Abstract—The EASY-FCFS heuristic is the basic building block of job scheduling policies in most parallel High Performance Computing platforms. Despite its simplicity, and the guarantee of no job starvation, it could still be improved on a per-system basis. Such tuning is difficult because of non-linearities in the scheduling process. The study conducted in this paper considers an online approach to the automatic tuning of the EASY heuristic for HPC platforms. More precisely, we consider the problem of selecting a reordering policy for the job queue under several feedback modes. We show via a comprehensive experimental validation on actual logs that periodic simulation of historical data can be used to recover existing *in-hindsight* results that allow to divide the average waiting time by almost 2. This results holds even when the simulator results are noisy. Moreover, we show that good performances can still be obtained without a simulator, under what is called bandit feedback - when we can only observe the performance of the algorithm that was picked on the live system. Indeed, a simple multi-armed bandit algorithm can reduce the average waiting time by 40%.

Online Tuning of EASY-Backfilling using Queue Reordering Policies.

Eric Gaussier
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
eric.gaussier@imag.fr
Jérôme Lelong
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK
jerome.lelong@imag.fr
Valentin Reis, Denis Trystram
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
denis.trystram@imag.fr

1 Introduction

1.1 Context

Providing the computational infrastuctures needed to solve actual complex problems arising in the various fields of modern society (including climate change, health, green energy or security) is a strategic challenge. The main pillar to adress this problem is to build extreme-scale parallel and distributed platforms. The never-ending race for more computing power and storage capacity leads to sophisticated specific Exascale platforms, and the objective of the community is to design efficient sustained Petascale platforms. This large-scale evolution and the increasing complexity in both architecture and applications create many scientific and technical problems. Accordingly, system management software still has a long way to go. The existing job and resource management software allow to run tens of thousands of jobs on hundreds of thousands of cores. Such software are based on robust policies with positive properties such as relative simplicity and prevention of job starvation. However, there is still room for improvement. Here, we study how to tune the ordering of the submission queues in the frame of the classical so-called EASY[1]-Backfilling family of heuristics using two online approaches. One approach is based on a simulator and the second uses a multi-armed bandit algorithm.

Most common work in the literature consider *a posteriori* optimization of a scheduling algorithm on a (actual) data set. The increasing amount and diversity of data generated by large scale platforms motivate the community to study more sophisticated policies that could leverage these data. During the last few years, there was an explosion in the amount of work at the interface of the HPC and Big Data fields, dealing with learning algorithms. Recent sophisticated approaches developed *ad hoc* supervised machine learning algorithms for reducing uncertainty in problem parameters (e.g., job execution times [22], memory requirements, etc.). However, only few studies evaluate the impact of such techniques on the performances of the resource manager.

Unlike most existing studies, which consider the learning of specific parameters of a trace, we provide algorithms that use feedback from the system in order to alter the scheduling behavior online. Our focus is to learn how the platform behaves within the usual framework of EASY-Backfilling described in Section 2 . More precisely, we provide strategies for choosing the most appropriate queue reordering policy to be used for job selection. This approach is assessed by an extensive resampling-based experimental validation that uses 7 actual traces.

1.2 Contributions

Our main contribution is to investigate two methods for learning a good scheduling policy on a per-system basis. In this work, we focus on the average waiting time of the jobs. Each method aims at choosing the best queue reordering policy in a fixed search set of semantically diverse options.

The first method, **full feedback**, continuously uses simulation on data gathered from the past and current system workload. In order to reflect the uncertainties in the data, we also evaluate a more realistic "noisy" variant in which simulations are imprecise.

The second method, **bandit feedback**, is a simulator-less approach based on a multi-armed bandit algorithm, which derives feedback by measuring system performance.

Both of these methods are evaluated with a specifically developed simulator, which is open-sourced [28].

Our main results are as follows: The approach based on the simulator provides very good results, close to the best policy in the search set. In essence, we are able to recover in an online fashion existing results that reduce

1. Extensible Argonne Scheduling sYstem (EASY)

average waiting times by 11% to 60% compared to EASY-FCFS. Indeed, we show that the required sample sizes for policy selection in batch schedulers are small enough to avoid workload resampling in practice. The variant based on noisy data, which is closer to the actual conditions of uncertainties on both jobs and platform, behaves similarly. The alternate approach based on a multi-armed bandit also provides reasonable results at a much lower cost, reducing the average waiting times by 8% to 46% compared to EASY-FCFS.

Let us emphasize the main results of this study. We show how to select a good alternative to the First-Come, First-Serve policy in an online manner. We present two automated methods for selecting a good alternative policy in two cases (depending on whether a simulator is available or not), and provide the difference in performance to be expected between both situations.

2 Related Work

This section presents and discusses the most significant results related to job scheduling and learning algorithms in this context. Let us start by recalling some basics results on heuristics in HPC platforms:

Parallel job scheduling is an old studied theoretical problem ([21], [19]) whose practical ramifications, varying hypotheses, and inherent uncertainty of the problem applied to the HPC field have driven practitioners to use simple heuristics (and researchers to study their behavior). The two most popular heuristics for HPC platforms are EASY [31] and Conservative [25] backfilling.

While Conservative Backfilling offers many advantages [32], it has a significant computational overhead and is more complex, which mainly explains why many of the machines of the top500 ranking [38] rather use a variant of EASY-Backfilling instead. There is a large body of work seeking to improve/tune EASY. Indeed, while the basic mechanism is used by some actual resource and job management software (most notably SLURM [30]), this is seldom done without fine tuning by system administrators.

The original EASY mechanism refers to a First-Come-First-Serve basis. Several works explore how to tune EASY by reordering waiting and/or backfilling queues [39], sometimes even in a randomized manner [27]; some implementations use this method as well [23]. However, as successful as they may be, these works do not address the dependency of scheduling metrics on the workload [2]. These studies most often report *post-hoc* performance since they compare algorithms after the workload is known.

The dynP scheduler [34] suggests a systematic method to tuning these queues. The algorithm requires simulated scheduling runs at decision time, therefore the time-to-decision of the scheduling is higher than that of EASY. The adaptation of EASY was studied under a different framework than ours in [35]. Other works study the reservation mechanism [33]. Online portfolio approaches are also studied for the cloud setting [10].

2.1 Data-aware Resource Management.

There was a recent focus on leveraging the huge amount of data available in large scale computing platforms to improve system performance. Some works use collaborative filtering to co-locate tasks in clouds by estimating application interference [41]. Some others are closer to the application level. For instance, [42] uses binary classification to distinguish benign memory faults from application errors so as to execute recovery algorithms.

Several works use similar techniques in the context of HPC, in particular [39], [22], hoping that better job runtime estimations should improve the scheduling [9]. Some algorithms estimate runtime distributions model and choose jobs using probabilistic integration procedures [26]. However, these works do not address the duality between the cumulative and maximal scheduling costs, as mentioned in [22]. While these previous works intend to estimate uncertain parameters, we consider here a more pragmatic approach, that directly learns a good scheduling policy from a given policy space.

Existing work [24] takes this approach offline, by splitting workload data in a training/testing fashion. This work studies EASY-Backfilling and shows that while the relative performance of some well-known priority orders for starting jobs differs between workloads, it is relatively stable over time for a given workload.

2.2 Multi-Armed Bandits.

A multi-armed bandit (MAB) problem is a sequential allocation problem with partially observable rewards [7]. At every round an action, called an "arm" in the literature, must be chosen from a fixed set and the corresponding reward is observed. The goal of a MAB algorithm is to maximize the total reward obtained in a successive number of rounds. This is essentially achieved using a combination of explorative actions, which help estimate the quality of each arm, and of exploitative actions, which leverage the current estimates by using a good arm. This problem can be addressed using various hypotheses. Most common are the (original) stochastic setting and the adversarial setting of regret minimization. See [36] for the original work on the stochastic setting and [3] for the "Upper Confidence Bound" family of algorithms. To the best of our knowledge, the work of [5] is the earliest for the adversarial setting; see [4] for the "Exponential-weight" family of algorithms. We refer to [7] for a comprehensive review of the field. While these algorithms bound the cumulative difference in loss to the best arm (the regret), they have functional constraints such as the fact that the rewards should be contained in a range. The simple heuristic called *Epsilon-Greedy* introduced in [3] does not have this requirement. For this reason, this heuristic will be used in this paper and is described in Subsection 5.3.

