One can only gain by replacing EASY Backfilling: A simple scheduling policies case study

Danilo Carastan-Santos, Raphael y de Camargo, Denis Trystram, Salah Zrigui

One can only gain by replacing EASY Backfilling: A simple scheduling policies case study

Danilo Carastan-Santos*
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
danilo.santos@ufabc.edu.br

Raphael Y. de Camargo*
Center of Mathematics, Computation and Cognition
Federal University of ABC
Santo André, Brazil
raphael.camargo@ufabc.edu.br

Denis Trystram*
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
trystram@imag.fr

Salah Zrigui*
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
salah.zrigui@univ-grenoble-alpes.fr

***Abstract*—High-Performance Computing (HPC) platforms are growing in size and complexity. In order to improve the quality of service of such platforms, researchers are devoting a great amount of effort to devise algorithms and techniques to improve different aspects of performance such as energy consumption, total usage of the platform, and fairness between users. In spite of this, system administrators are always reluctant to deploy state of the art scheduling methods and most of them revert to EASY-backfilling, also known as EASY-FCFS (EASY-First-Come-First-Served). Newer methods frequently are complex and obscure and the simplicity and transparency of EASY are too important to sacrifice.**

In this work, we used execution logs from five HPC platforms to compare four simple scheduling policies: FCFS, Shortest estimated Processing time First (SPF), Smallest Requested Resources First (SQF), and Smallest estimated Area First (SAF). Using simulations, we performed a thorough analysis of the cumulative results for up to 180 weeks and considered three scheduling objectives: waiting time, slowdown and per-processor slowdown. We also evaluated other effects, such as the relationship between job size and slowdown, the distribution of slowdown values, and the number of backfilled jobs, for each HPC platform and scheduling policy.

We conclude that one can only gain by replacing EASY-backfilling with SAF with backfilling, as it offers improvements in performance by up to 80% in the slowdown metric while maintaining the simplicity and the transparency of FCFS. Moreover, SAF reduces the number of jobs with large slowdowns and the inclusion of a simple thresholding mechanism guarantees that no starvation occurs. Finally, we propose SAF as a new benchmark for future scheduling studies.

***Index Terms*—High Performance Computing, Online Scheduling, EASY, Backfilling, SAF**

I. Introduction

It is well known that High Performance Computing (HPC) is becoming a requirement in order to solve the arising complex problems that come from many fields of science and industry (health, climate, economics, *etc*). The ever increasing demand of computing power has led to the construction of extreme-scale, parallel and distributed computing platforms, with an impressive fast evolution of computing power, as it can be seen in the Top500 [1] supercomputer ranks. In order to keep this fast evolution, it is necessary to solve many scientific and technical problems that arise from many aspects of HPC, ranging from hardware architecture, to resource management and applications.

The resource management aspect plays a key role in the performance of HPC platforms: it is the component that assigns when and where the applications will be executed in such platform. In this regard, a common practice of HPC platform administrators is to deploy a Resources and Jobs Management System (RJMS) to perform the resource management. In a standard scenario, HPC applications (here called as *jobs*) arrive in the RJMS to be executed in the HPC platform. These jobs arrive in an unpredictable (on-line) manner in the RJMS queue and one of the challenges of the RJMS is to assign a priority order of the jobs in the queue, in order to satisfy one or many performance metrics.

This priority assignment problem is a nontrivial task and it is known in the literature as the Parallel On-line Job Scheduling Problem [2]. While there is indeed many works that propose clever approaches to solve this problem (see Section II), yet there is a noticeable distance between theory and practice [3], as most of the HPC platform administrators opt to use simple scheduling heuristics, with the Aggressive Backfilling with First-Come-First-Served order (also called EASY Backfilling [4]) being by far the most popular heuristic.

Many reasons can be devised to justify the choice of EASY Backfilling: it is established that EASY Backfilling increases the overall utilization of the platform, while keeping a relative simplicity and job starvation guarantees. Furthermore, although it is also established that there is room for improvement in the scheduling, replacing EASY Backfilling with another algorithm might be seen as a risky change: one can see this change as a "jump into the dark", with the changes in performance only noticeable after a long period of time, and potentially after many strong-worded emails from many

*Authors' names are sorted alphabetical order

(important) users.

This work goes towards bringing light to this jump. We selected a class of scheduling algorithms that keep the same simplicity and starvation guarantees of EASY Backfilling and we used a fast and reliable HPC simulation software to provide sound evidence on what could be gained – considering many relevant performance metrics – if one replaces EASY Backfilling. More specifically, this paper presents the following contributions:

- We present an experimental study that addresses the expectations and potential gains that come from replacing the EASY Backfilling scheduling policy in typical high-performance computing platforms;
- We highlight the Shortest Area First (SAF) scheduling policy, which, we argue, has the best-observed overall performance among the tested policies. In fact, we propose SAF as a new benchmark for future batch scheduling studies;
- We highlight an aspect that is often overlooked when evaluating the performance of a scheduling policy, which is the link between the number of resources used by jobs and the fairness of a given scheduling policy;
- We address the influence of the aggressive backfilling mechanism on the transparency and predictability of scheduling algorithms.

The remainder of this paper is organized as follows. In Section II we present some closely related works, while in Sections III and IV we explain the scheduling problem under study and the performed experimental protocol. In Section V we present and discuss the obtained experimental results. Finally, we summarize the main conclusions of the paper and present future works in Section VI.

II. Related Work

Due to the several classes of scheduling problems and their different levels of complexity, many works have been proposed to solve them using a wide range of approaches, ranging from integer linear programming [5], [6] to genetic algorithms [7], [8] and neural networks [9]. Xhafa and Abraham [10] present an overall review of scheduling algorithms, mainly focused on HPC platforms.

In the literature, the parallel job scheduling problem is, in its majority, studied under the view of a more general and closely related problem called multiple-strip packing problem [11] in which, informally, aims to find a packing configuration of rectangles (jobs) into a set of strips (set of processors) in order to minimize the maximum height among the used strips. It is known that the single strip case is NP-hard [11] and, thus, many approximation algorithms and performance bounds were proposed [12]–[16]. One notable characteristic of these works is that most of them are concerned to minimize objectives that are arguably analytically easier to be treated such as makespan (the largest completion time among the jobs).

