one random job from the schedule is selected and it is removed from its current position (Lines 3–4). Next, this job is returned to the schedule on a randomly chosen position (Line 5) and the new schedule is immediately updated (Line 6). The modified schedule is evaluated with respect to applied optimization criteria. This multi-criteria evaluation is performed using a simple weight function[2] that has been successfully used

[2] Our system uses equal weights ($w = 1$) for wait time and bounded slowdown while the normalized user wait time (fairness) has ten times higher weight ($w = 10$).

Algorithm 3 ScheduleOptimization(*schedule*)

```
 1: schedule_best := schedule;
 2: while not interrupted do
 3:    job := select random job from schedule;
 4:    remove job from schedule;
 5:    move job into random position in schedule;
 6:    schedule := Update(schedule);
 7:    if schedule is better than schedule_best then
 8:       schedule_best := schedule;
 9:    end if
10:    schedule := schedule_best;     (reset candidate)
11: end while
12: return schedule_best;
```

in our previous works [10, 9]. If the new *schedule* is better than the best so far found $schedule_{best}$ then the $schedule_{best}$ is updated with this new, better *schedule*. Otherwise, the $schedule_{best}$ remains unchanged (Lines 7–9). Then the *schedule* is updated/reset with the $schedule_{best}$ (Line 10) and a new iteration starts. Once the loop ends, the newly found $schedule_{best}$ is returned (Line 12).

The metaheuristic is fully randomized and does not employ any "advanced" search strategy. In fact, during the design process we have observed that this simple randomized optimization is very robust and produces good results, often beating more advanced methods such as Simulated Annealing or Tabu Search. The beauty of RS is that it is simple (i.e., fast) and — unlike, e.g., Simulated Annealing — its performance does not rely on additional (hand-tuned) parameters.

Certainly, optimization is a potentially time consuming operation. Therefore, the optimization is only executed if the last optimization ended at least 60 seconds ago (see Lines 9–12 in Algorithm 1). This interval has been chosen experimentally in order to avoid frequent — thus time consuming — invocations of the ScheduleOptimization function. Furthermore, several parameters are used when deciding whether to interrupt the main loop of the optimization procedure or not (Line 2 in Algorithm 3). We use the maximal number of iterations and the given time limit. Currently, the time limit is 20 seconds and the number of iterations is set to 300 in our system.

3.4 User Perspective: System Interfaces

From the user perspective, the newly developed scheduler does not introduce any major difference with respect to other standard schedulers. It uses the same syntax of the `qsub` command as a "normal" TORQUE-based system, so users can use the same commands as they are used to from different systems. The TORQUE's `pbs_server` have been slightly extended, such that it can read job-related data from the schedule and then provide them to the users. For this purpose, the `pbs_server` queries the schedule and then displays the informa-

tion obtained, including currently planned start time and execution node(s). We support both textual (`qstat` command) and graphical user interfaces using a complex web application called *PBSMon*[3] which monitors the whole infrastructure and workload.

4 Evaluation and Deployment

The developed scheduler and its optimization metaheuristic has been thoroughly tested using various methodologies. In this section we present the results of three different evaluation scenarios. First, Section 4.1 shows the comparison of system performance before and after the new scheduler has been deployed in practice. These results represent the actual behavior of the system, but include one major but unavoidable drawback — the comparison is not based on the same workload, since the results were obtained from a real system in two different consecutive time periods. This problem can be avoided by testing the new scheduler (and its predecessor) using a computer testbed, where the same set of jobs is submitted to both schedulers and their resulting performance is compared. Although this approach is quite realistic, such a comparison is very time consuming (see discussion in Section 4.2), limiting the "size" of the data sets that can be used. Therefore, we also include a third type of evaluation, where the major features of both the original and the newly proposed scheduler have been implemented within a job scheduling simulator and a large data set from the actual system has been used. This comparison is presented in Section 4.3.

In all cases, the proposed planning-based scheduler using Random Search metaheuristic (denoted as *Plan-RS*) has been evaluated against the backfilling-based algorithm (denoted as *Orig-BF*) that was originally applied in the system (see Section 2). Additional algorithms were not considered either because their implementations within TORQUE were not available or their performance was very poor (e.g., plain FCFS without backfilling). All experiments used the original inaccurate runtime estimates.

4.1 Real-Life Deployment

First, we present real-life data that were collected in the Czech Centre for Education, Research and Innovation in ICT (*CERIT Scientific Cloud*)[3], where our new scheduler has been operationally used since July 2014. *CERIT Scientific Cloud* provides computational and storage capacities for scientific purposes and shares them with the Czech National Grid and Cloud Infrastructure *MetaCentrum*. Both MetaCentrum and CERIT use the same version of TORQUE resource manager. Before July 2014, CERIT was using the same scheduler (Orig-BF) as MetaCentrum. CERIT consists of 8 computer clusters with $\sim 5,200$ CPU cores that are managed by our new scheduler (Plan-RS) since July 2014.