3 Problem Setting

3.1 System Description

The crucial part of batch scheduling software is the scheduling algorithm that determines where and when the submitted jobs are executed. The process is as follows: jobs are submitted by end-users and queued until the scheduler selects one of them for running. Each job has a provided bound on the execution time and some resource requirements (number and type of processing units). Then, the

Resource and Job Management System (RJMS) drives the search for the resources required to execute this job. Finally, the tasks of the job are assigned to the chosen nodes.

In the classical case, the management software needs to execute a set of concurrent parallel jobs with rigid (known and fixed) resource requirements on an HPC platform represented by a pool of m identical resources. This is an online problem since the jobs are submitted over time and their characteristics are only known when they are released. Below is the brief description of the characteristics of job j:

- Submission date r_j (also called *release date*)
- Resource requirement q_j (number of processors)
- Actual running time p_j (sometimes called *processing time*)
- Requested running time $\widetilde{p_j}$ (sometimes called *walltime*), which is in general an upper bound of p_j.

The resource requirement q_j of job j is known when the job is submitted at time r_j, while the requested running time $\widetilde{p_j}$ is given by the user as an estimate. Its actual value p_j is only known *a posteriori* when the job really completes. Moreover, the users have incentive to over-estimate the actual values, since jobs may be "killed" if they surpass the provided value.

3.2 Brief Description of EASY Backfilling

The selection of the job to run is performed according to a scheduling policy, which determines the order the jobs are executed. EASY-Backfilling is the most widely used policy due to its simple and robust implementation and known benefits such as high system utilization [31]. This strategy has no worst case guarantee beyond the absence of starvation (i.e. every job will be scheduled at some moment).

EASY-FCFS uses a job queue to select and backfill jobs. At any time a scheduling decision is required (i.e. job submission or termination), the scheduler goes through the job queue in First-Come,First-Serve (FCFS) order and starts jobs until it finds a job that can not be started right away. Then, it makes a reservation for this job at the earliest predictable time and starts *backfilling* the job queue in FCFS order using any job that does not delay the unique reservation.

3.3 Scheduling Objective

A system administrator may use one or multiple cost metric(s). Our study of scheduling performance relies on the waiting times of the jobs, which is one of the more commonly used objectives. The waiting time of a job is

$$\mathbf{Wait}_j = start_j - r_j \tag{1}$$

where $start_j$ is the starting time of a job. Like other cost metrics, the waiting time is usually considered in its *cumulative* version, which means that one seeks to minimize the average waiting time. It is worth noting that other metrics such as the maximum waiting time of all the jobs are also worthy of interest. Unfortunately, this criterion is antagonistic in nature with the average waiting time. Section 4.2 will outline our approach to address the bi-objective aspect of this problem.

4 Tuning EASY by Reordering and Thresholding the Job Queue

This section presents two mechanisms for safely tuning the EASY-Backfilling: job queue reordering and job thresholding. Together, these two building blocks make a robust framework for tuning EASY.

4.1 Reordering

The EASY heuristic uses a job queue to select and backfill jobs. While this job queue is ordered in FCFS order in the original heuristic, it is possible to reorder it at will. We focus on a search space of 12 reordering policies.

1) FCFS: First-Come First-Serve, which is the most commonly used policy [31].
2) LCFS: Last-Come First-Serve.
3) SPF: Smallest estimated Processing time $\widetilde{p_j}$ First [32].
4) LPF: Longest estimated Processing time First.
5) LQF: Largest resource requirement q_j First.
6) SQF: Smallest resource requirement First.
7) LEXP: Largest Expansion Factor First [32], where the expansion factor is defined as follows:
$$\frac{wait_j + \widetilde{p_j}}{\widetilde{p_j}} \tag{2}$$
where $wait_j$ is the time job j has been waiting until the time at which the decision is taken. This corresponds the the value of the stretch(or slowdown) where the user runtime estimate is used.
8) SEXP: Smallest Expansion Factor First
9) LRF: Largest Ratio $\frac{\widetilde{p_j}}{q_j}$ First
10) SRF: Smallest Ratio First
11) LAF: Largest Area $\widetilde{p_j} \times q_j$ First (often called "Total Resources")
12) SAF: Smallest Area First

This search space is designed with the goal of being as semantically diverse as possible without making any judgement on which policy should perform well in practice. It does include most policies from related works that we are aware of. In the following, we denote these policies by P_i with $i = 1 \dots 12$.

4.2 Thresholding

Reordering the job queue means losing the no-starvation guarantee; some individual jobs therefore can wait an undue amount of time. It is possible to introduce a thresholding mechanism to prevent this behavior: When a queued job's waiting time exceeds a fixed threshold Θ, it is at the head of the queue. We denote by EASY(P, Θ) the scheduling policy that starts and backfill jobs according to the (thresholded) reordering policy P. For the sake of completeness, Algorithm 1 describes the EASY(P, Θ) heuristic.

5 Online Tuning

Here, we present our policy selection strategies. We will refer to the period during which a selected policy is applied as the *policy period* and will denote the length of this period

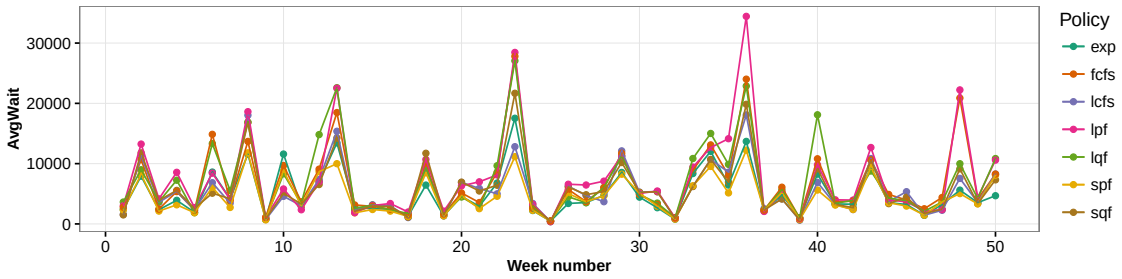

Fig. 1. Variability in the weekly average waiting time in the KTH-SP2 trace (see Subsection 6.1.1) for 7 different policies with $\Theta = 0$. The policy set is reduced as not to obstruct the figure with too many colors.

Algorithm 1 EASY(P, Θ) policy
Input: Queue Q of waiting jobs.
Output: None (calls to $Start()$)
1: Sort Q according to P
2: Move all jobs of Q for which $wait_j > \Theta$ ahead of the queue (breaking ties in FCFS order).
//Starting jobs until the machine is full
3: **for** job j in Q **do**
4: **if** j can be started given the current system use **then**
5: Pop j from Q
6: $Start(j)$
7: **else**
8: Reserve j at the earliest time possible according to the estimated running times of the currently running jobs.
//Backfilling jobs
9: **for** job j' in $Q \setminus \{j\}$ **do**
10: **if** j' can be started without delaying the reservation on j. **then**
11: Pop j' from Q
12: $Start(j')$
13: **end if**
14: **end for**
15: **break**
16: **end if**
17: **end for**

as Δ. The time interval is thus divided into periods of equal lengths (Δ): $\Delta_0, \cdots, \Delta_T$, where T is the index of the current policy period. A new policy is selected at the beginning of each period and applied during the whole period.

