Adapting Batch Scheduling to Workload Characteristics: What can we expect From Online Learning?

Arnaud Legrand, Denis Trystram, Salah Zrigui
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
Email: firstname.lastname@inria.fr

***Abstract*—Despite the impressive growth and size of super-computers, the computational power they provide still cannot match the demand. Efficient and fair resource allocation is a critical task. Super-computers use Resource and Job Management Systems to schedule applications, which is generally done by relying on generic index policies such as First Come First Served and Shortest Processing time First in combination with Backfilling strategies. Unfortunately, such generic policies often fail to exploit specific characteristics of real workloads. In this work, we focus on improving the performance of online schedulers. We study mixed policies, which are created by combining multiple job characteristics in a weighted linear expression, as opposed to classical pure policies which use only a single characteristic. This larger class of scheduling policies aims at providing more flexibility and adaptability. We use space coverage and black-box optimization techniques to explore this new space of mixed policies and we study how can they adapt to the changes in the workload. We perform an extensive experimental campaign through which we show that (1) even the best pure policy is far from optimal and that (2) using a carefully tuned mixed policy would allow to significantly improve the performance of the system. (3) We also provide empirical evidence that there is no one size fits all policy, by showing that the rapid workload evolution seems to prevent classical online learning algorithms from being effective.**

I. Introduction

The number of applications that require the usage of super-computers is increasing rapidly. Hardware producers, despite their best efforts, are simply unable to match this ever-growing demand. As a result, we have today a large number of applications competing for limited resources. Thus, ordering the jobs in a way that guarantees maximum efficiency and fairness is more crucial than ever.

Super-computers rely on Resource and Job Management Systems (RJMS), for monitoring and control. A major part of any RJMS is the job scheduler, whose main task is to decide in which order the jobs will be executed. However, taking the right decision is a complex problem that requires considering a large number of factors. Some of which are clear and visible but most are not. In the face of such growing complexity, many system administrators opt for the "simple" answer: use simple dispatching rules that are based on intuition and that offer certain guarantees, e.g First Come First Served (FCFS) to prevent starvation or Shortest processing time First (SPF) because it favors interactivity. However, they are far from optimal and many studies [1]–[3] show that there is still room for software optimization. A common practice for RJMS is to keep execution logs that detail the history of the platform: the characteristics of the submitted jobs, their arrival times and other important information. In this work, we explore the possibility of employing this historical data to adapt to future workload using more flexible scheduling policies. We base our experiments on EASY [4], which is one of the most popular backfilling schemes, and we propose a data-driven experimental campaign through which we exploit real execution traces in the form of logs extracted from the parallel workload archives [5]. First, we show the limits of simple, index policies. Then, we propose a new class of policies, which we call "Mixed policies". Using this class we prove that simple policies are far from optimal and that under the correct conditions, we can obtain significant gains.

- We show that simple scheduling policies used in the scientific literature and in industrial applications like FCFS and SPF are far from optimal and that Smallest Area First (SAF), another simple policy, performs better overall.
- We also prove that it is possible to generate policies that significantly outperform these pure policies by mixing job features such as the estimate processing time, the required resources, and the waiting time in a simple weighted linear combination.
- We present a mapping of the space of possible policies through which we show that the evolution of the workload through time is very chaotic, which prevents online learning algorithms from being effective.

The remainder of this paper is organized as follows. Section II presents the context under which the experimental campaign was performed. In Section III we provide a background on the works done to improve the performance of EASY and aggressive backfilling, and the works that implement machine learning techniques to improve the performance of schedulers in general. Sections IV and V respectively define the index policies

1530-2075/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPS.2019.00077

and the methodology that was used throughout the work. In Section VI we compare a set of pure policies, and in Section VII we present and test the proposed method to obtain mixed scheduling policies. Finally, we give some concluding remarks and an open discussion in Section IX.

II. Context

A scheduler uses a scheduling heuristic to order the jobs in an execution queue and a metric, also called objective, to measure the quality of the scheduling method.

A. Jobs

We consider an online scheduling model, where the jobs arrive at different times unknown in advance. The information available about the job upon its arrival are: the requested resources (number of requested processors), the requested processing time also called the estimated processing time (an estimation/upper limit of the processing time given by the user) and the arrival time itself. The scheduler chooses one or more of the waiting jobs to execute at each time-step. The jobs cannot be preempted.

B. Scheduling heuristic: EASY-backfilling

Scheduling is the process of selecting the order in which the jobs will be executed. One of the most popular techniques used to perform such task is the backfilling algorithm [4]. Backfilling works by finding holes in the scheduling Gantt chart and moving forward smaller jobs to fill these holes.

EASY [4] is a scheduling algorithm that uses a queue to select and backfill jobs. Algorithm 1 recalls how it works. At any time a scheduling decision is required (*i.e.* job submission or termination), the scheduler goes through the job queue in a *primary order* predetermined by the selected index policy and starts them until it encounters a job that cannot be started immediately. At this point, the scheduler makes a reservation for this particular job which ensures that it will not be delayed from its initial position. Then, it goes through the rest of the job queue in a *backfilling order* and execute any jobs as long as it does not delay the unique reservation mentioned earlier. This is known as backfilling. One of the most popular variations of the EASY algorithm is EASY-FCFS-FCFS where the jobs are ordered and backfilled by their arrival time.

All the comparisons and the techniques in the remainder of this paper are applied to the primary queue and the backfilling policy to fixed to SPF. We chose this setting because in [2], Lelong *et al.* showed that reordering the primary queue is more beneficial than simply reordering the backfilling queue and in [6], The authors showed that SPF is a good policy for backfilling.