Following comparative examples are based on the historic workload data that were collected when either the original Orig-BF scheduler or the new Plan-RS

[3] `http://metavo.metacentrum.cz/pbsmon2/`

scheduler were used respectively. In the former case (Orig-BF), the data come from the January – June 2014 period. In the latter case the data are related to the new scheduler (Plan-RS) and cover the July – December 2014 period[4].

The first example in Figure 1 (left) focuses on the average system CPU utilization. It was observed that — on average — the new scheduler was able to use additional 10,000 CPU hours per day compared to the previous scheduler. This represents 418 fully used CPUs that would otherwise remain idle and causes that the avg. CPU utilization has increased by 9.3%.

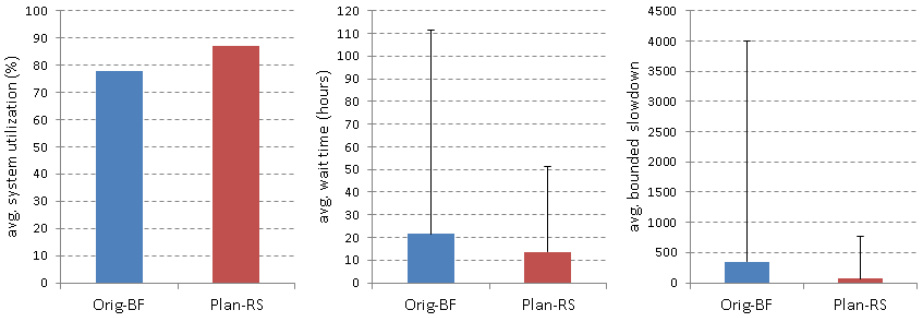

Fig. 1. Real-life comparisons showing (from the left to right) the avg. system utilization, the avg. wait time and the avg. bounded slowdown.

Although the increased utilization is beneficial, it may come at the cost of decreased performance for selected classes of jobs, which is a known feature [13]. As users tend to watch how the system is processing their jobs, improved utilization, i.e., higher throughput, may cause that users will send more jobs into the system. As the total available computing power is limited, these "additional jobs" may have to wait longer until resources become available. Moreover, reservations established by backfilling often represent a pessimistic scenario as jobs are typically completing earlier than their estimates suggest [19]. Even though existing reservations are shifted to those appearing free time slots (see the discussion on schedule compression in Section 3.2), short/narrow jobs would still have a higher chance to fill these gaps compared to long and/or highly parallel jobs. Therefore, we have performed further analysis of the data focusing on additional performance indicators.

First, we have compared the avg. wait time and the avg. bounded slowdown as well as their standard deviations for the two considered schedulers. The results are shown in Figure 1 (2nd and 3rd chart from the left, respectively), where the "error bar" depicts the standard deviation of the metric. As we have observed,

[4] Those two periods were chosen because the physical infrastructure was identical during that time. Since January 2015, the system became larger (4,512 CPUs vs 5,216 CPUs) which would skew any direct comparison of system performance.

the original Orig-BF scheduler often produced very bad wait times and slowdowns for many jobs, causing high average values and large deviations. From this point of view, Plan-RS was much more efficient, significantly decreasing both the averages and the deviations. Given the increased utilization observed in Figure 1 (left) this is a good news.

Furthermore, we have also analyzed the average job wait time with respect to job parallelism (number of requested CPUs) as shown in Figure 2 (left). The results for Plan-RS are again promising as most job classes now have better average wait times compared to the former Orig-BF scheduler, i.e., Plan-RS is not causing significant delays for (highly) parallel jobs.

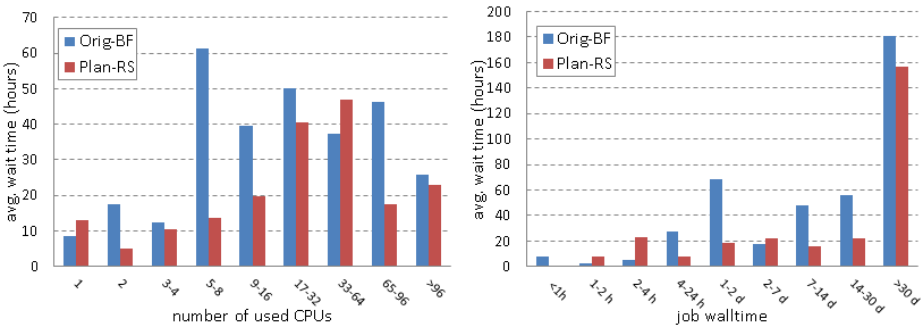

Fig. 2. Real-life comparison of job wait times with respect to job parallelism (left) and job walltime (right).