We further assume that there is a certain regularity among periods, i.e. that the distribution of the job submission process does not radically differ between consecutive weekly periods. This assumption is validated in the study presented in [24]. It further entails that the behavior of a policy during the previous periods reflects its behavior on the current one, so that the selection of a policy can be based on its past behavior. However, there may be some variability between different periods for certain cost metrics [15]. This is illustrated in Figure 1 which displays the average waiting time of various policies for the KTH-SP2 trace (see 6.1.1 for a description of the workloads) using weekly periods encompassing one year. As one can note, the average waiting time varies a lot from one week to the other, for all 7 policies considered. This indicates that when the cost metric is averaged over different periods, there is a trade-off to find in between longer periods that would somehow limit the variability, and shorter ones that yield more values for the estimation. We will come back to this issue below.

The selection of a policy is reminiscent of reinforcement learning, where solving a search and inference problem in a (perhaps restrictive) policy space is easier than in the space of the original problem. It is important to note, however, that a pure reinforcement learning approach is difficult to develop in our context. Indeed, while we have outlined a reduced action space, the state space to consider is infinite. This is not prohibitive *per se*, as modern methods [8] can bypass dimensionality issues via function approximation. However, these methods rely on large amounts of batch data, and we are facing an online problem with no pre-existing data[2]. We rely here on simpler, yet we believe more effective, strategies to solve this problem. These strategies are applied online and rely on exact simulation, noisy simulation and ϵ-greedy bandit exploration.

5.1 Policy Selection with Exact Simulation

Several simulators have been developed for "playing" reordering policies on a given set of jobs. We rely in this study on a lightweight simulator (see Section 6) that can efficiently simulate different policies. Such simulators are interesting inasmuch as they provide an estimate of the cost of a given policy on a set of jobs, as described below.

Let $l(\Delta_t)$ denote the number of jobs *submitted* during the period Δ_t ($0 \leq t$), and P_i ($1 \leq i \leq 12$) one reordering policy (defined in Section 4.1). The cost of policy P_i during period Δ_t is defined as:

$$w(t; P_i) = \sum_{j=1}^{l(\Delta_t)} \text{Wait}_j^{P_i} \tag{3}$$

where $\text{Wait}_j^{P_i}$ denotes the estimate provided by the simulator of the waiting time for job j according to policy P_i. The estimation of the cost of a policy over all the periods preceding the current period can then be defined as:

$$w(\rightarrow T; P_i) = \sum_{t=0}^{T-1} \lambda^{T-1-t} w(t; P_i) \tag{4}$$

where $\lambda \in [0, 1]$ is a decaying discount factor that can be used to privilege recent history (*i.e.* recent periods). The

2. The reinforcement learning literature uses the term "*on-policy*" for this setting.

above formulation is general and covers several possible situations : From stationary job distributions where λ can be set to 1 to non-stationary distributions where the more recent history has to be privileged. As explained in Section 6.1.3, in our experiments λ is set to 1 as the distributions considered are stationary but the reader has to bear in mind that our framework can be use on different distributions by resorting to different values of λ that can be set by cross-validation.

One then selects the policy P that minimizes the above cost for the period Δ_T:

$$P = \text{argmin}_{P_i, 1 \leq i \leq 12} w(\rightarrow T; P_i) \tag{5}$$

The above cost directly corresponds to the cumulative waiting time of the policy over the preceding periods (more precisely to the cumulative simulated estimate of the waiting time of the policy over the preceding periods) so that the length of the period has little impact here. The only bias comes from the boundary states of the simulation. Indeed, we run simulations from an empty system and wait for the system to be empty when job submissions cease at the end of the period (see Figure 2). Note that starting the simulation using the system state at the end of the previous period should improve the results. Here it will not be necessary hence we keep the simple version with an empty system. However, it may prove useful in the non-stationary case mentionned above.

Furthermore, in the context of this study and as detailed in Section 6, we rely on averages over several execution traces in order to obtain reliable estimate of the behavior of different policies and policy selection strategies. Such traces are typically obtained by simulation. Using the same simulator for generating the traces and estimating the cost as defined in Eqs (3) and (4) would however be too optimistic and would represent an upper bound on what can be achieved by a selection strategy based on simulation.

In order to have a more realistic estimate of the behavior of a selection strategy based on simulation, we introduce noise in the simulator, as described below.

5.2 Policy Selection with Noisy Simulation

We take another step to evaluate how the simulation strategy for selecting policies would work in a batch scheduler. Simulation of real systems can be imprecise. While existing work reports values as low as 2% imprecision we take a conservative approach and consider imprecision up to of 20%. We randomly introduce multiplicative noise in the estimate of the policy cost defined by Eq. (3) by rescaling it, either down or up:

$$w_{\text{noisy}}(t; P_i) = \rho_t^i \sum_{j=1}^{l(\Delta_t)} \text{Wait}_j^{P_i} \tag{6}$$

where ρ_t^i is uniformly sampled in the interval $[0.80, 1.20]$, thus adding a +/- 20% multiplicative noise on the waiting time estimated by the simulator. This value is taken as to be sufficiently larger than values previously reported, see e.g. [12] which reports 2 percent variations of the simulated metric compared to the metric on the real systems. The overall cost is then defined in the same way as before, leading to:

$$w_{\text{noisy}}(\rightarrow T; P_i) = \sum_{t=0}^{T-1} \lambda^{T-1-t} w_{\text{noisy}}(t; P_i) \tag{7}$$

As before the policy that minimizes the above cost is selected for the period Δ_T.

$$P = \text{argmin}_{P_i, 1 \leq i \leq 12} w_{\text{noisy}}(\rightarrow T; P_i) \tag{8}$$

Instead of introducing noise in the output of the simulator when selecting the policy, one could have introduced it when collecting the execution traces. Our choice to add the noise after the simulation step is motivated by simplicity. As we will see in Section 6, there is little difference between the two selection methods, which is an important result of our study.

Lastly, as before, the length of the period has little impact here as the cost metric corresponds to the cumulative simulated waiting time for each policy.

5.3 Policy Selection with ϵ-greedy Exploration

The previous strategies estimate the cost of all policies over all previous periods; the best policy according to this cost is then selected for the current period, the costs of all policies being updated for the next period. This is interesting as one maintains a complete knowledge of all policies over time. However, this requires computing many estimates at each period (as many as there are policies), which can be time consuming or cumbersome even with lightweight simulators.

Thus, we explore here a more efficient strategy that dispenses with estimating the cost of all policies at a given time and only updates the cost of the policy that is currently being used. This strategy makes use of the ϵ-greedy exploration method, standard in reinforcement learning [43] and bandit problems [3], to trade-off exploitation and exploration, and relies, as before, on past estimates of the cost to select the best policy in the exploitation mode.

Let $l(\Delta_t^{'})$ denote this time the number of jobs *finished* during the period $\Delta_t^{'}$. We define the cost of policy P_i during period $\Delta_t^{'}$ as:

$$w_\epsilon(t; P_i) = \begin{cases} \sum_{j=1}^{l(\Delta_t^{'})} \textbf{Wait}_j^{P_i} & \text{if } P_i \text{ used during } \Delta_t^{'} \\ 0 & \text{otherwise} \end{cases}$$

As one can note, the actual waiting time is used here, so that one dispenses with the use of a simulator for efficiency considerations. This leads to a faster and easier estimate of the cumulative waiting time, however only for the policy that is used during $\Delta_t^{'}$. Note that there is as previously a bias due to boundary effects in the measurement method, as we do include jobs that had been started before $\Delta_t^{'}$, as well as disregard jobs that were submitted during $\Delta_t^{'}$ but finish later.

