Online (Bandit) Policy Selection for EASY-Backfilling

Eric Gaussier
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG
France
eric.gaussier@imag.fr

Jérôme Lelong
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK
France
jerome.lelong@imag.fr

Valentin Reis
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
France
valentin.reis@imag.fr

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

ABSTRACT

The EASY-FCFS heuristic is the basic building block of job scheduling policies in most parallel High Performance Computing (HPC) platforms. Despite its good properties (simplicity, no starvation), it could still be improved on a per-system basis. This tuning process is difficult because of non-linearities in the scheduling process. The study proposed here 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 of the job queue under several feedback modes. We show via a comprehensive experimental campaign that noisy feedback (using a weak simulator) recovers existing in-hindsight results that allow to divide the average waiting time up to a factor of 2. Moreover, we show that bandit feedback can be used by a simple multi-armed bandit algorithm to decrease the average waiting time down to 40% of its original value without using a simulator.

ACM Reference format:
Eric Gaussier, Jérôme Lelong, Valentin Reis, and Denis Trystram. 2016. Online (Bandit) Policy Selection for EASY-Backfilling. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Context

Providing the computing 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 of the answer to this challenge is to build extreme-scale parallel and distributed platforms. The never-ending race for more computing power and storage capacity does not only lead to sophisticated specific Exascale platforms, but 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 softwares allow to run tens of thousands of jobs on hundreds of thousands of cores. They are based on robust policies, however, despite their positive properties such as relative simplicity and prevention of job starvation, there is still room for improvement. We propose here to study how to tune the ordering of the submission queues in the frame of the classical EASY-BackFilling family of heuristics using two new online learning approaches (an approach based on a simulator and an approach using multi-armed bandit).

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 of the number of works at the interface of HPC and BigData 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, memory requirements, etc.) via prediction. However, only few studies evaluate the impact of such techniques on the performances of the resource manager.

Contrarily to most existing studies, which consider the learning of specific parameters of a trace, we provide a complete system that uses feedback from the system for adapting the whole behavior online. Our focus within this work is to learn how the management system behaves within the usual framework of EASY-BF. 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 campaign that uses 7 actual traces.

1.2 Contributions

Our main contribution is to investigate two original methods for learning a good scheduling policy on a per-system basis. We focus in this work on the average waiting time of the jobs. Each method aims to choose 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 that derives feedback by measuring system performance.

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

Both of these methods are evaluated with a specially developped simulator that is open-sourced and packaged for the community.

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, this means that we are able to recover in an online fashion existing results that divide average waiting times by factor of X to X. Indeed, we show that the required sample sizes for policy selection in batch schedulers are small enough for workload resampling to be avoided in practice. The variant based on noisy data, which is closer to actual conditions of uncertainties on both jobs and platform, also behaves very well. The alternate approach based on a multi-armed bandit also provides reasonable results at a much lower cost, dividing the average waiting times by a factor of X to X.

Let us emphasize the main results of this study: While the First-Come, First-Serve approach to minimizing the waiting times is a common default strategy, it can absolutely be outperformed. We show how to automate selection of a good alternative policy in two cases (depending 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 discuss the most significant results related to job scheduling and learning algorithms in this context. Let us start by recalling some basics about scheduling heuristics in HPC platforms:

Parallel job scheduling is a old studied theoretical problem [15, 17] whose practical ramifications, varying hypotheses, and inherent uncertainty of the problem applied on 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 [26] and Conservative [20] backfillings.

While Conservative Backfilling offers many advantages [27], it has a significant computational overhead, which mainly explains why most of the machines of the top500 ranking [3] rather use a variant of EASY-BF 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 softwares (most notably SLURM [2]), this is rarely done without fine tunings 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 [31], sometimes even in a randomized manner [22], as well as some implementations [19]. However, as successful as they may be, these works do not address the dependency of scheduling metrics on the workload [5]. Indeed these studies most often report *post-hoc* performance since they compare algorithms after the workload is known.

The dynP scheduler [28] proposes a systematic method to tuning these queues, although it requires simulated scheduling runs at decision time and therefore, it costs much more than the natural execution of EASY.

2.1 Data-aware resource management.

There was a recent focus on leveraging the huge amount of data available in large scale computing platforms in order to improve system performance. Some works use collaborative filtering to colocate tasks in clouds by estimating application interference [32]. Some others are closer to the application level. For instance, [33] uses binary classification to distinguish benign memory faults from application errors in order to execute recovery algorithms.