Also, job walltime (processing time estimate) is an important factor that has some influence on job's chances to obtain a good (early) reservation. Longer jobs are less likely to obtain early reservations, i.e., their wait times may be (very) large in some cases. Therefore we have compared average wait times of jobs with respect to their walltime estimates as were specified by users. As can be seen in Figure 2 (right), there are no significant side effects associated with the use of Plan-RS. Importantly, with a single exception (2-7 days), the average wait time of jobs that have their runtime estimate larger than 4 hours was always smaller compared to the former Orig-BF scheduler. Furthermore, such jobs represent nearly 83% of the whole workload, i.e., they are very frequent, yet they are not significantly delayed by the new scheduler which is very important. To sum up, our new Plan-RS scheduler has increased the utilization in CERIT system, without producing any significant undesirable side effect. In fact, also the avg. wait time and the avg. bounded slowdown have been significantly reduced compared to the original Orig-BF scheduler.

As discussed in Section 3.3, user-to-user fairness is maintained by minimizing the mean and the standard deviation of Normalized User Wait Times (NUWT). Figure 3 shows the fairness-related results for both schedulers. The mean and the standard deviation (shown by error bar) of NUWT values are very close for Orig-

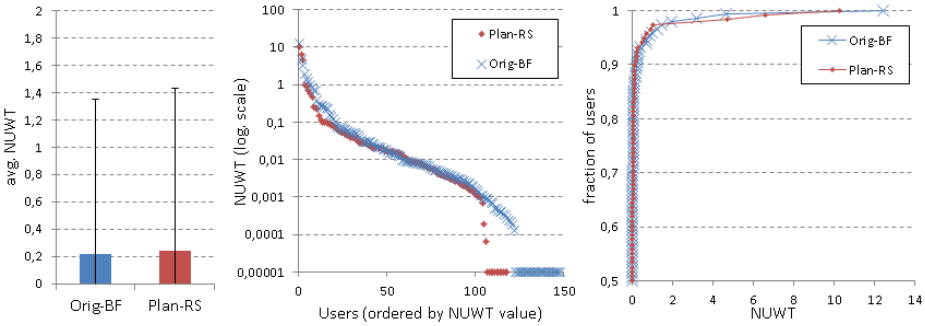

Fig. 3. Fairness-related results showing the avg. Normalized User Wait Time (NUWT) and its (per user) distribution as well as corresponding CDF.

BF and Plan-RS (see Figure 3 (left)). More detailed results are shown in Figure 3 (middle and right), showing the NUWT values per user and the corresponding cummulative distribution function (CDF) of NUWT values, respectively. The results for both schedulers are quite similar — most users ($\sim$ 97%) have their NUWT below 1.0, meaning that they spent more time by computing than by waiting which is beneficial and indicate that both Plan-RS and Orig-BF are capable to maintain reasonable fairness level.

In order to demonstrate the capability of our metaheuristic to improve the quality of the schedule in time we have also recorded all successful optimization attempts during the October – December 2014 period. Then, we have plotted the corresponding relative improvements (and deteriorations) of those criteria with respect to the time. Figure 4 shows the results for wait time and fairness criteria respectively. Commonly, the main reason that an attempt was accepted is that the user-to-user fairness was improved. This is an expected behavior. In the CERIT system, user-to-user fairness is not directly guaranteed by the underlying Conservative backfilling-like algorithm, and it can only be improved through the optimization. Without optimization, the only way to assert fairness is to periodically re-compute the schedule from scratch, i.e., reinsert all waiting jobs into the schedule following a new job ordering computed according to updated user priorities. This is potentially very time consuming, thus non-preferred option. Therefore, it is very common for the optimization algorithm to find a schedule with improved fairness. Figure 4 also reveals that the majority of accepted optimization attempts represents rather decent improvements, where the relative improvement of a given criterion is less than 2% in most cases. Still, several large improvements can be seen for both criteria during the time (and few more are not shown since the y-axis is cropped for better visibility). These rarer attempts are very important as they help to reduce those few extremely inefficient assignments that can be seen in nearly every production workload. As the optimization is continuously evaluating the schedule, it is able to detect jobs having (very) high wait times, slowdowns, etc. Then, it can develop better