A lot of effort has also been applied in list scheduling [17] based algorithms, whose its core relies on queue ordering policies. There is a vast number of queue ordering policies conceived, from hand-engineered [18] to tuned or machine-learned [19]–[22] policies. Although almost all queue ordering policies are easier to understand, it is well known [3] that most RJMSs deploy the First-Come-First-Served (FCFS) policy with some backfilling mechanism [4], and optionally with an arbitrary job prioritization, represented by multi-queue priorities [23].

This unwillingness to apply different policies other than FCFS is arguably due to the lack of clarity and interpretability of these policies, making the whole scheduling algorithm less transparent to the users. In this paper we go towards resolving this unwillingness, by arguing and showing that there exist other policies that are equally simple and clear as FCFS – notably the Shortest Processing Time First and Shortest Area First (SPF and SAF, see Section V) – that can provide significant performance improvements in comparison to FCFS under many different performance objectives, and they only need an equally simple threshold mechanism to provide the same job starvation guarantees as FCFS. We also argue that these two policies (SPF and SAF) with threshold should be considered as new baselines of performance for future online parallel job scheduling research.

III. Online Batch Scheduling Problem

A. Preliminary Definitions

In this work, we consider an HPC platform as constituted by a set of m homogeneous processing resources connected by a interconnection topology. In the on-line setting, parallel and rigid (i.e. fixed and known in advance number of required resources) *jobs* arrive in a centralized waiting queue at any moment in time. For each job t, we consider the following characteristics:

- The actual processing time p_t of the job (only known after the job has been executed);
- The estimated processing time $\tilde{p}_t$ of the job informed by the user (frequently considered an upper bound of p_t);
- The resource requirement of the job, measured as the number of processors q_t;
- The arrival time r_t of the job (also called release date)

Typically several simplifications about t are made under the perspective of the RJMS: parallel efficiency, interdependence, and computation and communication intensities are often ignored. Instead, t is seen as an independent "black box" that will require q_t resources for $\tilde{p}_t$ units of time.

It is a common practice to log the characteristics of the jobs submitted in an HPC platform. One of the initiatives in maintaining such logs (also called traces) are the logs present in the Parallel Workloads Archive, shared in the Standard Workload Format (SWF) [24]. We exploit the rich information present in these logs to drive the simulation workflow explained in Section IV.

There is a large number of cost metrics [25] – which focus on different performance aspects of the scheduling – that can be used by HPC platform administrators. In this regard, we focus on three platform-wise, job-oriented metrics. The first

metric is the *waiting time* (Equation 1) which, measures the time that the job waited for execution, and it can be defined for a job t as:

$$w_t = start_t - r_t \tag{1}$$

where $start_t$ is the time that t started its execution. The second metric is the *bounded slowdown* (bsld or just slowdown, Equation 2) which, informally, measures the ratio between the time that a job t spent on the platform, and the actual processing time of t. Formally, the bsld can be defined as follows for a job t:

$$bsld_t = \max\left(\frac{w_t + p_t}{\max(p_t, \tau)}, 1\right) \tag{2}$$

where w_t is the waiting time of t and τ is a constant to prevent smaller jobs from reaching very high bsld values, and it is often set to 10 seconds. The reasoning behind slowdown is based on the expectation that the waiting time of a job should be proportional of its processing time, thus giving a balanced waiting time distribution among jobs with different characteristics, notably the processing time p_t.

Finally, the third metric is the *per-processor bounded slowdown* [26] (pp-bsld or pp-slowdown, Equation 3), which is defined for a job t as:

$$\textit{pp-bsld}_t = \max\left(\frac{w_t + p_t}{q_t \cdot \max(p_t, \tau)}, 1\right) \tag{3}$$

where w_t and τ are the same as for bsld. The reasoning behind the per-processor bounded slowdown is to normalize the slowdown results for jobs who perform the same amount of work, though with different degrees of parallelism (number of processors). The pp-slowdown can be seen as a more appropriate objective for the parallel batch scheduling problem, as it tries to balance the waiting time of the jobs in function of the number of processors q_t, which is not taken into account by the waiting time and slowdown.

Like other cost metrics, the waiting time, slowdown and pp-slowdown are usually considered in their cumulative versions, which means that one seeks to minimize the average waiting time, bsld or pp-bsld. It is worth noting that other metrics such as the maximum waiting time bsld or pp-bsld of all the jobs are also worthy of interest, though they must be taken more carefully, as we explain in Section V-C.

For a queue of jobs Q, we consider the average waiting time, slowdown or pp-slowdown of Q as being the average of the respective metric, over all jobs $t \in Q$.

B. Fairness and User Satisfaction

Specifically for slowdown and pp-slowdown, the expectation of a good scheduling performance is that the waiting time of the jobs should be proportional of its running time, that is, a job that must perform a larger amount of work (and thus requires many resources and/or for a longer period) could "afford" a longer waiting time. Indeed, it is arguable that the slowdown metric can be a good performance metric for a job-centric fairness, in comparison to other metrics such as waiting time. One could envision, however, that a better performance metric could be an user-centric metric, that captures the overall satisfaction among users. Although this could be indeed the case, one can not simply simulate user behavior by reproducing a workload trace due to the fact that the workload would change (in an on-line manner) in function of the scheduler's performance (e.g. a more efficient scheduler would stimulate users to submit more jobs and vice versa). At the time of writing of this paper, there is no consensus in the community about accurate and/or meaningful ways to simulate user behavior, which leads us to choose a job-centric approach rather than a user-centric one.

C. On-line Batch Scheduling Algorithm

In this work we consider a queue-ordering based, on-line batch scheduling algorithm that works as follows: the scheduler sorts – in increasing order according to a scheduling policy $f(t)$ – its waiting queue in two distinct events: (i) when a job arrives in the queue or (ii) when a resource (set of processors) is released and becomes available. When a job t is selected for execution and if the requested number of processors q_t is lower than the total number of processors available, then q_t processors are reserved for this job and they become unavailable. These processors will become available again only when p_t units of time have passed since the start of the execution of t. If the actual processing time p_t is larger than its estimate $\tilde{p}_t$, t is *killed*, that is, its execution is terminated.