Algorithm 1: EASY Algorithm

```
Input : Queue Q of waiting jobs sorted by
        increasing submission times.
Order primary queue according to an index policy
for job j in Q do
    if j can be started then
        Start j
        Remove j from Q
    else
        Reserve j at the earliest possible time
          according to the estimated running times
          of the currently running jobs.
        break
    end
end
Backfill according to SPF
L ← Q - [reserved job]
Order L according to SPF
while L not empty do
    Start all the jobs that can be backfilled without
      delaying the reservation from Q
end
```

C. Metric

Throughout this paper we use the bounded slowdown (*BSLD*) metric as it is accepted as one of the most popular one metrics to measure the performance of scheduling heuristics [7], [8]. The *BSLD* of a job j is defined as follows:

$$BSLD_j = max\left(\frac{wait_j + p_j}{max(p_j, \tau)}, 1\right), \tag{1}$$

where $wait_j$ and p_j are respectively the waiting time and the processing time of job j. τ is a constant that prevents the slowdown of smaller jobs from surging. We set τ to 10 seconds for the experiments.
For this paper, we focus on the average *BSLD* for all the jobs over a period of time. The average *BSLD* of n jobs is computed in the following way:

$$average_{BSLD} = \frac{1}{n}\sum_{j=1}^{n} BSLD_j \tag{2}$$

It is worth noting however that our work does not particularly depend on our choice of metric. The average *BSLD* could be replaced by any objective function or metric the user seeks to optimize.

III. Related work

A great amount of research has been devoted to improving the performance of EASY. Most are based on the idea of manipulating the main and/or the backfilling queues. In [9] Perkovic *et al.* proposed the use of

speculative backfilling to counter the almost systematic overestimation of the execution time of submitted jobs. However, these works do not address dependencies between the workload and the objectives [10]. The dynP scheduler [2] offers an online approach to tune EASY queue. dynP requires the full simulated schedule for each of the candidate policies in every scheduling step, which makes the scheduling cost much higher than simple EASY. Several works attempted to use machine learning techniques to predict and enhance HPC systems performance. In [11], Papadopoulou *et al.* developed an approach to predict the communication cost. They constructed a set of descriptive metrics and used a multiple variable regression model. Their approach proved to be successful in predicting and subsequently controlling the cost of communication.

Many researchers focused on predicting the running time of jobs as a mean to take better scheduling decisions. In [12] Duan *et al.* proposes a hybrid Bayesian-neural network approach to model and predict the run times of scientific application. It requires a detailed analysis of the system and the jobs. It also incorporates expert domain knowledge. The most relevant work that uses running time predictions is the EASY++ algorithm presented in [1], where Tsafrir *et al.* used a history-based system generated predictions of jobs lengths to backfill instead of user estimations. This method proved to be quite successful despite its relative simplicity. This work was followed by Gaussier *et al.* [3] where they used a machine learning technique to obtain even better predictions. In [13] a framework was proposed to automatically detect and diagnose performance anomalies in HPC systems.

Perhaps the most comparable works to the one presented in this paper are [14] and [8]. In [14], the authors developed DeepRM, a multi-resource cluster scheduler that uses deep reinforcement learning to solve the problem of packing with multiple resource demands. In [8] Carastan-Santos and Camargo used synthetic workloads to create general heuristics that improve the slowdown metric. They combined the basic job characteristics in a non-linear function and they used linear regression to devise new heuristics. Both [14] and [8] rely on synthetic data to train their approach.

IV. Index policies

We study two distinct types of index policies in this work, namely pure and mixed policies. Both types are based on job characteristics.

A. Job characteristics

We use the following job characteristics during the experimental campaign:

- q_j: (requested resources) the number of processors the user requested.
- $\tilde{p}_j$: (requested/estimated processing time) the estimated processing time provided by the user, it also serves as an upper limit to the time the job is allowed to run. The actual processing time p_j can only be obtained after the execution of the job.
- $wait_j$: (waiting time) How long a job j spent in the waiting queue:
  $wait_j = current\_time - submission\_time_j$
- ρ_j: (estimated ratio) $\frac{\tilde{p}_j}{q_j}$.
- a_j: (estimated area j) $\tilde{p}_j q_j$.
- exp_j: (estimated expansion Factor) $\frac{wait_j + \tilde{p}_j}{\tilde{p}_j}$: the ratio of the total time a job is expected to stay in the system (waiting time plus estimated processing time) normalized by its estimated processing time. This characteristic is rather special since it reflects the *estimated* value of the objective function. Note that the *BSLD* could have been used instead of the expansion Factor but it makes very little to no difference in term of ordering since only the smaller jobs (which usually have the least impact on performance) are marginally concerned.
  Scheduling using exp_j is expected to be a good or at least an important strategy. But it is unknown how it will perform at this point since it does not account for q_j.

B. Pure policies

With each of the six aforementioned job characteristics, we construct two scheduling policies: one that prioritizes the lowest score given by the characteristic and another the highest. So we have the following 12 ***pure*** policies:

- FCFS: First Come First Served
- LCFS: Last Come First Served
- SPF: Smallest estimated Processing time First
- LPF: Longest estimated Processing time First
- SQF: Smallest Resource Requirement First
- LQF: Largest Resource Requirement First
- SAF: Smallest estimated Area First
- LAF: Largest estimated Area First
- LEXP: Largest estimated Expansion Factor First
- SEXP: Smallest estimated Expansion Factor First
- LRF: Largest estimated Ratio First
- SRF: Smallest estimated Ratio First

In this work, we only focus on these 12 pure policies. Many others were not included. Our aim is not to do an exhaustive review of all the policies in the literature but to illustrate certain characteristics of generic scheduling policies.

C. Mixed policies

We now introduce the concept of mixed policies and the method we use to construct them.