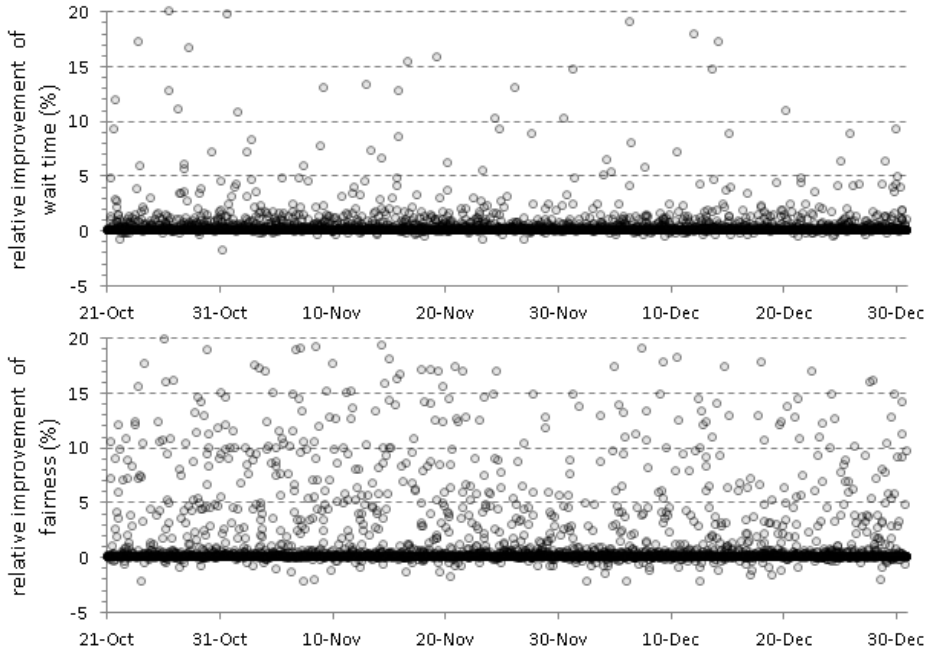

Fig. 4. Successful optimization attempts over the time.

schedules where these extremes are reduced. These results help to explain the large improvement of wait times and slowdowns observed in Figure 1.

We were also careful about the runtime requirements of our rather complex solution. Therefore, we have measured how the size of the schedule (number of jobs) affects the runtime of critical schedule-maintaining routines. For this analysis, the runtime of the *backfilling-like* policy was measured as well as the time needed to perform the subsequent *schedule-update* routine which updates the schedule-related data structures subject to modifications (see Section 3.2). Also the total runtime (backfilling + update) and the average runtime of one iteration of Random Search metaheuristic were recorded. The results are presented in Figure 5, where the y-axis shows consumed runtime (in microseconds) and the x-axis depicts the number of jobs in the schedule.

The results show that there is no simple correlation between the overall size of the schedule and algorithm runtime. This is a natural behavior caused by several factors. First of all, jobs being added to the schedule have different requirements — some jobs are generally very flexible, i.e., they can be executed on several clusters while other jobs can only use a small subset of system's clusters. Then the algorithm runtime may vary significantly depending whether one, two, or more cluster schedules must be analyzed for a given job. Since we use backfilling, if a suitable gap is found in an early time slot (close to the beginning of the schedule), the runtime is lower as we do not have to traverse the whole

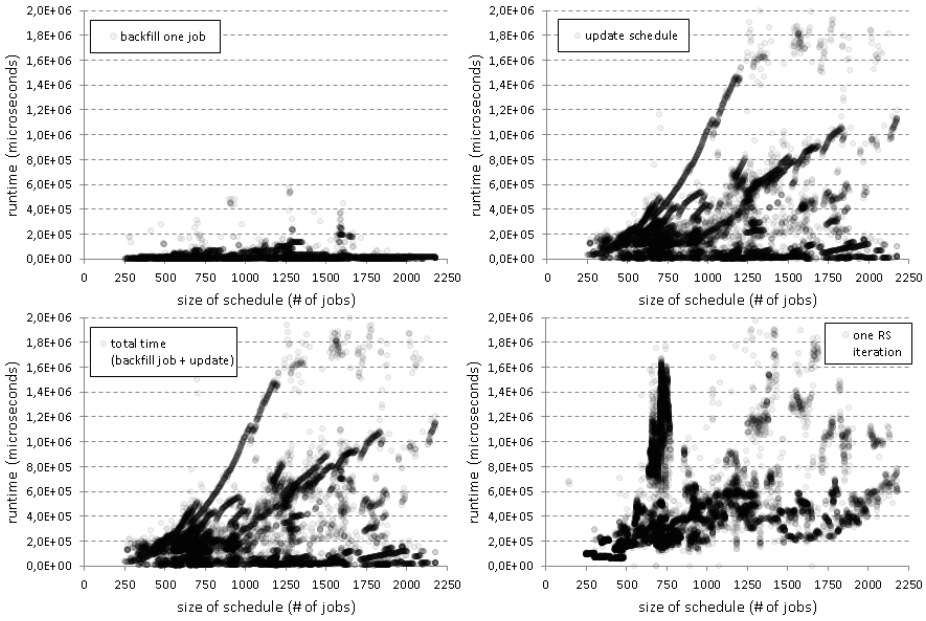

Fig. 5. Runtime requirements with respect to the number of jobs in the schedule.