In the case that there are not enough processors to process t, an *aggressive backfilling* subroutine [4] is applied. In this case, it is estimated at which time there will be enough resources to process t. Next, the scheduler looks for jobs in the waiting queue – following the order of jobs already established by the scheduling policy $f(t)$ – for which there are enough processing resources and that do not delay the execution of t. If a job meets these aforementioned conditions, then it "jumps ahead" and is scheduled for execution.

A key component of this scheduling algorithm is the scheduling policy $f(t)$. Although many scheduling policies can be devised, in this work we are concerned in comparing *simple* scheduling policies. Table I shows the simple scheduling policies considered in this paper. We define a scheduling policy $f(t)$ as *simple* if $f(t)$ is equal to one of the jobs' characteristics (notably FCFS, SPF and SQF, in which FCFS and SPF are well known policies in the off-line batch scheduling literature) or its meaning is intuitive and transparent to the platform user (notably SAF, which sorts the jobs according to their "area" or "geometry"). One can observe that we could also envision the "largest" variant of the presented scheduling policies. However, we decided to not consider them because in our preliminary experiments, and as well as reported in the works of Gaussier *et al.* [22], the "largest" variants present consistent worse scheduling performances than their "shortest" variants.

1) Starvation Prevention: It is possible to observe that among all scheduling policies presented in Table I, only FCFS

TABLE I
SCHEDULING POLICIES USED FOR COMPARISON.

Name	Description	Function
FCFS	First-Come-First-Served [27]	$f(t) = r_t$
SPF	Smallest Estimated Processing Time First [28]	$f(t) = \tilde{p}_t$
SQF	Smallest Resource Requirement First	$f(t) = q_t$
SAF	Smallest Estimated "Area" First	$f(t) = \tilde{p}_t \cdot q_t$

TABLE II
REAL WORKLOAD TRACES USED FOR EVALUATION OF THE SCHEDULING POLICIES.

Name	Year	# CPUs	# Jobs	Util %	Duration
HPC2N	2002	240	202,871	60.1	42 Months
SDSC Blue	2000	1,152	243,306	76.7	32 Months
SDSC SP2	1998	128	59,715	83.4	24 Months
CTC SP2	1997	338	77,222	85.2	11 Months
KTH SP2	1996	100	28,476	70.1	11 Months

can straightforwardly prevent starvation, that is, it guarantees that no job will wait for execution for an unbounded amount of time. Therefore, some starvation prevention mechanism is mandatory for the remaining policies in order to be applicable in real scenarios. In this regard, we adopted a simple thresholding mechanism [22], in which a job t would receive a maximum priority (bypassing the priority given by the scheduling policy $f(t)$) if its waiting time exceeds a maximum threshold value Θ. If many jobs receive a maximum priority at the same time, they will follow a FCFS order.

The threshold value Θ is an important parameter and indeed must be set with some caution: a too small threshold value would constrain the scheduling policy $f(t)$, by assigning the maximum priority at too many jobs and thus too many jobs will follow FCFS order instead of $f(t)$. Conversely, a too large threshold value could be too prohibitive in the Quality of Service point of view of the platform. In this work we set the threshold value Θ as being three times the maximum processing time estimate allowed by the platform. In general, it is a common practice in HPC platforms to set a maximum processing time allowed, setting the threshold as being three times this value means that the slowdown of larger jobs (i.e. jobs with processing time close to the maximum) would have a value of around three. Since the policies used this work prioritize sorter/smaller jobs. This thresholding mechanism is dedicated to the longer/larger jobs and is proven to be effective in preventing starvation, as we show in Section V that the majority of jobs that reach the threshold are larger ones.

IV. EXPERIMENTAL PROTOCOL

In this Section we present the experimental protocol adopted in this work. We make use of BatSim [29], which is a scientific instrument based on SimGrid [30] – a well known HPC systems simulator – specially tailored to simulate and study the behavior of batch scheduling algorithms. BatSim and SimGrid allows us to rapidly and accurately simulate the scheduling of many workload traces with using only a single workstation and in only a matter of days, which would not be feasible without simulation.

Table II shows the real workload traces used in this work. In order to provide statistically meaningful results with the scheduling of the traces, we adopted a sampling technique based on [31]. Algorithm 1 presents the pseudo-code. The idea is to generate new data using existing user profiles. A profile can be defined as the activity of a single user throughout the trace, split into many weekly time periods. To generate a new trace we combine several random permutations of each user's profiles. One can observe that this sampling technique is not capable of reflecting the workload changes in function of the scheduler's performance (as discussed in Section III-B). However, it allows to generate as many logs as needed while preserving the jobs' properties of each user.

Algorithm 1 Workload trace resampling algorithm.

Input: list of user $profiles$ P extracted from the original workload trace. Number of weeks in the resampled trace n_w
Output: resampled trace W_{res}

```
W_res ← ∅
for i = 1 to n_w do
    w_res ← ∅
    for each user profile p in P do
        p_{i_res} ← random weekly split from p
        add p_{i_res} to w_res
    end for
    append w_res in W_res
end for
return W_res
```

For each trace of Table II, we generate 10 samples using the aforementioned procedure. The size of each sample is proportional to the size of the original trace. Each sample is then simulated following the scheduling algorithm presented in Section III-C, taking into consideration each of the scheduling policies presented in Table I. The results for each scheduling policy and workload trace presented in the next Section are statistical summaries of the ten samples of each trace.

V. EXPERIMENTAL RESULTS

In this section we present the main results obtained by the experimental procedure described at Section IV. We perform several analysis in order to provide a better understanding of the behavior of the scheduling policies and what gains could be expected if a certain scheduling policy is chosen.