A job j is characterized by a feature vector $x_j = (q_j, \tilde{p}_j, wait_j, \rho_j, exp_j, a_j)$.
At each scheduling decision, we define the score of any job j using Equation (3).

$$score(\mathbf{w}, x_j) = \mathbf{w}^T x_j \quad \mathbf{w} \in \mathbb{R}^n \tag{3}$$

where $\mathbf{w}$ is the weight vector of the mixed policy: each feature x_i has a corresponding weight $\mathbf{w}_i$. These weights are what determine how the mixed policy behaves. The absolute value of a weight $|\mathbf{w}_i|$ indicates the importance of the corresponding characteristic x_i when ordering the jobs. While the sign determines the ordering itself, a positive value means that shorter/smaller jobs are prioritized, while a negative value means that longer/larger jobs are prioritized.

The scoring function is scale-invariant; the order given by $score(\lambda\mathbf{w}, x_j)$ is the same as the order given by $score(\mathbf{w}, x_j)$ for all $\lambda > 0$. Hence, we normalize $\mathbf{w}$ and impose that $||\mathbf{w}||_1 = 1$. This constraint reduces the size of the search space and stabilizes the learning process (which will be explained in detail in Section VIII-A). Every pure policy corresponds to a vertex of the polytope $||\mathbf{w}||_1 = 1$. *E.g.* FCFS corresponds to (0,0,1,0,0,0) and LCFS corresponds to (0,0,-1,0,0,0).

Mixed policies are an alternative method to model the scheduling problem. We move from a discrete optimization to a continuous optimization problem. We construct a search space that is small in size and instead of finding the best ordering of n independent jobs we intend to find the best weight for i features where i is much smaller than n.

V. Experimental methodology

We tried to be as transparent as possible and to make our work reproducible [15]. We provide a snapshot of the workflow we used throughout this work as a link to a git repository[1], which includes a nix [16] file that describes all the dependencies and four R notebooks that allow regenerating all the figures.

We make several simplifying assumptions about the platform. We discard all topological information related to the platforms that generated the traces. We do not take into account the topology of the cluster and we treat it as a single collection of homogeneous resources where all processors are considered indistinguishable from each other and the cost of communication is considered non-existent.

In this work, we replace the RJMS with a EASY-backfilling lightweight simulator [1] written in OCaml. It supports tuning the Primary, Backfilling queues and it discard all topological information from the machine. So equation 2 can be written in the following form:

[1]https://gitlab.inria.fr/szrigui/mixed-policies

Trace	#CPU (#nodes*node_size)	#Duration	#Jobs	Average job duration
KTH-SP2	100 (100*1)	11 Months	27670	8579 (s)
CTC-SP2	338 (338*1)	11 Months	68687	9807 (s)
SDSC-SP2	128 (128*1)	24 Months	49809	6318 (s)
SDSC-BLUE	1,152 (144*8)	32 Months	208716	3184 (s)

TABLE I
Workloads

$$average_{BSLD} = \frac{1}{n}\sum_{j=1}^{n} F(x_j, \mathbf{w}), \tag{4}$$

where $\mathbf{w}$ represents the weight of the index policy, and F represents the simulator that will take all the jobs, execute them, and return the value of $BLSD$ of each job.

A. workload and platform

a) data: The goal is to improve scheduling performance using information extracted from job characteristics. For this reason, we choose real-world traces (from the parallel workload archives [5]) instead of artificially generated data. Table I outlines the workload used throughout the experimental campaign.

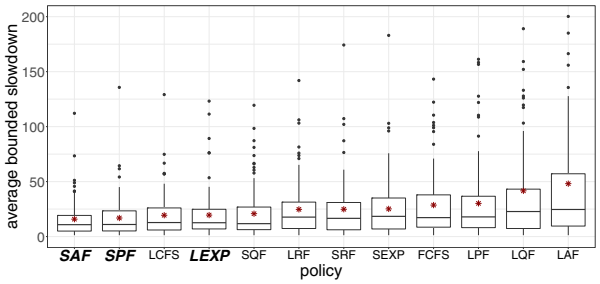
For every trace, we ignore the first period since it generally corresponds to a benchmarking/testing phase and is not representative of the true workload of the system. Then, we split the trace on a weekly basis and remove the jobs that start in one week and finish in another.

B. Starvation

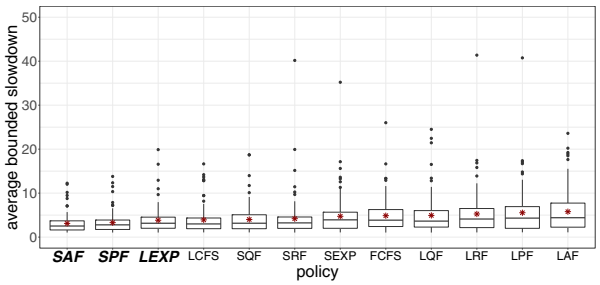
Starvation occurs when a job is denied the resources necessary for its execution for a very long (possibly unbounded) period of time. EASY, as defined in section II-B, has a risk of causing some jobs to starve. (*e.g* using SPF to order the primary queue may cause longer jobs to starve). The popularity of FCFS in RJMS stems mainly from its natural ability to prevent starvation.

To avoid starvation, we rely on a simple but effective *thresholding* mechanism. When the waiting time of a job exceeds a certain value, it is moved to the head of the queue immediately regardless of the scheduling heuristics in play. When fixing a threshold several factors needs to be taken into consideration (the size of the machine, the size of the jobs...). Choosing a very low value limits the scheduling policy and forces the system to a quasi-FCFS regime. A high threshold grants the scheduling policy a lot of freedom but low priority jobs risk starvation. In this work, the threshold is fixed at 2.10^5 seconds which roughly translates to 2.31 days [2].

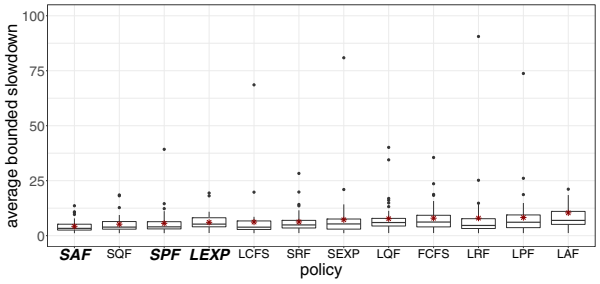
For a more detailed explanation of our choice of the threshold value please refer to Section X of the research report [17].