schedule, and vice versa. Moreover, the physical system consists of 8 different clusters that have different types of nodes and amounts of system resources and each such cluster has its own *schedule* instance. Naturally, *schedule* for larger cluster requires more runtime to be backfilled/updated. Still, some basic trends are visible in the figures, such as approximately linear upper bound of requested runtime. With the current typical backlog of CERIT system — where the number of jobs in the schedule is usually bellow 2,200 — 61% of jobs require less than 0.2 seconds to be placed in the schedule (backfill + update), most jobs (96%) then require less than 1 second while 99% of jobs fit within 2 seconds. Clearly, schedule construction does require nontrivial time, however we usually have $< 1,000$ new job arrivals per day which is well within the current capacity of our implementation and we have not observed any delays/overheads so far.

The avg. runtime of one iteration of Random Search (RS) may be higher than of previous routines (see the bottom right part of Figure 5), which is natural. First of all, the evaluation of the whole schedule requires some time. Second, when an iteration is not improving, the schedule must be reset to its previous state which requires an additional update. Therefore, the runtime of one RS iteration is often at least twice as high as the corresponding runtime of the update procedure. Finally, while the backfill + update part of the scheduler is only executed upon new job arrival, RS is executed periodically (~ 60 seconds). If there is a "complicated schedule" at that time, the chart of RS runtime will show a large peak, as this runtime-demanding schedule is repeatedly updated.

An example of such situation is visible in the chart, showing a rather large peak of runtime for schedules having $\sim$ 700 jobs. This particular peak was caused by a specific situation on one of the cluster's schedules where a set of similar jobs — all belonging to a single user — have remained for a couple of days. It was basically impossible to optimize the schedule for such jobs, and frequent time-demanding updates (schedule resets) were inevitable, producing a runtime peak clearly visible as a clump of dots in the chart.

4.2 Comparison Using Testbed

Another possible way how to properly compare the proposed solution is to test both the former Orig-BF and the new Plan-RS schedulers using a simulation testbed, feeding both schedulers with identical workloads. The problem is that such experiments are very time consuming. One cannot simply use a long, realistic workload "as is" because the experiment would last for several weeks/months depending on the original length of the workload. Instead, only several "promising" job intervals from a workload can be extracted. We have chosen those with a significant activity and contention (using the number of waiting jobs at the given time as a metric). Based on this information we have extracted those promising intervals that lasted for at least 5 days. Such intervals were more likely to show differences between the two schedulers. Furthermore, all job runtimes and all corresponding inter-arrival times between two consecutive jobs were divided by a factor of 7. The resulting workload was then proportional to the original one, exhibiting similar behavior but having 7-times shorter duration (makespan), making the simulation possible within a reasonable time frame. For example, if the original data covered one week then it took only 1 day to perform the whole simulation. Eight such sub-workloads were then used — four of them based on data from the CERIT's Zewura cluster, two from the HPC2N log and the two remaining came from the KTH-SP2 log. Detailed analysis and further description of this experiment have been already presented in [8], therefore we only briefly recapitulate that the proposed Plan-RS scheduler dominated over Orig-BF in all cases. Table 1 shows the relative decrease of the avg. wait time (WT) and the avg. bounded slowdown (SD) achieved by the Plan-RS scheduler with respect to the Orig-BF scheduler[5].

4.3 Comparison Using Simulator

As explained in Sections 4.1-4.2, evaluation based on practical deployment as well as testbed-based comparison are somehow problematic (different workloads and time-related constraints, respectively). Therefore, we have also used the *Alea* job scheduling simulator [2] where both Orig-BF and Plan-RS have been emulated. Moreover, instead of using the classical static approach where a given workload is

[5] The small size and short makespan of these experiments meant that there were few distinctive users in the workload — most of them with just few jobs — making the use of the fairness-related criterion rather impractical and inconclusive in this case.

Table 1. Achieved relative decrease (in %) of the avg. wait time (WT) and the avg. bounded slowdown (SD) when using the new Plan-RS scheduler.