A. Overall Scheduling Performance

In this Section we aim to answer the following question: *If a certain scheduling policy of the Table I is chosen, what would be its overall scheduling performance if the chosen policy is kept throughout time?*

Figure 1 shows the overall performance results for the average slowdown, waiting time, and pp-slowdown. Each subplot refers to a workload trace from Table II. To avoid outlier interference in the results, for each trace and scheduling policy we discarded the best performing and the worst performing workload sample (see Section IV) from the 10 initial workload samples. In other words, we present only the scheduling results of the samples whose performance belongs to the 10-90% percentile range. Each subplot contains statistics of the scheduling simulation of these remaining samples. The solid lines in the subplots represent the cumulative mean of the objective metric (average slowdown, waiting time, or pp-slowdown) of the finished jobs at each week of simulation, from the beginning to the end of the workload, and the dashed lines represent the cumulative maximum and minimum average values of the respective metric at each week.

Looking at the scheduling performance in Figure 1, we can cluster the tested policies in two classes: the ones that are oblivious of the processing time estimate $\tilde{p}$ (FCFS and SQF), and the ones that are not oblivious (SPF and SAF). From the aforementioned Figure, we can observe a strong correlation between the scheduling performance of these clusters, with the former cluster consistently presenting worst performances than the latter. This result is expected: for the slowdown and pp-slowdown, jobs with a lower $\tilde{p}$ – and thus lower p, since $\tilde{p}$ is an upper bound of p – have a higher risk of inflating the metrics if they wait too much (see Equations 2 and 3). By favoring jobs with a lower $\tilde{p}$ (SPF and SAF), we assure that these high risk jobs are executed quickly, and thus the average for both slowdown and pp-slowdown are kept under control. The waiting time is also favored by prioritizing jobs with lower $\tilde{p}$, since for all traces these jobs are more frequent [32].

One point that is worth noticing is how much can be gained in quantitative values if a policy other than FCFS (notably SPF or SAF) is chosen and kept during a long period. In our experiments we achieve performance gains up to 83.4% (SPF), 61.4% (SAF), and 85.1% (SAF) for the slowdown, waiting time, and pp-slowdown respectively, in comparison with FCFS. It is important to note here that the scheduling simulation is performed with a starvation prevention mechanism (see Section III-C). Therefore, these gains can be obtained while guaranteeing that no job will starve.

Another important observation is how SAF – which in contrast with SPF, is less known in the literature – performs consistently well in all objectives considered. We further address this phenomenon in the next Section.

B. Is SAF the ultimate simple policy?

As highlighted in the previous Section, the scheduling policies that are not oblivious to the processing time estimate $\tilde{p}$ (notably SPF and SAF) are the ones who achieved the most consistent good performances in the experiments that we performed. In this Section we make a further analysis on which are the characteristics of the jobs that make them prioritized/delayed by these two policies, with an emphasis on the delayed jobs.

For the processing time estimate $\tilde{p}$ this analysis can be easily devised: SPF delays jobs with a larger $\tilde{p}$ and SAF is similar, with the distinction that it considers the number of processors q as well. This raises the importance of our thresholding mechanism, which specifically concerns jobs with a large $\tilde{p}$.

In its turn, for the number of processors q, Figure 2 shows the number of processors q of the top 100 jobs – of each sample of each trace – who got delayed the most (here defined as the jobs with the highest slowdown) for each scheduling policy. An interesting observation here is that SPF is oblivious to the number of processors q and thus no correlation should be expected for the delayed jobs in function of q. Therefore, SPF had a high risks of delaying jobs with smaller q which, in principle, should be easier to be scheduled in an HPC platform.

Indeed, we recall a known observation [25] that the slowdown and the waiting time metrics (arguably the most popular ones) do not take into consideration one important dimension of the scheduling problem: the number of requested processors q. Jobs that perform the same amount of work though with different shapes are treated indifferently by these metrics. The pp-slowdown generalizes the standard slowdown by including the number of processors q in the metric.

At this light SAF shows up as a solid policy among the simple ones we evaluated. It achieved close to best observed performances for the slowdown and waiting time objectives, and systematically outperformed all other simple policies for the pp-slowdown objective (Figure 1) . This complies with the results of our previous work [20], where the machine learned policies converged to functions that contain a SAF-like component. Although one can claim that SAF could be biased towards pp-slowdown, since with pp-slowdown we would seek to minimize an objective function that is related to the area of the jobs, we argue that the pp-slowdown is a more appropriate objective for the parallel batch scheduling problem, in comparison with waiting time or slowdown.

C. Accounting the Maximum: one should care with caution

One can notice that in this work we only seek to find good scheduling algorithms aiming at the average of the objective functions and not the maximum. Although one can argue that the maximum of the objective functions are important as well, in this Section we present some observations found by our study that show that aiming only for the maximum can be potentially problematic.

The first point is that the maximum metric is centred at the performance of only one job, meaning that the value of the maximum can be unstable and subject to unpredictable factors, such as unavoidable bursts of jobs submissions and/or jobs that have some characteristic that can potentially mistakenly inflate the metric. To illustrate this potential, we clustered the jobs into two classes: the premature jobs, in which the difference between the processing time estimate $\tilde{p}$ and the actual processing time p is at least 100 times higher, and the standard jobs, which are the remaining jobs. Table III shows the percentage of premature jobs found for each workload trace. What is interesting to observe is that the number of