The cost over all previous period of a policy P_i is then defined as:

$$w_{\text{epsilon}}(\rightarrow T; P_i) = \frac{1}{\sum_{t=0}^{T-1} \mathbb{1}(P_i, t) l(\Delta_t^{'})} \sum_{t=0}^{T-1} \mathbb{1}(P_i, t) \lambda^{T-1-t} w_\epsilon(t; P_i) \tag{9}$$

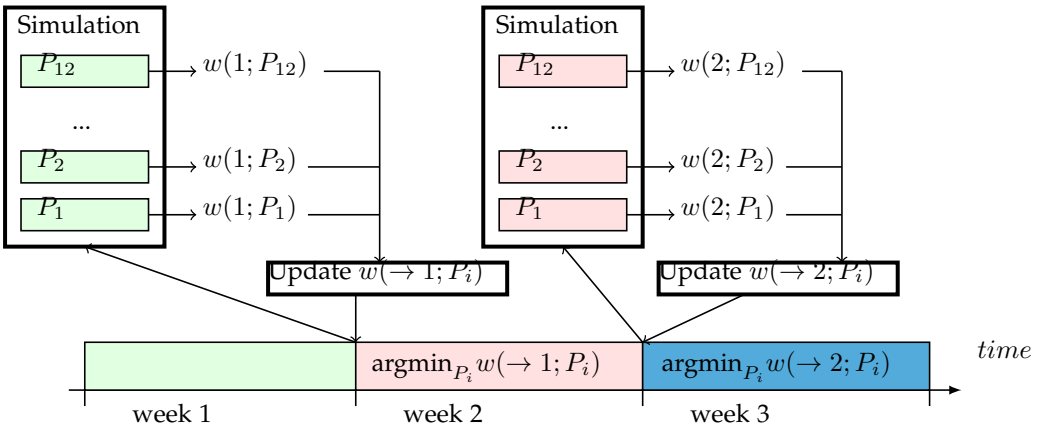

Fig. 2. Illustration of the online simulation-based strategy. Every week, the workload from the past week is used to run simulations using every reordering policy. The "model" w is updated according to (4) (or (7) in the Noisy case from Section 5.2) and the policy for the next week is chosen according to (5)

where $\mathbb{1}(P_i,t) = 1$ if P_i is the policy used during the period $\Delta_t^{'}$ and 0 otherwise. The normalization by $\sum_{t=0}^{T-1} \mathbb{1}(P_i,t)l(\Delta_t^{'})$ is necessary to ensure that policies remain comparable over time. Not normalizing would favor the policy that was first selected, even if this choice was random. This normalization however entails that the size of the policy period is important: it should not be too large so as to avoid relying on too few points for estimating the cost, and it should not be too small either to avoid extreme variations between periods. As explained in Section 6, we rely in this study on periods of one day and one week.

The selection of the policy to be used during the current time period Δ_T is then based on the standard ϵ-greedy exploration strategy:

- With probability $(1-\epsilon)$, select the policy P that minimizes the cost over previous time periods (exploitation mode):

$$P = \text{argmin}_{P_i, 1\leq i\leq 12} w_{\text{epsilon}}(\rightarrow T; P_i) \qquad (10)$$

- With probability ϵ, select a policy P_i, $1 \leq i \leq 12$, at random (exploration mode).

Figure 3 illustrates this process. Several studies study the decay of ϵ over time, the exploration being less important once the estimates for the different policies are reliable [37]. This strategy has however not been beneficial in our case. We believe this is due to the properties of the traces we use, as explained in Section 6. The ϵ-greedy strategy is known to be outperformed in terms of regret by other classical algorithms such as the UCB or EXP family of algorithms (see [7]). See [7] for a precise definition and thorough treatment of this subject. These algorithms however all rely on pre-normalized rewards. Here, the scale of $w_\epsilon(t;P_i)$ is not known in advance. While the usual doubling tricks can be devised to go around sequential issues of scale with good asymptotical performance, they ruin performance in practice. Accordingly, we only perform experiments using the simple ϵ-greedy method.

6 Experiments

This section compares the different approaches to policy selection via a comprehensive experimental validation. Subsection 6.1 outlines our experimental protocol and Subsection 6.3 contains the experimental results and their discussion.

TABLE 1
Workload logs used in the simulations.

Name	Year	# MaxProcs	# Jobs	Duration
KTH-SP2	1996	100	28k	11 Months
CTC-SP2	1996	338	77k	11 Months
SDSC-SP2	2000	128	59k	24 Months
SDSC-BLUE	2003	1,152	243k	32 Months
ANL-Intrepid	2009	163,840	68k	9 Months
CEA-Curie	2012	80,640	312k	3 Months
Unilu-Gaia	2014	2,004	51k	4 Months

6.1 Experimental Protocol

The evaluation of computer systems performance is a complex task. Here, the two main difficulties are choosing a simulation model and taking into account the variability in the average waiting times. This section presents our approach, which is based on lightweight simulation and trace resampling.

6.1.1 Traces

The experimental validation performed in this work uses a mixture of small and large systems from different periods. These platforms are all targeted by this work, since they all are homogenous systems that run a batch scheduler. Table 1 presents the 7 logs, which can all be obtained from the Parallel Workload Archive [20]. We use the "cleaned" [14] version of the logs as per the archive. We impose an additional filtering step to the workloads[3] in order to clean erroneous data:

1) If the number of allocated or requested cores of a job exceeds the size of the machine, we remove the job;

3. This filtering step is available in the reproducible workflow [29] as shell script `misc/strong_filter`

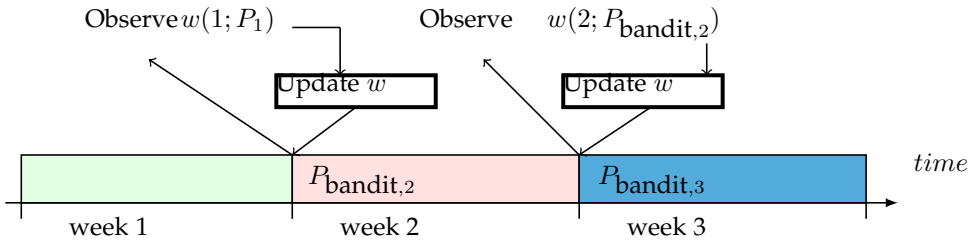

Fig. 3. Illustration of the bandit-based setting. Every week, the wokload from the past week is used to run simulations using every reordering policy. The "model" w is updated according to (9) and the policy for the next week is chosen according to (5.3). We denote by $P_{\text{bandit},t}$ the policy chosen by the bandit at week t.

2) If the number of allocated or requested cores is negative, we use the available positive value as the request. If both are negative (no data is available), we remove the job;
3) If the runtime or submission time is negative (no data is available), we remove the job.

6.1.2 Simulator

The choice of a simulator is a critical part of experimental validation that raises a trade-off between precision and runtime. High precision can be obtained by carefully modeling the platform and its network topology, or extracting information from the jobs from their sources or post-mortem logs. This is the approach used by high-fidelity batch scheduling simulators such as BatSim [13]. In the present work, the focus is on studying the EASY-Backfilling mechanism in itself, without addressing the allocation problem. Moreover, the experimental protocol used here requires many simulation runs. Therefore, we set the precision/runtime trade-off at the point which minimizes simulation runtimes. We discard all topological information relative to the platform and use the job processing times of the original workloads. In this setup, the processors are considered to be undistinguishable from each other, and jobs can be discontinuously mapped to any available processor on the system. During this work, we developed a lightweight simulator [28] and now release it to the community along with this article. See the reproducibility paragraph below for more information. With this simulator, we are able to replay EASY-FCFS on the CEA-CURIE trace in 33 seconds with a machine equipped with an Intel(R) Core(TM) i5-4310U CPU @ 2.00GHz. As a comparison, the Batsim simulator with network communication modeling disabled and no resource contiguity takes more than 7 hours to replay the same trace on the same hardware. The complete set of simulations presented below is executed on a single Dell PowerEdge R730 in 22 hours, including input preparation and statistical analysis code.