	Zewura set 1	Zewura set 2	Zewura set 3	Zewura set 4	HPC2N set 1	HPC2N set 2	KTH-SP2 set 1	KTH-SP2 set 2
WT	-18.8%	-40.0%	-57.2%	-41.1%	-81.0%	-26.6%	-31.6%	-7.2%
SD	-32.6%	-49.7%	-84.7%	-39.3%	-89.6%	-42.0%	-64.0%	-45.7%

"replayed" in the simulator, we have adopted the recently proposed dynamic approach of Zakay and Feitelson [21], where job submission times are not dictated by the workload but are the result of the (simulated) scheduler-to-user interaction. As explained in [21], job submission times in a real system depend on how users react to the performance of previous jobs. Moreover, usually there are some logical structures of dependencies between jobs. It is therefore not reasonable to use a workload "as is" with fixed (original) job submission timestamps, as the subsequent simulation may produce unrealistic scenarios with either too low or too high load of the system, skewing the final results significantly. Instead, dependency information and user behavior can be extracted from a workload trace, in terms of job batches, user sessions and think times between the completion of one batch and the submission of a subsequent batch. Then, each user's workload is divided into a sequence of dependent batches. During the simulation, these dependencies are preserved, and a new user's batch is submitted only when all its dependencies are satisfied (previous "parent" batches are completed). This creates the desired feedback effect, as users dynamically react to the actual performance of the system, while major characteristics of the workload including job properties or per-user job ordering are still preserved. More details can be found in [21, 22] while the actual implementation of the model (using user agents instead of standard workload reader) is available within the Alea simulator [2].

We have used a workload trace from the CERIT system that covered 102,657 jobs computed during January – April 2015[6]. Again, we have compared the "historical" Orig-BF with the newly proposed Plan-RS scheduler. All experiments using Plan-RS have been repeated 20 times (and their results averaged) since RS is not deterministic and uses a randomized approach. The results for the avg. wait time and the avg. bounded slowdown are shown in Figure 6, error bars in the left chart shows the standard deviation of the 20 runs of Plan-RS. As previously (see Sections 4.1-4.2), Plan-RS decreases significantly the wait time and the bounded slowdown. The explanation is quite the same as was in Section 4.1 and can be nicely demonstrated on the CDF of job wait times which we show in Figure 6 (right). The mean wait time for Plan-RS is 1.6 hours, while the CDFs for both scheduler show that 85% of jobs wait shorter than 1.6 hours. This can

[6] This workload is available at: `http://www.fi.muni.cz/~xklusac/workload/`

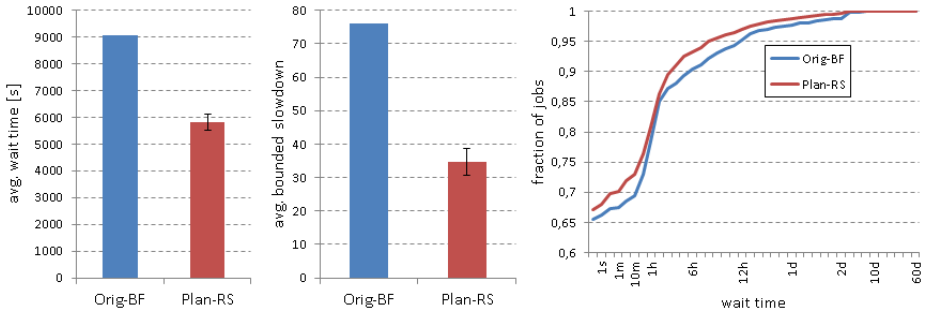

Fig. 6. Performance-related results showing (from left to right) the avg. wait time, the avg. bounded slowdown and the CDF of job wait times.

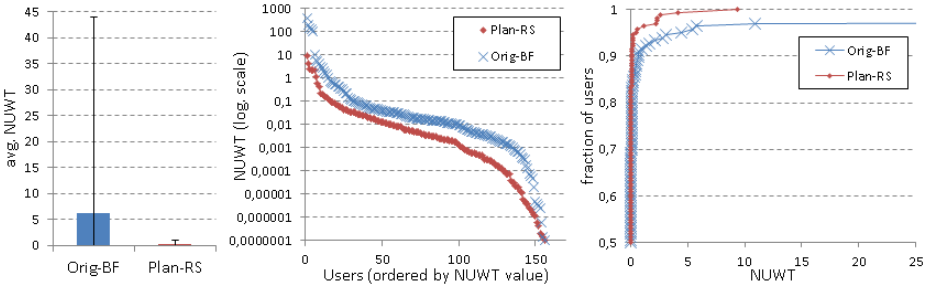

Fig. 7. Fairness-related results showing (from left to right) the mean Normalized User Wait Time (NUWT), NUWT histogram wrt. users and corresponding CDF.

only mean — and it is clearly visible in the CDF — that Plan-RS decreases some of those excessive wait times of the remaining 15% of jobs.