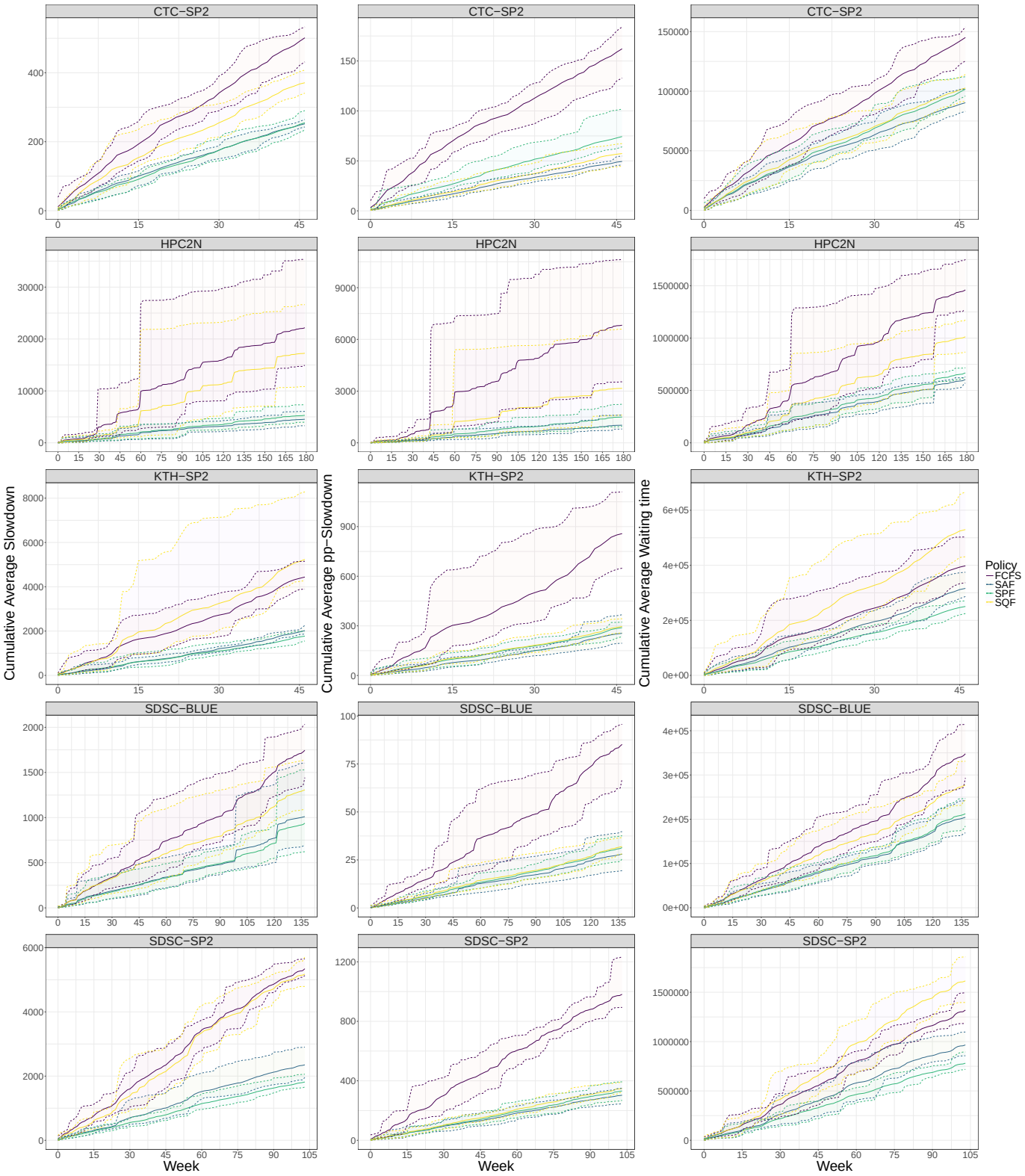

Fig. 1. Cumulative weekly average slowdown, pp-slowdown and waiting time: For each trace, the middle solid line represents the mean and the two dashed lines represent the lower and upper 10-90 percentiles.

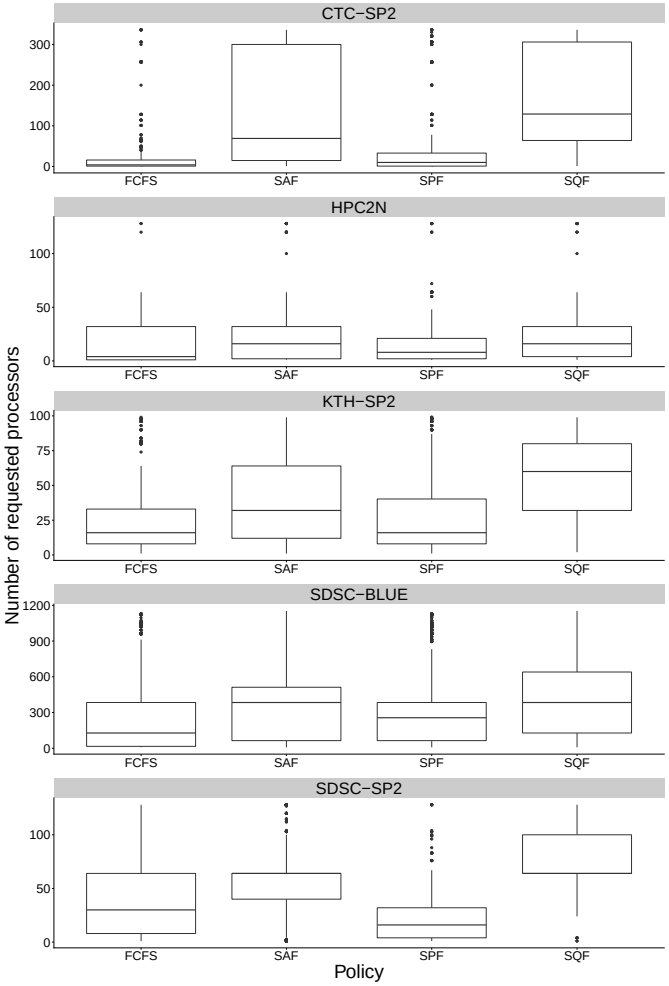

Fig. 2. Number of processors of the top 100 jobs with highest slowdown values.

premature jobs is not negligible, up to one third of all of the jobs of the trace. Furthermore, the difference between $\tilde{p}$ and p can be sometimes quite extreme: jobs that are marked as successful jobs (i.e. job that did not crash) and require the maximum processing time allowed $\tilde{p}$, though actually execute for around one minute happen in every trace. Since these jobs are marked as successful, we can not discard them from the analysis.

Therefore, any scheduling policy that prioritizes jobs in function of the processing time estimate $\tilde{p}$ has a risk of delaying these premature jobs and, when the objective function of these jobs are evaluated, they will obtain poor results which will harm the maximum of the objective function. To illustrate this effect, Table IV shows the ratio between the average slowdown of the premature jobs and the average slowdown of the standard ones, for all traces and scheduling policies. We can notice that the difference in scheduling performance of these two classes of jobs is large, up to 17 times larger for all policies in the HPC2N trace, and this difference in the maximum slowdown between these two classes (result not shown in Table IV) is even larger. We can also notice that the this difference is often amplified by policies that takes $\tilde{p}$ into account (SPF and SAF).