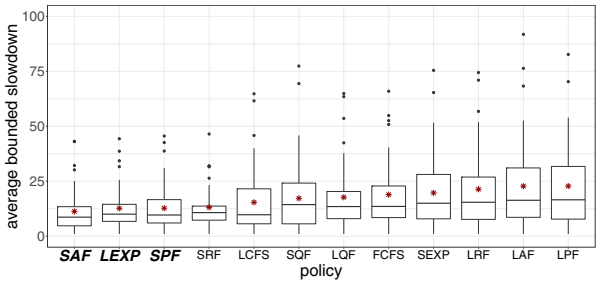
(a) SDSC-SP2



(b) SDSC-BLUE



(c) CTC-SP2



(d) KTH-SP2

Fig. 1. Tukey box-plot of the weekly average bounded slowdown of pure policies for the 4 traces. The policies are sorted in an increasing order by the mean of the weekly average bounded slowdown for all the weeks. The three most efficient policies are highlighted.

VI. Performance evaluation of pure policies

A. Comparison

We compare the pure policies presented in Section IV using the traces from Table I. We consider 45 consecutive weeks from CTC-SP2 and KTH-SP2 and 100 consecutive weeks from SDSC-SP2 and SDSC-BLUE, and we simulate the execution of all the policies for each week and measure the weekly average *BSLD* given in Equation (1).

Figure (1) illustrates the performance for all the 4 traces. The order of the policies with regard to performance changes between the traces. In general, the policies that prioritize shorter jobs, namely SAF and SPF and LEXP, are better for the average *BSLD*. SAF comes on top for all the tested traces followed by SPF and LEXP.

As expected, FCFS is not a good policy for minimizing the average *BSLD*. Although its exact position changes between traces, it always ranks among the worst policies. Interestingly, LEXP, the policy that represents the estimate of the very metric we are trying to optimize, is not the top policy, which indicates the importance of considering the amount of required resources when taking a scheduling decision.

The good performance of SAF, SPF, and LEXP can be explained by the fact that the slowdown of a job is proportional to its length. Longer jobs can wait for a longer time without having their slowdown grow drastically. The slowdown of shorter jobs, however, increases very fast the longer they wait.

B. The one size fits all policy?

From the previous comparison, we can notice that SAF is overall better than all the other tested policies to optimize the average BLSD. It gives the lowest mean on an aggregation of weeks and its outliers are not as extreme as other policies.

Figure 2 illustrates a more detailed comparison between SAF and the other policies on a given workload. We compare the average *BSLD* of SAF with the average *BSLD* of the best pure policy for every week individually. As expected, SAF performs well for most weeks. It is either the best policy or very close to the best. However, we can spot many weeks where another pure policy performs better than SAF by a significant margin (e.g. 38, 44, 56, and 85). Regardless of which policy outperformed SAF, the observation is the same for the four studied traces; SAF is good overall but it remains far from the optimal in many cases.

Finally, it is worth noting that traces where SAF fails can be found. For example, for the ANL-Intrepid trace from the Parallel Workload Archive [5], SPF is the best pure policy with an average slowdown of 35.92 while SAF ranks at 7 over 12 with 39.78. Likewise, with the Sandia trace, LAF is the best with an average slowdown of 7.353 while SAF ranks again at 7 over 12 with 10.396. We pick the traces in Table I to study due to their popularity in the literature [3], [8]. Moreover, the focus of this work is not to show that a single pure policy is dominant but to study the possibility of improving the performance of schedulers using historical data.

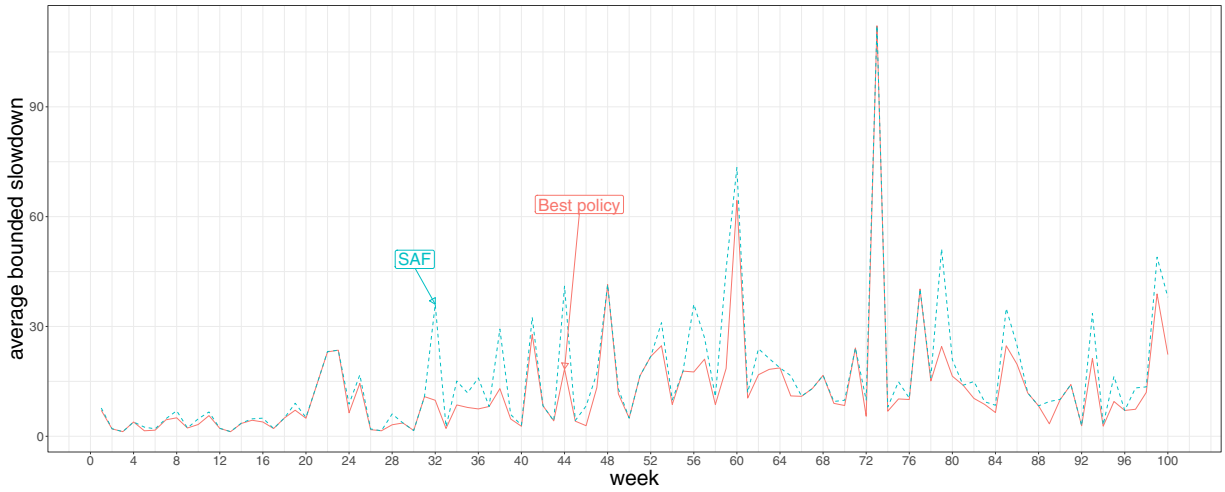


Fig. 2. Comparing SAF, the best pure policy on average, with the best pure policy for every week for the SDSC-SP2 trace.