Concerning the fairness, the Plan-RS performed much better than Orig-BF as shown in Figure 7. The mean and the corresponding standard deviation of NUWT values were significantly lower compared to Orig-BF. When analyzed on a detailed per-user basis (see the middle and the right chart in Figure 7), the results clearly show that Plan-RS decreases NUWT across the whole user base.

In the final experiment, we have developed a new *experimental model to measure user (dis)satisfaction* with the system performance. Here we were inspired by the future work discussed in the recent Zakay and Feitelson paper [22], which suggest that (in reality) users may leave if the performance is too poor. In our case a user agent does not leave the system, instead it "reports" that it is not satisfied with the current waiting time. Also, it measures "how large" this dissatisfaction is by calculating the actual to expected wait time ratio. In our simple model, a user agent expects that the system shall start its jobs in a time which is proportional to job's requirements. In other words, the longer a job is (higher walltime estimate) and the more CPUs it requires the higher is the tolerable wait time and vice versa. However, this dependence is not linear, since our experience

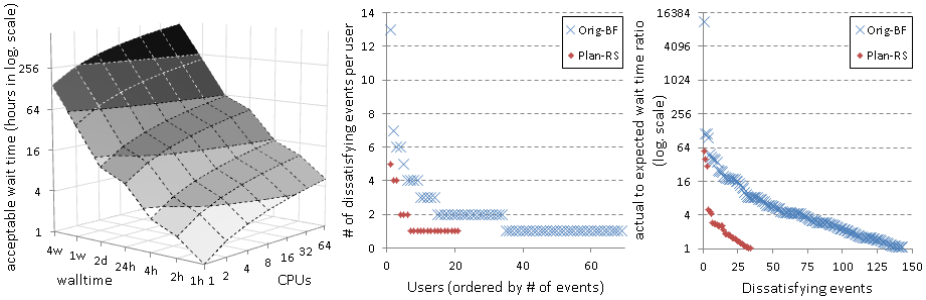

Fig. 8. Acceptable wait time with respect to CPU and walltime requirments (left), the number of dissatisfying events per user (middle) and all dissatisfying events ordered by their seriousness (right).

shows that real users usually have some "upper bound" of their patience. For example, if a job requiring 1 CPU starts within an hour, then users are usually satisfied. However, that does not imply that a job requiring 64 CPUs can wait for 64 hours. We have similar experience concerning walltime, i.e., user's patience is not linear with respect to job duration, instead it quickly runs out. Therefore, we have developed a simple formula to calculate "acceptable wait time" for a given job which captures this nonlinearity[7]. Figure 8 (left) shows the non-linear distribution of acceptable wait times with respect to job durations and their parallelism, as produced by the applied formula. Of course, this simple "hand tuned" formula is just a rough approximation used for demonstration purposes and it certainly does not represent a truly realistic model of user's expectations.

During a simulation, *job's acceptable wait time* is calculated upon each new job arrival. If more jobs of a single user are present in the system we sum up their acceptable wait times. Then, whenever a job is started, a corresponding user agent checks whether the actual wait time was within the calculated overall limit. If not, a user agent "reports dissatisfaction" and calculates the level of such dissatisfaction, which is the actual wait time divided by the acceptable wait time. Further details can be found in the `AgentDynamicWithSatisfactionModel` class of the simulator [2]. The results of such experiment are shown in Figure 8 (middle) and (left), showing the number of dissatisfying events per user and all dissatisfying events ordered by their seriousness, respectively. It shows superior performance of Plan-RS, which is able to significantly minimize the number of "complaining users", the number of excessively waiting jobs, as well as the "size", i.e., seriousness of such job delays.

[7] The formula is: $acceptable\_wait = (\ln(req\_CPUs) + 1) \cdot (walltime/factor)$. req_CPUs denotes the number of requested CPUs and job's $walltime$ is divided by an integer ($factor \geq 1$) which increases as the walltime increases, emulating the non-linear user's wait time expectations. Currently, we use five factors $1, 2, .., 5$, which apply for walltimes $< 3h$, $3h..7h$, $7h..24h$, $1d..7d$, $\geq 1w$, respectively.

5 Conclusion

In this paper we have provided a detailed analysis of the real production scheduler which uses *planning* and *metaheuristic-based schedule optimization*. Using various types of evaluation we have demonstrated that both planning and some form of optimizing metaheuristic can be used in practice. In reality, the planning feature is useful for users as well as for system administrators. On several occasions, the constructed plan revealed problems long before someone else would normally notice (e.g., suboptimal job specification leading to very large planned start time). Also, system administrators often use prepared plan when reconsidering various system-wide setups, e.g., too strict limits concerning resource usage. Certainly, this approach is not suitable for every system. Not surprisingly, planning (in general) is more time-consuming approach compared to plain queuing. The time needed for construction and maintenance of job schedules grows with the size and complexity of the system and its workload (see Section 4.1). Surely, our current implementation can be further improved. For example, all schedule-related routines are currently sequential — running in a single thread — and can be relatively easily parallelized. So far, this is not an issue within CERIT system and no problems concerning scalability/speed were recorded so far. Our scheduler is freely available at: `https://github.com/CESNET/TorquePlanSched`.