Agreeing whether or not these performance gaps are due to the scheduler is always up to argument. However, Figure 3 shows a more holistic view of the scheduling performance: we grouped the jobs in many categories that are in function of the jobs' scheduling performance, from the jobs that were executed immediately (slowdown of 1), to the jobs that were poorly scheduled (slowdown of at least 100). We can observe that choosing another policy than FCFS shows performance improvements in all categories: the number of jobs who got executed immediately increases and the number of jobs in all other categories (the jobs who had to wait) decreases, with an exception of the SPF policy at the 1-10 slowdown range. These results are even more impressive for the category of jobs with poorer scheduling performances (100+ slowdown). For instance, by choosing SAF, the number of jobs who got badly scheduled can be lowered by more than half, up to 2.8x less poorly scheduled jobs in comparison with FCFS.

All of these points elucidate the importance of analyzing the scheduling performance in a holistic view, and the caution that must be taken into account when evaluating the scheduling performance with maximum values. We would certainly overlook these good properties of the studied scheduling policies if we had considered only the maximum of the objective functions.

TABLE III
PERCENTAGE OF PREMATURE JOBS FOR EACH WORKLOAD TRACE

Trace	% of premature jobs
HPC2N	17.4
SDSC Blue	30.2
SDSC-SP2	16.1
CTC-SP2	9.5
KTH-SP2	12.4

D. Backfilling Influence

One important question that raises when the queue ordering policy is changed (see Section III-C) is how the backfilling mechanism behaves in function of the queue ordering policy. Although it is well known that backfilling increases the platform's utilization and is unlikely to harm the original (without backfilling) schedule, its relevance in performance is not clear. This question is also worth of importance to bring a clearer notion about the predictability of the scheduling policies, that is, given one policy, how much it is likely that the jobs will actually follow such an order.

In order to clarify this point, for all samples of each trace and each scheduling policy we kept track on how many jobs got scheduled to execution by the backfilling mechanism. Figure 4 shows the distribution of the number of backfilled jobs over all samples, for each workload trace and scheduling policy. One interesting observation is the absence of backfilled jobs for the SQF policy for every trace and sample. This result is expected and we formalize it with the following proposition:

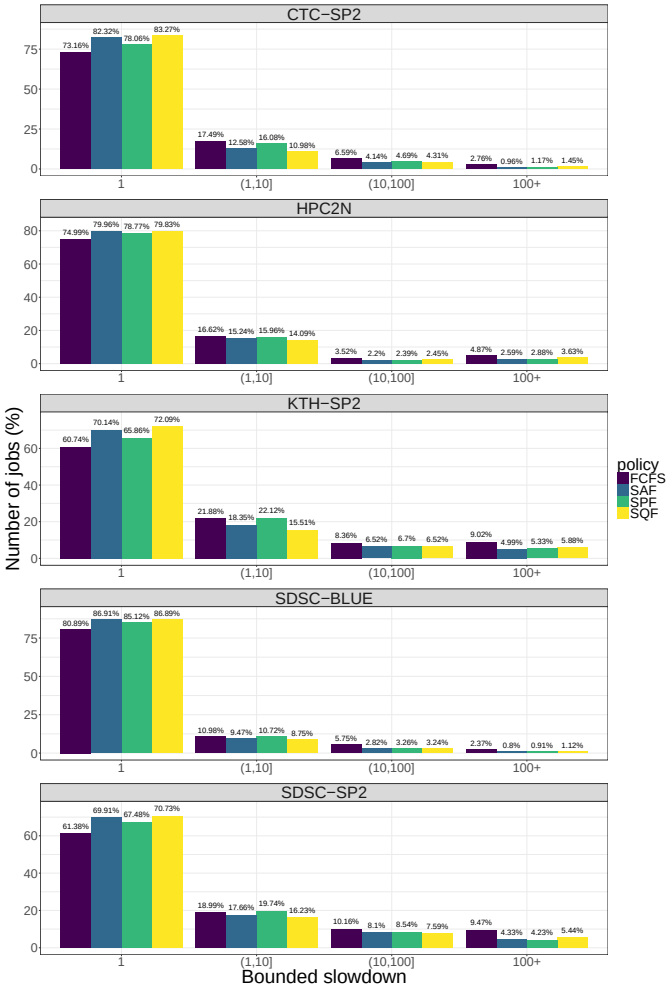

Fig. 3. Distribution of the bounded slowdown values for all jobs

TABLE IV
RATIO OF THE AVERAGE SLOWDOWN BETWEEN THE PREMATURE THE STANDARD JOBS

Policy	HPC2N	SDSC Blue	SDSC SP2	CTC SP2	KTH SP2
FCFS	17.84	3.59	3.94	5.58	8.69
SPF	17.29	7.17	4.36	5.04	12.09
SQF	14.02	2.96	2.04	1.67	9.31
SAF	17.88	7.41	3.79	2.61	11.41

Proposition 1. *If the aggressive backfilling algorithm uses a queue of jobs sorted by SQF and there is no threshold mechanism added to the scheduling, no job is backfilled.*

Proof. As explained in Section III-C, scheduling decisions are performed in two cases:

1) When a job arrives in the queue: in this case, let t_h be the job with the highest priority in the queue. Job t_h is in the queue, therefore there is not enough resources to process t_h. Since the queue is sorted by SQF order, there is no job in the queue that requires less resources than t_h, so none of them can be backfilled. If a new job t arrives in the queue and its number of required processors is lower than the number of processors required by t_h, SQF will assign t with the highest priority and thus backfilling will no longer be applied for t. Conversely, where t requires more processors than t_h, t cannot be backfilled as aforementioned.
2) When a job is finished and its allocated resources are released: in this case, the jobs will be scheduled for execution following SQF order until it is no longer possible to schedule jobs with the current available resources. At this point, there are not enough resources to schedule the job with the highest priority in the queue and, since the queue is sorted in SQF order, no other job in the queue can be backfilled as aforementioned.

Since in both of the above cases it is impossible to backfill jobs, no jobs are backfilled.