To make the reading and the analysis easier and to avoid redundancy, all the experiments in the following sections are done using a single trace: SDSC-SP2. We refer the interested reader to Section XI of the research report [17] for a complete analysis of the other traces from which similar conclusions can be reached

VII. Scheduling using mixed policies

In the previous Section, we showed that among all the pure evaluated policies there is no single policy that is dominant across all weeks. SAF offers a reasonable compromise but it fails in many cases. This motivates the need for developing a scheduling approach that adapts to the state of the system and the workload.

In this section, for the sake clarity, we limit the mixed policies vector to only three elements: $x_j = (q_j, \tilde{p}_j, wait_j)$. Further results involving all the six features will be presented in Section VIII.

A. Performance of pure and mixed policies

We consider a set of 100 weeks from SDSC-SP2 and we separate them in the same way as in Section VI. Then, for each week we perform the following:

- Simulate using the two pure policies: (1)FCFS because of its popularity (although it is not very effective for the Average *BSLD*), and (2)SAF because, as observed in Section VI-A, it is the best policy overall.
- Generate a large number of weight by performing a uniform discretization of the search space. We take a sequence of 100 points from each dimension which can take a negative or a positive value. Thus, for three features we have $100^3.2^3 = 8.10^6$ points. Then we simulate each point and we pick the best vector *i.e.* the one that gives the lowest scores for this week, and which we denote **w***.

The results are shown in Figure 3. **w*** represents the average *BLSD* of the best weekly linear combination. The gain of **w*** compared to the pure policies varies significantly. We can classify the weeks into two types.

- Weeks where there is no or a very small difference in performance between both pure and mixed policies. The average *BSLD* of such weeks tends to be very close to 0. Weeks 43, 92 and 94 are good examples of this type. Their workload is so relaxed that no optimization is required. According to Figure 3, around half of the weeks of SDSC-SP2 belong to this type.
- Weeks where there is a difference in performance between the policies. For weeks such as 64, 73, 79, and 100, we observe significant variation in performance and a much higher *BLSD*. For this type, we also notice that **w*** is significantly better than all other policies. In week 73, for example, **w*** reduces the average *BLSD* by a substantial margin, approximately 2.5 times less than SAF, the best pure policy for that week, and 3 times less than FCFS.

Pure policies are thus far from the optimal and a carefully selected combination of features can give substantial improvement. However, the value for the best weight for each week can be quite different from the others ($\mathbf{w*}_i \neq \mathbf{w*}_j \quad \forall i \neq j \in 1..100$). This shows the changing nature of the workload through time and will be discussed in detail in Section VII-C.

B. Learning: scheduling using best combination learned from a previous part of the trace.

In this section, we evaluate the generalization capacity of our approach. We investigate how the best combination **w*** for a part of the trace performs on another part. We evaluate this ability by using two different strategies.

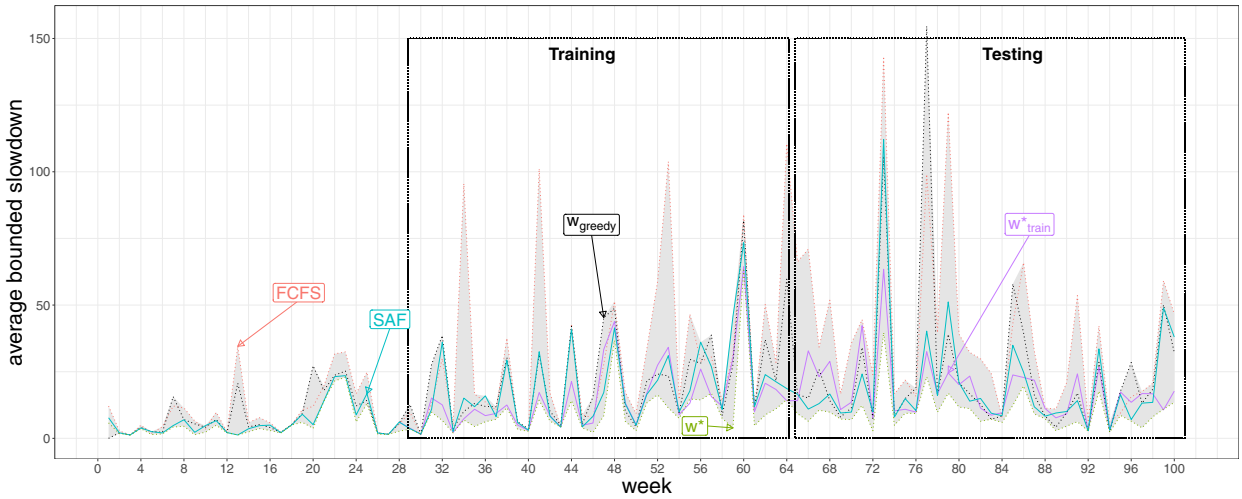


Fig. 3. Comparing the performance of various policies on the SDSC-SP2 trace. w* represents the best policy in hindsight for every week. $w*_{train}$ is the policy obtained from learning on the *Training* weeks, and w_{greedy} gives the results of testing the best policy of one week on the next.

Policy	*Training*	*Testing*
w*	376.67	357.51
$w*_{train}$	682.11	778.44
SAF	691.10	721.54
SPF	706.24	787.92
w_{greedy}	818.71	902.55
LEXP	820.94	934.21
SQF	970.49	869.41
SEXP	1016.52	1204.73
LRF	1041.18	1134.92
SRF	1147.96	1114.46
FCFS	1180.24	1398.13
LPF	1239.79	1483.35
LQF	1702.14	2191.97
LAF	2109.84	2355.16

TABLE II
COMPARING THE SUM OF THE AVERAGE *BSLD* FOR SDSC-SP2 FOR WEEKS: 65 TO 100. THE HIGHLIGHTED VALUES ARE OBTAINED IN HINDSIGHT.