6.1.3 Resampling

Comparing algorithms in batch scheduling is made especially difficult by the fluctuations in commonly used metrics [18]. The variability in performance is known to outrank the available sample sizes: *It is not enough to simply replay an algorithm using a workload trace*. Indeed, as presented in Figure 1, the objective function we are using in this work is subject to high variability. In our experience, randomly shuffling months in a trace at random is enough to generate contradictory conclusions about which algorithm is better. This is a known problem [40] that motivated decades of research in workload modeling [16]. Workload models have drawbacks in that they usually lose the details of the workloads. This is problematic due to the non-linear aspect of the job scheduling process. This is discussed in [17], an approach that attempts to solve this problem by resampling data directly from the original traces. In this work, we use a simple implementation of these ideas. More precisely, we want to sample the job submission process, which is achieved in the following manner: (a) For every week in a resampled trace, we iterate over every existing user in the original trace; (b) For each of these users, we choose a week of the original trace uniformly at random; (c) The jobs from this user in this original week are chosen to be present in the week being resampled. This simple mechanism enables us to preserve the dependency within a week. Although the system state may not be independent from one week to an other, as illustrated in [24], it is reasonable to assume that the system is in a stationary state. Indeed, [24] performs train/test experiments using the same policy space as we use here. We do provide guidelines as how to incorporate tracking into our algorithms in Section 5 by introducing a decaying discount factor λ. In our experiments, we use stationary resampled data, and therefore no decay is necessary: λ here is set to 1.

6.1.4 Experimental Validation

For each of the 7 workloads considered, we use the original data to resample 60 traces, each with a length of two years. For each resampled trace, we run simulations using the EASY(P, Θ) scheduling heuristic. We fix the value of the job waiting time threshold Θ at 40 hours. The choice of this default stems from the analysis developed in [24]. Essentially, this value is high enough for queue reorderings to be leveraged while being low enough to provide the functional requirement of preventing starvation. We simulate EASY$(P, \Theta = 40h)$ using the following strategies:

- All fixed P_i for $1 \leq i \leq 12$ from the search set defined in Subsection 4.1. This is done for two reasons. First, it allows to see which policy works well for which platform. Second, it allows to compare how well strategies used in this work with the best *a-posteriori* known policy;
- For all the above strategies based on exact simulation (referred to as "Simulated feedback" in the remainder), on noisy simulation (referred to as "Noisy

TABLE 2
Hyper-parameter leave-one-out selection for ϵ.

Name	best ϵ
KTH-SP2	0.1
CTC-SP2	0.1
SDSC-SP2	0.1
SDSC-BLUE	0.1
ANL-Intrepid	0.3
CEA-Curie	0.3
Unilu-Gaia	0.1

Simulated Feedback") and on ϵ-greedy exploration (referred to as "Bandit Feedback"), we experiment with two policy periods: $\Delta \in \{1\text{day}, 1\text{week}\}$. This choice is a consequence of the natural rhythm of human activity: Daily or weekly simulation periods seem to ease comparing of performance between two periods. Indeed, arbitrary period values would needlessly increase the variability of the cost metric. For the bandit feedback strategy, λ is set to 1. This means that we do not use any decay in the estimation of the overall cost. Choosing to eliminate decay in this experimental validation is natural since our re-sampled data is in a stationary regime *by construction*, as explained in 6.1.3. The ϵ hyperparameter will be set at a value of 0.1 for all traces, which is the result of a leave-one out majority vote selection process on $[0.1, 0.3, 0.5, 0.7, 0.9]$. In this process, one trace among the 7 from Table 1 is taken out, and the most frequent best hyperparameter value is used. Table 2 outlines the result of this cross-validation. Assuming workload traces are i.i.d, this ensures that the reported results are fair with respect to the situation where the reader of the study would implement the strategy directly.

- Lastly, an additional "Random" strategy is used as a reference point, with, as before, $\Delta \in \{1\text{day}, 1\text{week}\}$. This strategy corresponds to choosing P_i uniformly at random in $\{P_1, \cdots, P_{12}\}$ during every period Δ_t.

The experimental validation uses of 7 traces $\times$ 60 resamples $\times$ (12+2+2+2) algorithms, which correspond to 12600 scheduling simulations spanning two years. This means a total simulated timespan of 25200 years. Moreover, both the full-feedback and noisy feedback strategies will also be calling simulations as part of their procedure. Indeed, these strategies re-simulate the system 12 times at every period Δ, which bumps the total effective simulation timespan to 92400 years. This motivates the use of a lightweight simulator in the context of this research.

6.2 Replicability

We publish a declarative workflow [29] based on our simulator [28] that runs experiments and generates all the figures from this paper.

There are various approaches for ensuring computational experiments are replicable, among which distributing complete operating system images, containers, or packaging software. As our experiments are CPU bound, we decide to opt for the software packaging approach [11]. This allows to replicate the experiments on clusters where virtualization is not available or kernels are too old for containers and decreases simulation runtime.

All the dependencies of our experiments (including our own code, dependencies for data processing and visualization and workflow engine) are automatically managed by Nix up to the actual execution of the code. The Nix package manager is available at https://nixos.org/nix.

The archive containing the experimental workflow can be obtained at:

```
https://github.com/freuk/obps
```

All the code associated with this work is released under the ISC [1] license. Running the experiments can be done on any platform equipped with the Nix package manager by running at the root of the extracted archive:

```
nix-build -A banditSelection
```

The experiments in this paper can run in less than 24 hours on a moderately parallel host. More precisely, we had access to a Dell PowerEdge R730 equipped with a total of 56 threads @2.4GHz each and 757 Gigabites of RAM. For this reason, the experimental workflow was designed to be executed on a single host.

All data from this paper were obtained from the Swf Parallel Workloads Archive [20].

All experiments are tied together using zymake [6], a minimalistic workflow system designed for computational experiments[4]. This system is analogous to a traditional build system with added workflow capabilities. The entire workflow that generates this article from the input data is contained in a single `zymakefile` to be found at the root of the main archive. This workflow is composed of the following steps:

- Data extraction from archives, data filtering, and data resampling. This is principally using shell scripts for data manipulation (see file `misc/strong_filter` for the filtering steps used) and ocaml code that implements the resampling method described in 6.1.3.
- Simulation. This step runs the lightweight ocaml backfilling simulator specially written for this work. This simulator is made available under the ISC [1] license as a persistent zenodo archive at [28].
- Analysis of the results. This step runs R code that generates the figures presented in this paper.

All resulting figures will be situated in the simlinked folder named `result` positionned at the root of the archive.

6.3 Results

6.3.1 Analysis of Basic Policies

We begin by drawing the attention of the reader to Table 3 that reports the average total waiting time performance improvement over EASY(FCFS,40h) of all the fixed policies defined in 4.1. Let us denote the resampled traces of a given workload by T_k, where k is the index of the resampled trace ($1 \leq k \leq 60$). Let the number of jobs in the resampled trace

4. The zymake system is also packaged by our nix expressions.

TABLE 3
Average total waiting time diminution of fixed policies with respect to EASY(FCFS). The precise definition of the value reported is provided in Eq (12)

Trace	LCFS	SPF	LPF	SQF	LQF	LEXP	SEXP	SRF	LRF	SAF	LAF
KTH-SP2	-13%	**-16%**	+5%	**-16%**	+3%	-8%	-15%	-8%	-13%	-12%	+15%
CTC-SP2	-40%	-19%	-14%	**-47%**	+15%	-36%	-19%	-38%	-12%	-44%	+22%
SDSC-SP2	-8%	**-15%**	+4%	-8%	+4%	-3%	**-15%**	-3%	-10%	-11%	+18%
SDSC-BLUE	-26%	-29%	0%	-29%	+17%	-17%	-26%	-15%	-15%	**-32%**	+23%
ANL-Intrepid	-25%	-9%	-3%	**-28%**	-2%	-21%	-24%	-21%	-10%	-19%	+5%
CEA-Curie	-46%	-23%	-45%	**-57%**	-19%	-51%	-23%	**-57%**	-24%	-40%	-17%
Unilu-Gaia	-58%	-33%	-10%	**-62%**	+16%	-49%	-27%	-56%	-16%	-62%	+20%