Acknowledgments. We kindly acknowledge the support and computational resources provided by the MetaCentrum under the program LM2015042 and the CERIT Scientific Cloud under the program LM2015085, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations". We also highly appreciate the access to CERIT Scientific Cloud workload traces. Last but not least, we thank Dror Feitelson for his kind help and explanation concerning the dynamic workload model presented in [21].

References

1. Adaptive Computing Enterprises, Inc. *Torque 6.0.0 Administrator Guide*, February 2016. `http://docs.adaptivecomputing.com`.
2. Alea simulator, February 2016. `https://github.com/aleasimulator`.
3. CERIT Scientific Cloud, February 2016. `http://www.cerit-sc.cz`.
4. D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *LNCS*, pages 1–34. Springer Verlag, 1997.
5. M. Hovestadt, O. Kao, A. Keller, and A. Streit. Scheduling in HPC resource management systems: Queuing vs. planning. In *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 1–20. Springer, 2003.
6. D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In D. G. Feitelson and L. Rudolph, editors, *Job Sched. Strategies for Paral. Proc.*, volume 2221 of *LNCS*, pages 87–102. Springer, 2001.

7. A. Keller and A. Reinefeld. Anatomy of a resource management system for HPC clusters. *Annual Review of Scalable Computing*, 3:1–31, 2001.
8. D. Klusáček, V. Chlumský, and H. Rudová. Planning and optimization in TORQUE resource manager. In *24th ACM International Symposium on High Performance Distributed Computing (HPDC)*, pages 203–206. ACM, 2015.
9. D. Klusáček and H. Rudová. Performance and fairness for users in parallel job scheduling. In W. Cirne, editor, *Job Scheduling Strategies for Parallel Processing*, volume 7698 of *LNCS*, pages 235–252. Springer, 2012.
10. D. Klusáček and H. Rudová. A metaheuristic for optimizing the performance and the fairness in job scheduling systems. In Y. Laalaoui and N. Bouguila, editors, *Artificial-Intelligence Applications in Information and Communication Technologies*, volume 607 of *Studies in Comput. Intelligence*, pages 3–29. Springer, 2015.
11. J. Koodziej and F. Xhafa. Integration of task abortion and security requirements in GA-based meta-heuristics for independent batch grid scheduling. *Computers & Mathematics with Applications*, 63(2):350 – 364, 2012.
12. B. Li and D. Zhao. Performance impact of advance reservations from the Grid on backfill algorithms. In *Sixth International Conference on Grid and Cooperative Computing (GCC 2007)*, pages 456 –461, 2007.
13. A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.
14. PBS Works. *PBS Professional 13.0, Administrator's Guide*, February 2016. `http://www.pbsworks.com`.
15. Z. Pooranian, M. Shojafar, J. Abawajy, and A. Abraham. An efficient meta-heuristic algorithm for grid computing. *Journal of Combinatorial Optimization*, pages 1–22, 2013.
16. K.-U. Stucky, W. Jakob, A. Quinte, and W. Süß. Solving scheduling problems in Grid resource management using an evolutionary algorithm. In *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1252–1262. Springer, 2006.
17. W. Süß, W. Jakob, A. Quinte, and K.-U. Stucky. GORBA: A global optimising resource broker embedded in a Grid resource management system. In *International Conference on Parallel and Distributed Computing Systems, PDCS 2005*, pages 19–24. IASTED/ACTA Press, 2005.
18. P. Switalski and F. Seredynski. Scheduling parallel batch jobs in grids with evolutionary metaheuristics. *Journal of Scheduling*, pages 1–13, 2014.
19. D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789 –803, 2007.
20. F. Xhafa and A. Abraham. *Metaheuristics for Scheduling in Distributed Computing Environments*, volume 146 of *Studies in Computational Intelligence*. Springer, 2008.
21. N. Zakay and D. G. Feitelson. Preserving user behavior characteristics in trace-based simulation of parallel job scheduling. In *22nd Modeling, Anal. & Simulation of Comput. & Telecomm. Syst. (MASCOTS)*, pages 51–60, 2014.
22. N. Zakay and D. G. Feitelson. Semi-open trace based simulation for reliable evaluation of job throughput and user productivity. In *7th IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom 2015)*, pages 413–421. IEEE, 2015.