1) Learning over a long period of time: The idea is to divide the trace into two equal parts and see how the best policy on the first half performs on the second.
For this particular trace, we decided to ignore the first 28 weeks because the workload at the beginning of the trace is rather light, hence all the tested policies perform similarly. So we consider the first 28 weeks as non-representative of the actual workload. Then divide the 72 remaining weeks into two parts of equal sizes. We call the first part *Training* and the second part *Testing*.

$$\mathbf{w*}_{train} = \underset{\mathbf{w}}{argmin} \sum_{week=28}^{64} average\_BSLD_{week}(\mathbf{w}) \quad (5)$$

Weeks 28 to 64 (*Training*): We aggregate using equation 5 and we find the weights $w*_{train}$ that minimizes the sum of the weekly average *BSLD* over all weeks.
Weeks 65 to 100 (*Testing*): we evaluate $w*_{train}$ on the new *Testing* weeks.

The aggregated results are illustrated in Table II and the details for each week are given in Figure 3.

Training: $w*_{train}$, the learned policy, slightly outperforms SAF in general. But if we look at individual weeks we see that SAF still has a lower *BSLD* sometimes over the training period (e.g. 34 and 52).

Testing: Table II show that $w*_{train}$ performs quite well compared to other policies. But it is still surprisingly equivalent and even outperformed by SAF.
Figure 3 shows the performance of both individual weeks. SAF is better for some weeks (namely 68, 81, and 82) but $w*_{train}$ is better for others like (e.g 73,79, 85). Sometimes both policies give similar results.

Although *Training* and *Testing* do not particularly appear as different, The best weights for *Training* are not the best for *Testing*: there is no one size fits all strategy. By comparing **w*** (see Section VII-A) and $w*_{train}$ in Figure 3, we observe that $w*_{train}$ is far from the best possible vector even for the weeks used for *Training*.

2) Learning over a short period of time: We investigate if the policy learned from one week can be effective on the next by evaluating the vector learned from week i ($\mathbf{w*}_i$) on the next week $i+1$.

In Figure 3, the policy w_{greedy} represents the results of simulating the workload of one week using the top policy from the previous week. There are unfortunately no patterns to distinguish. The vectors learned from the previous week seem to evolve and perform in a chaotic manner. Sometimes they perform better than SAF (weeks 56, 83, and 89), sometimes worse (weeks 20 and 55), and sometimes on par with SAF.

Using the policy learned from the previous week does not lead to good performance at all. We hypothesize that the structure of the workload (the jobs submitted) changes substantially from one week to the next. Thus,

online-learning the optimal weights may be very difficult.

C. Exploring the search space

In this section, we explain why there is no single vector of weights that is optimal for all cases. We visualize the search space and observe the position of the optimum for different weeks.

Figure 4 is a 2D representation of the search space for 4 consecutive weeks of the SDSC-SP2 trace. Each week is represented by two figures: the left figure displays the weekly average *BLSD* where $q \leq 0$ and the right figure, where $q \geq 0$. The $\nwarrow$ and $\nearrow$ axes respectively represent the weights of $\tilde{p}$ and $wait$. The optimal combination always lies in the lightest area and is represented by a red dot.

The coordinates of the optimal point change drastically from one week to another. Using the optimal point of week 72 to schedule week 73 give poor results because the optimal point in 72 lies in an area that has a very high slowdown in week 73. This explains why the short period learning failed.

Furthermore, with the exception of general similarities like the half where $q \geq 0$ have a lower *BSLD* than $q \leq 0$, we also observe that the position, shape, and even the size of the optimal area changes radically from one week to the next. This explains why online learning seems compromised without further information.

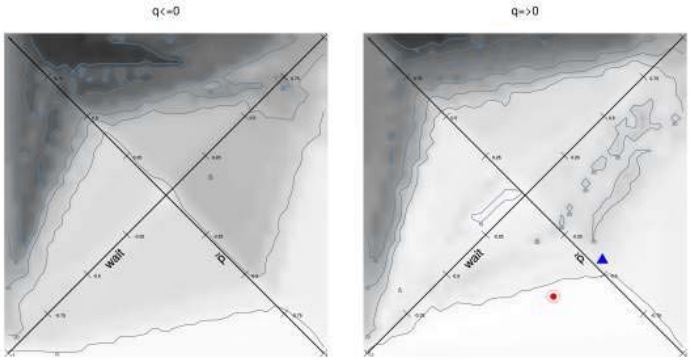
VIII. Increasing the size of the search space: using more jobs characteristics

In this section, we investigate the impact of using all six job characteristics on performance. Indeed, the experiments in all the previous sections were done with only the three basic job characteristics: p,q, and $wait$. In this Section we extend the search space to include the three other characteristics introduced in Section IV-A which are a,r,exp.

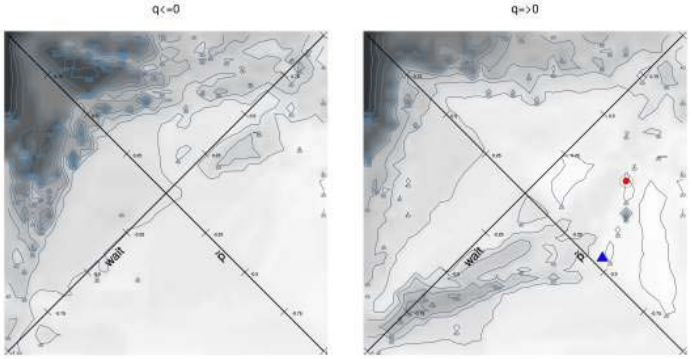
A. Black-box optimizers: a quick way to find the optimal

1) Algorithm: In the previous Section, finding the weekly best mixed policy was done using a uniformly "exhaustive" search. We made a fine discretization of the whole search space and we selected the weight vector **w*** that provides the lowest average $BLSD$. Performing an exhaustive space search becomes costly very fast because the size of the search space grows exponentially with the number of job characteristics we include in the linear combination. Thus another method to find the minimum is required.