□

Yet, some backfilling may happen when using SQF with jobs that exceeded the threshold in the waiting queue (since they break the SQF order). However, such jobs are expected to be very few. This explains some results found by Lelong *et al.* [21], in which they state that the SQF policy did not lead to many backfilling decisions in their experiments.

Interestingly, using SAF and SPF resulted in 78% and 56% less backfilled jobs on average, respectively, when compared to FCFS. Although it is unlikely that backfilling would harm the scheduling, as mentioned above, SPF and SAF are more consistent and predictable policies, since jobs are more likely to be scheduled for execution following the policy order, as oppose to being scheduled by "jumping ahead" in the waiting queue in unpredictable moments.

VI. Conclusions and Future Work

As the scale and power of high-performance computing (HPC) platforms increases, it becomes more crucial to deploy efficient resource management approaches (notably scheduling algorithms) in order to prevent the dampening of such increase in computing power. In an adversarial manner, it is also important that such scheduling algorithms stay simple and easily understandable by the users. Furthermore, changing the scheduling algorithm is often seen as a risky move, mainly due to the possibility of having unseen and unpredictable changes in the performance of the platform, which could be detected only after a long period of time.

In this paper, we move towards providing more knowledge and experience on what are the expectations if one decides to change the First-Come-First-Served (FCFS) scheduling policy with aggressive backfilling – the popular EASY Backfilling – scheduling algorithm. We selected a class of simple scheduling algorithms that differs from EASY Backfilling by changing the scheduling policy (other than FCFS) and adding a thresholding mechanism (to provide the same no starvation guarantees as FCFS). We used a flexible and reliable simulation software and exploited the rich information presented in HPC platform workload traces to find what could be observed and gained

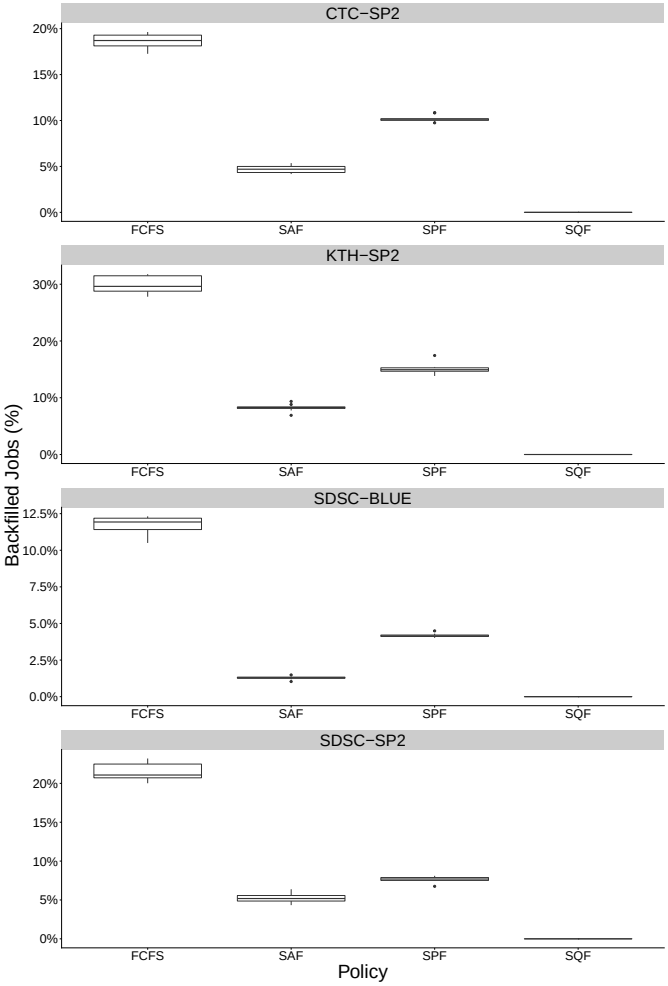

Fig. 4. Distribution of backfilled jobs between resamplings.

by using these other simple scheduling algorithms rather than EASY Backfilling.

Our results indicate that one can only gain by replacing EASY Backfilling with simple policies that consider the estimated processing time and the required resources, notably the Shortest Processing time First (SPF) and Shortest Area First (SAF). By adding a simple thresholding mechanism, it is possible to obtain significant performance improvements for the long run, using three relevant performance objectives, while also guaranteeing that no job will wait for an unbounded amount of time. We show that these simple policies not only present better performance in average values, but they also significantly increase the number of jobs executed instantly (without waiting) and lower the number of jobs that wait for a long time. The performance gains over EASY Backfilling is distributed among all waiting jobs.

These simple policies also show that they can perform well with less interference from backfilling: the scheduler is more likely to follow the original order as set by the chosen scheduling policy, and not by the rules of backfilling, thus providing more predictability and transparency, two properties that are sought by HPC platform administrators.

We also highlight a less known scheduling policy in the literature, the Shortest Area First (SAF). In our experimental campaign, we found that this policy managed to consistently provide close to the best (if not the best) observed performance in all scenarios and performance objectives we evaluated. For instance, considering the slowdown objective, SAF not only provided an average overall performance increase up to 83.4%, but as well increased the number of jobs that run immediately by up to 9% and lowered the number of jobs who waited for a long time (very long slowdown) by up to 2.8 times, in comparison with FCFS. This result reinforces the relevance of the jobs' area property, which was seen in our previous work [20], and raises the question about possible analytical properties of SAF. Nevertheless, we reinforce that SAF must be considered as a baseline of comparison in future parallel batch scheduling research.

Last but not least, we present some cautions that must be considered if one wants to provide a scheduling algorithm that minimizes the maximum of an objective function. Taking the slowdown objective function as an example, we observed a class of jobs whose presence in the workload is not negligible and can mistakenly lead to inflated maximum slowdown values. If one only looks at the maximum of an objective function to evaluate the scheduling performance, some good scheduling policies (as the aforementioned ones) can be overlooked.

There is still work to be done in this subject: the first point is to address how users play a role in the global scheduling. As mentioned in Section III-B, our approach on evaluating the scheduling performance is centred at the platform and the jobs, where users are not taken into account. An ideal scenario would be to simulate the users reacting to the scheduler's performance. In this regard, accurate and reliable user models are required to properly simulate user behavior. Another point is how we can exploit SAF to provide SAF-like scheduling policies that are adapted to certain situations and/or time periods.