T_k be $l(T_k)$. Let $\bar{w}(T_k, P)$ be the cumulative waiting time obtained by a simulation run for a strategy P:

$$\bar{w}(T_k, P) = \sum_{j=1}^{l(T_k)} \text{Wait}_j^P \tag{11}$$

The values appearing in Table 3 are exactly:

$$\frac{\sum_{k=1}^{k=60} \bar{w}(T_k, \text{EASY}(P_i, 40h)) - \bar{w}(T_k, \text{EASY}(\text{FCFS}, 40h))}{\sum_{k=1}^{k=60} \bar{w}(T_k, \text{EASY}(\text{FCFS}, 40h))} \tag{12}$$

The best values are outlined in bold in the table. These values give the best attainable performance in the search set defined in 4.1. Note that there seems to be some regularity; for instance the SAF policy is remarkably inefficient and the LQF policy works very well. However there are traces for which LQF is not optimal. For these machines, ordering jobs by decreasing size seems to be more efficient than ordering them by time. This study is limited in scope, and these results are possibly different on other machines. Moreover, the simulator model is simplistic and the relative order of policies may change, for instance when introducing a topology model in the simulations. This is however sufficient to study adaptive policies.

These are *a-posteriori* results, which means that we only know *post-factum* that this policy would have been better.

In the present work, we are studying adaptive policies that obtain a good performance on any given system by selecting one of these policies. Figure 4 reports the evolution of the average cumulative waiting time for the UniLu-Gaia trace. This figure shows how the average improvement in cumulative waiting time evolves as a function of time. In other words, this is the average curve obtained if one were to plot the cumulative sum of job waiting times for every resampled trace, incrementing the sum when a job finishes. Accordingly, the final values attained by every curve correspond exactly to the values reported in Table 3. The dashed values represent the 10th and 90th percentile of the values.

Observe that while the 10th and 90th percentiles of the improvement with respect to EASY-FCFS are large, the values are still contained enough to make definite statements about how these policies compare to the FCFS baseline. This figure does not report the variability of the policies against one another, and does not allow to compare them beyond their average performance. The fact that the size of the percentile band is roughly half the best attainable

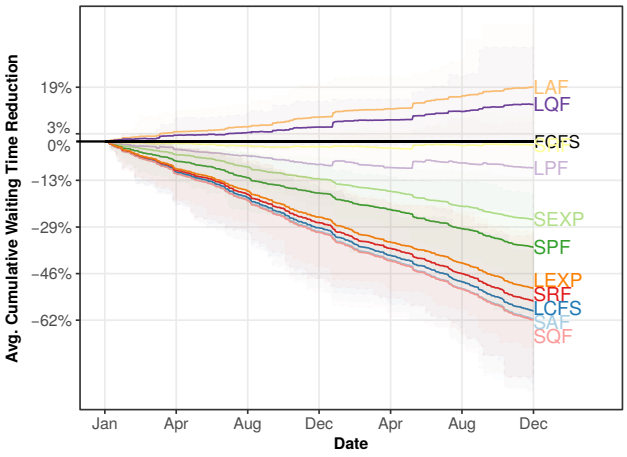

Fig. 4. Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the policies P_i for $i = 1 \ldots 12$ on the UniLu-Gaia trace. The average is obtained by resampling the original trace 60 times. The dashed lines represent the 10th and 90th percentiles of the values across this resampling.

improvement was our original motivation for this work, as one can expect to be able to distinguish between these policies in an online manner.

6.3.2 Analysis of Policy Selection Strategies

Figures 5 and 6 report the main experimental results concerning the strategies used in this paper. These figures show the average cumulative improvement over EASY(FCFS,40h) of the three strategies we use in the same format as Figure 4. There are five strategies displayed on the graph:

- Random $P \in P_i, 1 \leq k \leq 60$. ($\Delta$ =1 week)
- Bandit Feedback (Δ =1 week)
- Bandit Feedback (Δ =1 day)
- Simulated Feedback (Δ =1 week)
- Noisy Simulated Feedback (Δ =1 week)

Table 4 reports the average total waiting time performance improvement over EASY(FCFS,40h) of all the strategies used along with the random variant. We make the following observations.

FCFS vs Random. Randomly taking a policy in the search set defined in 4.1 is better than using the FCFS order. This is due to the known fact that the FCFS order is not optimal for optimizing the average waiting time objective, and confirms that the search set is well chosen. The value

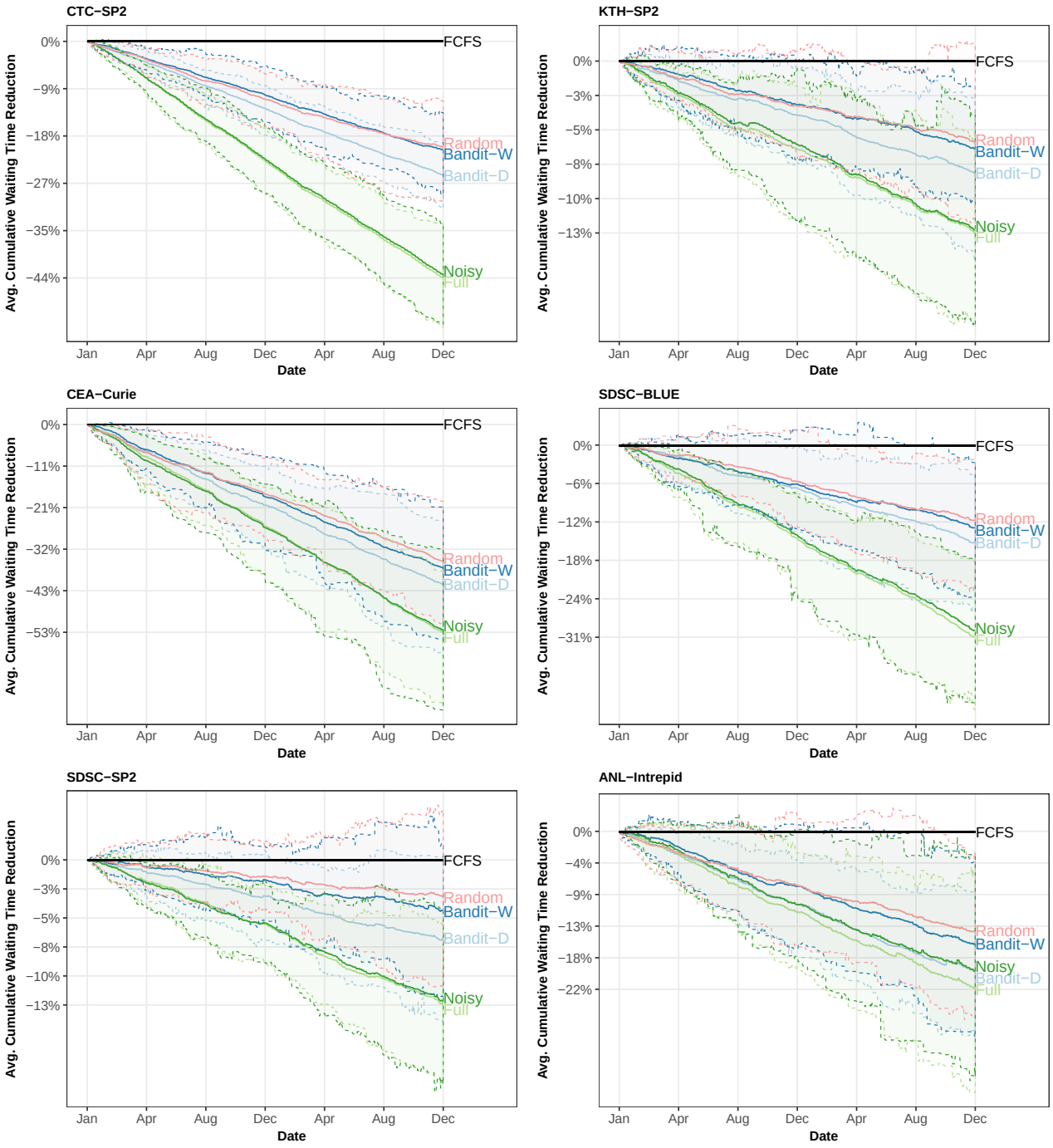

Fig. 5. Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the Simulated Feedback, Noisy Simulated Feedback and Bandit Feedback policies. The average is obtained by resampling the original trace 60 times. The dashed lines represent the 10th and 90th percentiles of the values across this resampling. Each figure is a different trace. This figure is followed-up in figure 6 for the remaining UniLu-Gaia trace.