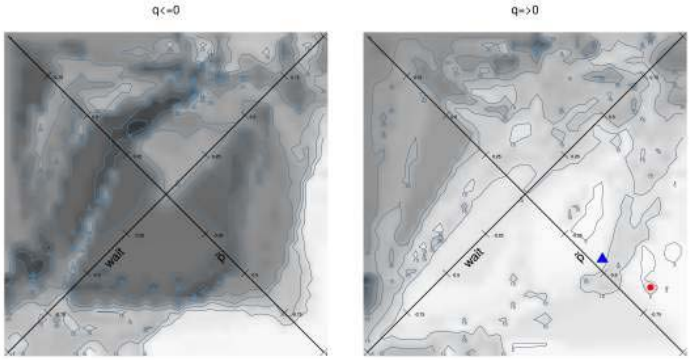
Our goal is to find a combination of weights **w*** that minimize equation (4) while enforcing the constraint



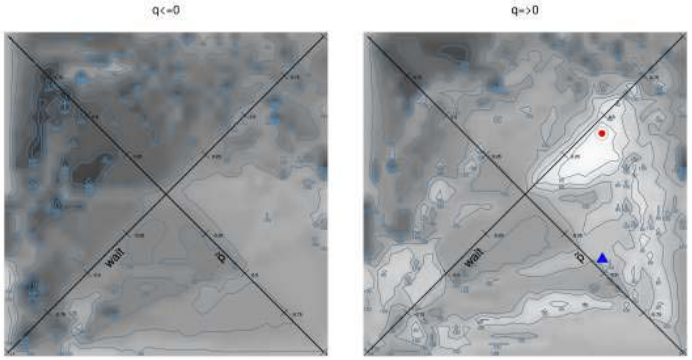
(a) week 70: w*$(q = 0.30, \tilde{p} = 0.37, wait = -0.33)$



(b) week 71: w*$(q = 0.27, \tilde{p} = 0.35, wait = 0.38)$



(c) week 72: w*$(q = 0.32, \tilde{p} = 0.62, wait = 0.06)$



(d) week 73: w*$(q = 0.47, \tilde{p} = 0.05, wait = 0.48)$

Fig. 4. Visualization of the search space for 4 consecutive weeks 70, 71,72, and 73. The two diagonal axis represent $\tilde{p}$ and $wait$. The lighter the area is, the better the performance (lower average *BSLD*). The optimal area change from one week to the next. The red dot (in the lightest area) represents $w*$ and the blue triangle represents $\mathrm{w}*_{train}$.

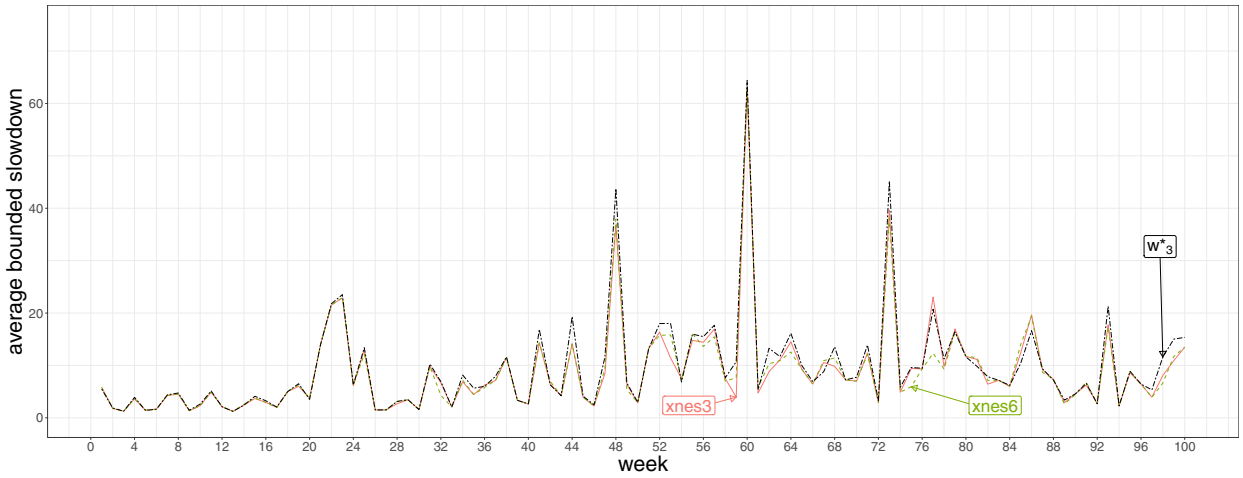


Fig. 5. Comparing average *BSLD* of the vectors of the 3 original features (xnes3) with the extended vector of 6 features (xnes6) and the minimum we obtain from space coverage (w*$_3$)

$||w||_1 = 1$. This can easily be done by optimizing the following objective function:

$$\sum_{j=1}^{n} F(score(\mathbf{w}, x_j)) + \lambda \left(||\mathbf{w}||_1 + \frac{1}{||\mathbf{w}||_1} \right) \quad (6)$$

Function F has *a priori* no particular properties. Furthermore, we have seen in Section VII-C that the search space is not convex and it may exhibit several local minima. Therefore, gradient-based methods cannot be used and we have to rely on stochastic derivative-free methods. We initially tried the standard simulated annealing method [18] but it got frequently stuck in local optimums. A study of the existing literature [19] led us to the evolutionary algorithms family that considers an ensemble of candidates. We tested several algorithms, Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [20] and eXponential Natural Evolutionary Strategy (XNES) [21] provided the best results. Since XNES is faster than CMA-ES, we chose the former.

2) Performance: For each week we apply the XNES algorithm to obtain a solution of Equation (6) for a vector of dimension 3 (*xnes3*) and a vector of dimension 6 (*xnes6*) and we compare the results with the minimum obtained from the space coverage which we call w*$_3$ (corresponds to the **w*** used in SectionVII-A). Figure 5 illustrates the results.