ACKNOWLEDGMENT

This research was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

The authors would like to thank Arnaud Legrand for his help and comments and Michael Mercier for his help in the simulation. We thank the contributors of the Parallel Workloads Archive, Victor Hazlewood (SDSC SP2), Travis Earheart and Nancy Wilkins-Diehr (SDSC Blue), Lars Malinowsky (KTH SP2), Dan Dwyer, Steve Hotovy (CTC SP2), and Ake Sandgren (HPC2N).

REFERENCES

[1] "TOP500 Supercomputer Sites," https://www.top500.org/, 2018, online; last access 30 november 2018.

[2] P. Brucker, *Scheduling Algorithms*, fifth edition ed. Springer, 2007.

[3] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "Theory and practice in parallel job scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1997, pp. 1–34.

[4] 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*, vol. 12, no. 6, pp. 529–543, 2001.

[5] C. A. Floudas and X. Lin, "Mixed integer linear programming in process scheduling: Modeling, algorithms, and applications," *Annals of Operations Research*, vol. 139, no. 1, pp. 131–162, Oct 2005. [Online]. Available: https://doi.org/10.1007/s10479-005-3446-x

[6] H. Al-Daoud, I. Al-Azzoni, and D. G. Down, "Power-aware linear programming based scheduling for heterogeneous computer clusters," in *International Conference on Green Computing*, Aug 2010, pp. 325–332.

[7] J. E. Pecero, D. Trystram, and A. Y. Zomaya, "A new genetic algorithm for scheduling for large communication delays," in *European Conference on Parallel Processing*. Springer, 2009, pp. 241–252.

[8] E. S. H. Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 2, pp. 113–120, Feb 1994.

[9] A. Agarwal, S. Colak, V. S. Jacob, and H. Pirkul, "Heuristics and augmented neural networks for task scheduling with non-identical machines," *European Journal of Operational Research*, vol. 175, no. 1, pp. 296 – 317, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0377221705004303

[10] F. Xhafa and A. Abraham, "Computational models and heuristic methods for grid scheduling problems," *Future Generation Computer Systems*, vol. 26, no. 4, pp. 608 – 621, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X09001782

[11] B. S. Baker, E. G. Coffman, Jr, and R. L. Rivest, "Orthogonal packings in two dimensions," *SIAM Journal on computing*, vol. 9, no. 4, pp. 846–855, 1980.

[12] M. Bougeret, P. Dutot, K. Jansen, C. Otte, and D. Trystram, "Approximation algorithms for multiple strip packing," in *Approximation and Online Algorithms, 7th International Workshop, WAOA 2009, Copenhagen, Denmark, September 10-11, 2009. Revised Papers*, 2009, pp. 37–48. [Online]. Available: https://doi.org/10.1007/978-3-642-12450-1_4

[13] D. Ye, X. Han, and G. Zhang, "Online multiple-strip packing," *Theoretical Computer Science*, vol. 412, no. 3, pp. 233 – 239, 2011, combinatorial Optimization and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397509006896

[14] J. L. Hurink and J. J. Paulus, "Online algorithm for parallel job scheduling and strip packing," in *International Workshop on Approximation and Online Algorithms*. Springer, 2007, pp. 67–74.

[15] S. Zhuk, "Approximate algorithms to pack rectangles into several strips," *Discrete Mathematics and Applications dma*, vol. 16, no. 1, pp. 73–85, 2006.

[16] D. Ye and G. Zhang, "On-line scheduling of parallel jobs in a list," *Journal of scheduling*, vol. 10, no. 6, pp. 407–413, 2007.

[17] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[18] W. Tang, Z. Lan, N. Desai, and D. Buettner, "Fault-aware, utility-based job scheduling on BlueGene/P systems," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.

[19] E. Gaussier, D. Glesser, V. Reis, and D. Trystram, "Improving backfilling by using machine learning to predict running times," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 64:1–64:10. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807646

[20] D. Carastan-Santos and R. Y. de Camargo, "Obtaining dynamic scheduling policies with simulation and machine learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 32:1–32:13. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126955

[21] J. Lelong, V. Reis, and D. Trystram, "Tuning EASY-Backfilling Queues," in *21st Workshop on Job Scheduling Strategies for Parallel Processing*, ser. 31st IEEE International Parallel & Distributed Processing Symposium, Orlando, United States, May 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01522459

[22] E. Gaussier, J. Lelong, V. Reis, and D. Trystram, "Online tuning of easy-backfilling using queue reordering policies," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2304–2316, Oct 2018.

[23] G. P. Rodrigo, P.-O. Östberg, E. Elmroth, K. Antypas, R. Gerber, and L. Ramakrishnan, "Towards understanding hpc users and systems: a nersc case study," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 206–221, 2018.

[24] D. G. Feitelson, D. Tsafrir, and D. Krakov, "Experience with using the parallel workloads archive," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2967 – 2982, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001154

[25] D. G. Feitelson and L. Rudolph, "Metrics and benchmarking for parallel job scheduling," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1998, pp. 1–24.

[26] D. Zotkin and P. J. Keleher, "Job-length estimation and performance in backfilling schedulers," in *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*. IEEE, 1999, pp. 236–243.

[27] J. Skovira, W. Chan, H. Zhou, and D. Lifka, "The easy—loadleveler api project," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 1996, pp. 41–47.

[28] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan, "Characterization of backfilling strategies for parallel job scheduling," in *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 514–519.

[29] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard, "Batsim: A realistic language-independent resources and jobs management systems simulator," in *Job Scheduling Strategies for Parallel Processing*, N. Desai and W. Cirne, Eds. Cham: Springer International Publishing, 2017, pp. 178–197.

[30] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: http://hal.inria.fr/hal-01017319

[31] D. G. Feitelson, "Resampling with feedback — a new paradigm of using workload data for performance evaluation," in *European Conference on Parallel Processing*. Springer, 2016, pp. 3–21.

[32] "Parallel Workloads Archive: Logs," http://www.cs.huji.ac.il/labs/parallel/workload/logs.html, 2018, online; last access 30 november 2018.