TABLE 4
Average Cumulative waiting time improvement of the policies used with respect to EASY(FCFS).

Trace	Best(hindsight)	Random		Full		Noisy		Bandit	
Δ		week	day	week	day	week	day	week	day
KTH-SP2	-16%	-6%	-8%	**-12**%	11%	**--12**%	**--12**%	-7%	-10%
CTC-SP2	-47%	-20%	-23%	**-44**%	**-44**%	-44%	-44%	-26%	-31%
SDSC-SP2	-15%	-4%	-7%	**-13**%	**-13**%	**--13**%	**--13**%	-5%	-8%
SDSC-BLUE	-32%	-12%	-14%	**-31**%	**-31**%	-30%	-29%	-12%	-17%
ANL-Intrepid	-28%	-13%	-20%	-22%	**-26**%	-19%	-24%	-13%	-20%
CEA-Curie	-57%	-35%	-42%	**-53**%	-47%	**--53**%	-48%	-36%	-46%
Unilu-Gaia	-62%	-29%	-31%	-59%	**-60**%	-58%	-58%	-33%	-34%

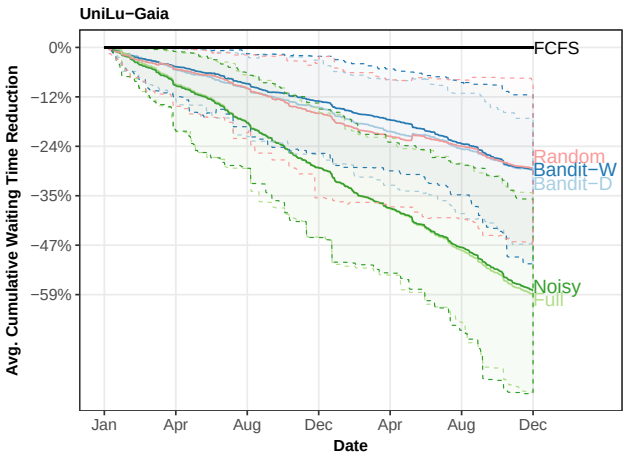

Fig. 6. Figure 5 continued, plot for the UniLu-Gaia log.

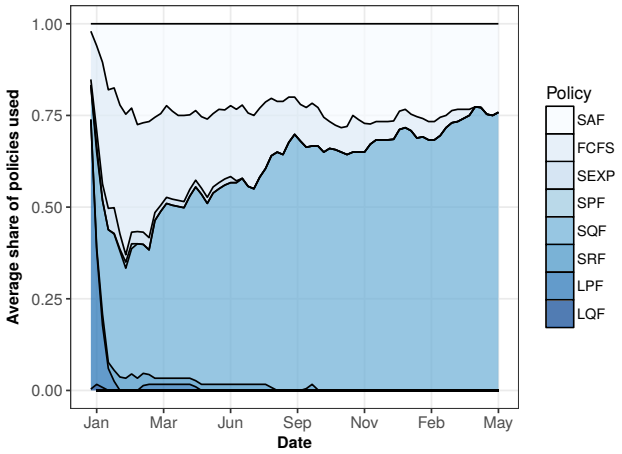

Fig. 7. Share of the policies chosen by the Noisy policy (Δ =1 day, λ = 1) as a function of time for the UniLu-Gaia trace. The average choice is obtained by resampling the original trace 60 times and aggregating by date. The legend excludes policies that are never used by the strategy.

of the period length Δ has an impact on the performance of the random policy. Indeed, shorter periods lead to better performance.

The Simulated Feedback strategy consistently performs almost as well as the best fixed policy. The values attained by the exact simulation based strategy almost match the highlighted values from Table 3. This corresponds to the curve labeled "Full" in Figures 5 and 6. Moreover, these average cumulative waiting time curves exhibit linearity (as opposed to being concave), which further shows that the choice of the best policy can be made rapidly. Note that the value of Δ has no impact over the performance of this strategy.

Noisy simulations can be used. The performance attained by the noisy variant almost matches that of the exact-simulation based strategy. This indicates that it is not necessary to have access to a precise simulator. Rather, we may hope that an imprecise but unbiased reporting of the performance is enough to rapidly select a reordering policy.

Full vs Bandit feedback. As expected, the bandit strategy evolves between the random strategy and the full feedback strategy. The bandit-based strategy always fares better with a shorter period Δ, recovering between a quarter to the whole gap between the Random and Full Feedback strategies.

Stability of the policy selection. Figure 7 gives insight regarding the behavior of the noisy feedback policy with $\Delta = 1$ week. This figure reports the share of the policies chosen by the strategy on average as a function of time. This is an area plot: The height of the colored region represents the value. Moreover, the proportion of times each policy is chosen by the strategy is aggregated across resampled traces.

While this policy obtains an excellent performance, the convergence to a fixed policy is slow. This is not harmful *per se* and just means that it is difficult to distinguish between the best policies. Additionally, we observe that the two increasingly dominant policies, LAF and LQF, are the best overall policies. This means that the estimation converges to the correct point.

SQF versus online tuning. Table 3 shows that the SQF policy works quite well on the traces considered here on average, and under this resampling scheme. However, the variability is high, and this policy is not the best on all traces. It interesting to have an experimental evaluation of exactly how much one gains on average by using an online policy versus a leave-one-trace-out majority vote policy choice. However, a good estimation of this quantity is tricky. Indeed, we expect the change in the relative performance of policies to increase with two factors. First, simply adding workload traces to our selection will naturally show more cases with a different behavior. Second, factoring in the topology model of the machines should increase the heterogeneity between platforms and therefore increase the depen-

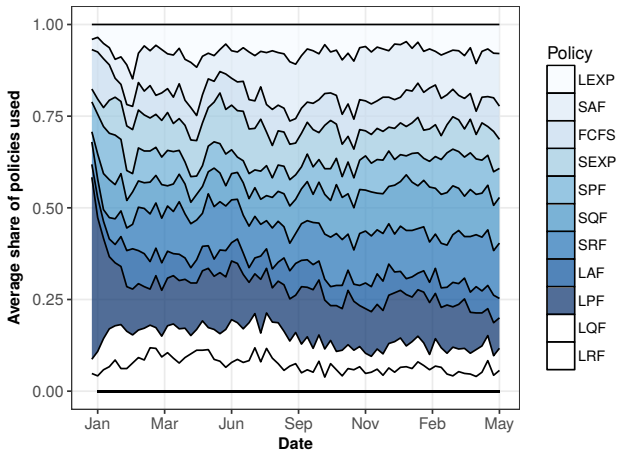

Fig. 8. Share of the policies chosen by the Bandit Feedback(Δ =1 day, $\epsilon = 0.5, \lambda = 1$) policy as a function of time for the UniLu-Gaia trace. The average choice is obtained by resampling the original trace 60 times and aggregating by date. The legend excludes policies that are never used by the strategy.

dence of the simulation performance on the trace/platform. Such a study would be much more ambitious and is beyond the scope of this study. Here the need for adaptivity is actually understated by the simulation approach.