For most weeks *xnes3* and w*$_3$ give the same result. For few other weeks, *xnes3* managed to slightly outperform w*$_3$. This is due to the method used to cover the search space: Each dimension of the vector gets 200 points distributed uniformly over [-1,1]. *XNES* does not have that constraint, hence it can produce policies that are more *"refined"*. The differences in performance are minor which indicate that *XNES* managed to find a vector that is the actual or at least very close to the optimum every time. Thus *XNES* can be considered as a viable option to find an optimal vector.

The *BSLD* of *xnes3* and *xnes6* are not very different from each other. For most of the weeks, both vectors perform equally. In some rare cases (weeks 86 for example), *xnes3* gives a slightly better performance than *xnes6* but the difference is marginal (*XNES* converged to a local optimum instead of the global optimum in the case of 6). On average *xnes6* is better than *xnes3* but not by a larger margin.

Increasing the size of the search space by adding job characteristics improves the results by a small very margin.

IX. Conclusion

Scheduling parallel jobs in a real HPC platform is a complex task plagued with many uncertainties. Determining an efficient scheduling strategy is difficult due to the volatile nature of the workload. The main result of this work was to optimize the EASY-Backfilling algorithm by reordering the primary queue using policies learned from historical data.

More precisely, we first showed that SAF (Smallest estimated Area First) performs overall better than more popular policies FCFS and SPF. Then, we looked at the scheduling problem from a new perspective by studying a larger class of heuristics obtained from mixed policies that enable us to move from a discrete to a continuous search space. We combined several characteristics extracted from the jobs in a linear expression and we determined the best weight for each characteristic.
We showed, moreover, that pure policies are far from the optimal and that important gains can be obtained by using mixed policies. For some weeks in the simulation, we obtained results that are up to 3 times better than the best pure policy. Unfortunately, we observed that the

structure of the workload changes too much over time and that whenever a policy performs well on a part of a trace, it does not mean necessarily that it will be efficient on another part of the trace.

Using historical data to predict good scheduling policies for future jobs is not a straightforward task. We observed that the workload itself changes drastically from one time period to the next. We have yet to identify any meaningful pattern to these changes, which raises the question of whether it is possible to apply machine learning on real execution logs or not.

Choosing a proper metric to evaluate the performance of a policy in an online scheduling context is also an interesting (but hard) task. In particular, the average *BSLD* may be enriched since it does not consider the required resource into consideration.

X. Acknowledgment

The authors are sorted in an alphabetical order.

The authors would like to thank Valentin Reis for his key contribution to the origin of this work and for providing the simulator used throughout the experimental campaign. We are also grateful for Danilo Carastan-Santos and Millian Poquet for their help and comments. 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 and Steve Hotovy (CTC SP2). This work was partially supported by the French research agency (Energumen ANR-18-CE25-0008)

References

[1] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Backfilling using system-generated predictions rather than user runtime estimates," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 6, pp. 789–803, Jun. 2007. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2007.70606

[2] 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, vol. 10773. Springer, May 2017, pp. 43–61. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01522459

[3] 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. ACM, 2015, pp. 64:1–64:10. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807646

[4] A. W. Mu'alem and D. G. Feitelson, "Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 6, pp. 529–543, Jun. 2001. [Online]. Available: https://doi.org/10.1109/71.932708

[5] D. G. Feitelson, "Parallel workload archive," http://www.cs.huji.ac.il/labs/parallel/workload/, 2008, [Online; accessed October-2018].

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

[7] D. G. Feitelson, "Metrics for parallel job scheduling and their convergence," in *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, Eds. Springer, 2001, pp. 188–205.

[8] 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

[9] D. Perkovic and P. J. Keleher, "Randomization, speculation, and adaptation in batch schedulers," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=370049.370063

[10] K. Aida, "Effect of job size characteristics on job scheduling performance," in *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, ser. IPDPS '00/JSSPP '00. Springer-Verlag, 2000, pp. 1–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=646381.689680

[11] N. Papadopoulou, G. I. Goumas, and N. Koziris, "A machine-learning approach for communication prediction of large-scale applications," in *CLUSTER*, 2015, pp. 120–123.

[12] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Modeling user runtime estimates," in *Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing*, ser. JSSPP'05. Springer-Verlag, 2005, pp. 1–35. [Online]. Available: http://dx.doi.org/10.1007/11605300_1

[13] O. Tuncer, E. Ates, Y. Zhang, A. Turk, J. M. Brandt, V. J. Leung, M. Egele, and A. K. Coskun, "Diagnosing performance variations in hpc applications using machine learning," in *ISC*, 2017.

[14] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. ACM, 2016, pp. 50–56. [Online]. Available: http://doi.acm.org/10.1145/3005745.3005750

[15] V. C. Stodden, F. Leisch, and R. D. Peng, *Implementing Reproducible Research*, V. Stodden, F. Leisch, and R. D. Peng, Eds. CRC Press, 2014. [Online]. Available: http://www.crcpress.com/product/isbn/9781466561595

[16] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 583–592. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999463

[17] A. Legrand, D. Trystram, and S. Zrigui, "Adapting Batch Scheduling to Workload Characteristics: What can we expect From Online Learning?" Grenoble 1 UGA - Université Grenoble Alpe, Research Report RR-9212, Oct. 2018. [Online]. Available: https://hal.inria.fr/hal-01896121

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[19] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing*, vol. 8, no. 2, pp. 239–287, Jun 2009. [Online]. Available: https://doi.org/10.1007/s11047-008-9098-4

[20] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*, J. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea, Eds. Springer, 2006, pp. 75–102.

[21] T. Glasmachers, T. Schaul, S. Yi, D. Wierstra, and J. Schmidhuber, "Exponential natural evolution strategies," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '10. ACM, 2010, pp. 393–400. [Online]. Available: http://doi.acm.org/10.1145/1830483.1830557