While we can not conclude on how much one gains by not using SQF in this case, we can comment on the added complexity and computing overhead of using the simulation-based and bandit online strategies. In both cases, there is no overhead in scheduling decisions as the policy is fixed during the period (one day, or week). The bandit strategy has a negligible computational overhead at the end of the period. Its two computational operations are maintaining a sequential average and generating a pseudo-random number. The simulation-based strategy requires to simulate the system K times every period end, where K is the number of alternative policies considered (here, K=12). Using our simulator, this takes a few seconds. Using a more precise simulator with topological modeling, one can argue that the overhead is still manageable since these simulations can be easily done in parallel. The most important constraint is that these simulations should not take too much time compared to the period size, in order not to delay too much the policy switch decision.

Figure 8 is the analogue of Figure 7, this time for the Bandit Feedback policy with Δ = 1 week. Observe that here the choice of policy is much less stable. The convergence is extremely slow, which is expected given the variability of the reward we give to the bandit algorithm.

7 Conclusion

The scheduling of parallel jobs on a given HPC platform is a very hard optimization problem with many uncertain parameters. Even though these uncertainties could be reduced, determining efficient strategies remains difficult.

In this work, we presented a new way of addressing this problem. The idea was to look directly at the scheduling process instead of trying to change its parameters.

More precisely, we showed that it is worth learning on the scheduling process itself. Indeed, reordering the submission queues under EASY-Backfilling leads to considerable gains in performance.

This approach was used in two methods: a method based on a simulator and a method based on a multi-armed bandit algorithm. The first one provides very good results, reducing the average waiting times of the baseline FCFS ordering policy by 11% to 60%. The second one is slightly less efficient with an improvement factor of 8% to 46%. However, the bandit-based method is easier to use and cheaper to run.

There are three directions to extend this work. First, the question of the existence of a more efficient selection strategy for bandit feedback for these systems is not addressed by this work. Is a simulator really necessary? Second, one can't help but wonder what performance could be achieved by extending the search space to a wider class of policies. This means studying learning performance using some function approximator.

As both these directions are data-bound, they will require careful application of trace resampling techniques to be successful.

Acknowledgements

Authors are sorted in alphabetical order. The authors would like to warmly thank Pierre Neyron for help with computational experiments, as well as Michael Mercier, Salah Zrigui, Bruno Raffin and Theophile Terraz for their comments and Arnaud Legrand for his work on reproducibility. We gracefully 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), Joseph Emeras (CEA Curie and UniLu Gaia), Susan Coghlan, Narayan Desay, Wei Tang (ANL Intrepid), and of course Dror Feitelson. The Metacentrum workload log was graciously provided by the Czech National Grid Infrastructure MetaCentrum. This work has been partially supported by the LabEx PERSYVAL-Lab(ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir. Experiments presented in this paper were carried out using the Grid'5000 testbed. Grid'5000 is supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations[5]. Access to the experimental machine(s) used in this paper was gracefully granted by research teams from LIG[6] and Inria[7].

References

[1] ISC license. https://opensource.org/licenses/ISC, accessed 03/23/17, 2017.

[2] K. Aida. Effect of job size characteristics on job scheduling performance. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPDPS '00/JSSPP '00, pages 1–17, London, UK, UK, 2000. Springer-Verlag.

5. https://www.grid5000.fr
6. http://www.liglab.fr
7. http://www.inria.fr

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, 2002.

[4] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. The non-stochastic multi-armed bandit problem, 2002.

[5] A. A. Banos, A. Baros, S. Fernando, and V. S. College. On pseudo-games. *Annals of Mathematical Statistics*, pages 1932–1945, 1968.

[6] E. Breck. zymake: a computational workflow system for machine learning and natural language processing. In *Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 5–13. Association for Computational Linguistics, 2008.

[7] S. Bubeck and N. Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[8] L. Buşoniu, D. Ernst, B. De Schutter, and R. Babuška. Approximate reinforcement learning: An overview. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*, pages 1–8. IEEE, 2011.

[9] S.-H. Chiang, A. Arpaci-Dusseau, and M. K. Vernon. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, number 2537 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, July 2002.

[10] K. Deng, J. Song, K. Ren, and A. Iosup. Exploring portfolio scheduling for long-term execution of scientific workloads in iaas clouds. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 55. ACM, 2013.

[11] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Universiteit Utrecht, Jan. 2006.

[12] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard. Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator. In *20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, United States, May 2016.

[13] P.-F. Dutot, M. Mercier, M. Poquet, and O. Richard. Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator. In *20th Workshop on Job Scheduling Strategies for Parallel Processing*, Chicago, United States, May 2016.

[14] D. Feitelson. Parallel Workloads Archive: Cleaning Logs. http://www.cs.huji.ac.il/labs/parallel/workload/logs.html#clean, accessed 03/23/17, 2017.

[15] D. G. Feitelson. Metrics for parallel job scheduling and their convergence. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 188–205. Springer, 2001.

[16] D. G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, New York, NY, USA, 1st edition, 2015.

[17] D. G. Feitelson. *Resampling with Feedback — A New Paradigm of Using Workload Data for Performance Evaluation*, pages 3–21. Springer International Publishing, Cham, 2016.

[18] D. G. Feitelson and L. Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–24. Springer, 1998.

[19] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling — a status report. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'04, pages 1–16, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] D. G. Feitelson, D. Tsafrir, and D. Krakov. Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967 – 2982, 2014.

[21] E. Frachtenberg and U. Schwiegelshohn, editors. *Job Scheduling Strategies for Parallel Processing: 14th International Workshop, JSSPP 2009, Rome, Italy, May 29, 2009. Revised Papers*. Springer-Verlag, Berlin, Heidelberg, 2009.

[22] 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*, SC '15, pages 641–6410, New York, NY, USA, 2015. ACM.

[23] D. Jackson, Q. Snell, and M. Clement. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing*. Springer, 2001.

[24] J. Lelong, V. Reis, and D. Trystram. Tuning EASY-backfilling queues. In *21st Workshop on Job Scheduling Strategies for Parallel Processing*, 31st IEEE International Parallel & Distributed Processing Symposium, Orlando, United States, May 2017.

[25] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6), June 2001.

[26] A. Nissimov and D. G. Feitelson. *Probabilistic Backfilling*, pages 102–115. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[27] D. Perkovic and P. J. Keleher. Randomization, speculation, and adaptation in batch schedulers. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 7–7, Nov 2000.

[28] V. Reis. The ocaml easy-backfilling simulator. http://github.com/freuk/ocst, 2017.

[29] V. Reis. Replication workflow. http://github.com/freuk/obps, 2017.

[30] SchedMD. SLURM online documentation. http://slurm.schedmd.com/sched_config.html, 2017.

[31] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. The EASY - loadleveler API project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, pages 41–47, London, UK, 1996. Springer-Verlag.

[32] 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*, pages 514–519. IEEE, 2002.

[33] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective reservation strategies for backfill job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 55–71. Springer, 2002.

[34] A. Streit. The self-tuning dynP job-scheduler. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, Apr. 2002.

[35] D. Talby and D. G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 10–pp. IEEE, 2005.

[36] W. R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[37] M. Tokic. Adaptive ϵ-greedy exploration in reinforcement learning based on value differences. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, pages 203–210, 2010.

[38] Top500.org. TOP500 online ranking. https://www.top500.org, 2017.

[39] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, June 2007.

[40] D. Tsafrir and D. G. Feitelson. Instability in parallel job scheduling simulation: The role of workload flurries. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–, Apr. 2006.

[41] Y. Ukidave, X. Li, and D. Kaeli. Mystic: Predictive scheduling for GPU based cloud servers using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 353–362, May 2016.

[42] A. Vishnu, H. v. Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie. Fault modeling of extreme scale applications using machine learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 222–231, May 2016.

[43] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, 1989.