Several works use similar techniques in the context of HPC, in particular [18, 31], hoping that better job runtime estimations should improve the scheduling [11]. Some algorithms estimate runtime distributions model and choose jobs using probabilistic integration procedures [21]. However, these works do not address the duality between the cumulative and maximal scheduling costs, as mentioned in [18]. While these previous works intend to estimate uncertain parameters, we consider here a more pragmatic approach, which directly learn a good scheduling policy from a given policy space.

Existing work [6] takes this approach in an offline manner, by splitting workload data in a training/testing fashion. This work studied EASY-Backfilling and shows that the relative performance of some well-known priority orders for starting jobs differs between workloads, but is relatively stable troughout 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. At every round, an action (represented by an arm of the Bandit) must be chosen in 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. There exist a bunch of works that address this problem under a variety of constraints, The two most popular settings are the original stochastic case and the adversarial case. See [29] for the original work on the stochastic case and [7] for the "Upper Confidence Bound" family of algorithms. According to our knowledge, the work of [9] is the earliest known work to us for the adversarial case see [8] for the "Exponential-weight" family of algorithms. We refer to the review [10] for a comprehensive overview 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. A simple heuristic called *Epsilon-Greedy* introduced in [7] is known to achieve good results in practice, and does not have this requirement.

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 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 a HPC platform represented by a pool of m identical resources. This is an on-line problem since the jobs are submitted over time and their characteristics are only known when they are released. Below is the brief description (with the corresponding notations) 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 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 that establishes the order in which 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 [26]. This strategy has no worst case guarantee beyond the absence of starvation (i.e. every job will be scheduled at some moment).

The EASY-FCFS heuristic uses a job queue to select and backfill jobs. At any time that requires a scheduling decision (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. It then makes a reservation for this job at the earliest predictable time and starts *backfilling* the job queue in FCFS order, starting 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.

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

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 (**AvgWait**). It is worth noting that the **MaxWait**, a.k.a the maximal value of the waiting time of all the jobs is also worthy of interest. Unfortunately, these criteria are antagonistic in nature. 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 constitute a robust framework for tuning EASY.

4.1 Reordering the job Queue

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 settle on a reasonable search space of 10 reordering policies.

(1) FCFS: First-Come First-Serve, which is the widely used default policy [26].
(2) LCFS: Last-Come First-Serve.
(3) SPF: Smallest estimated Processing time $\widetilde{p_j}$ First [27].
(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 [27], where the expansion factor is defined as follows:

$$\frac{start_j - r_j + \widetilde{p_j}}{\widetilde{p_j}} \tag{2}$$

where $start_j$ is the starting time of job j.
(8) SEXP: Smallest Expansion Factor First
(9) LRF: Largest Ratio $\frac{p_j}{q_j}$ First
(10) SRF: Smallest Ratio First
(11) LAF: Largest Area $p_j \times q_j$ First
(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. In the following, we denote these policies by P_i with $i = 1 \dots 10$.

4.2 Thresholding

As existing works point out, reordering the job queue means losing the no-starvation guarantee and some individual jobs therefore can wait an undue amount of time. It is possible to introduce a thresholding mechanism in order to prevent this behavior: When a job's *waiting time so far* exceeds a fixed threshold Θ, it is jumped 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

We present here the strategies we have retained for selecting a policy. 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 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.

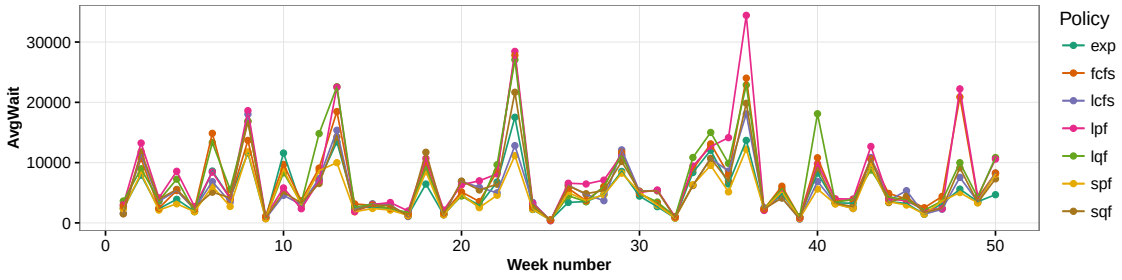

Figure 1: Variability in the weekly average waiting time in the KTH-SP2 trace (see Subsection 6.1.1) for 7 different policies. The policy set is reduced as not to obstruct the figure.

Algorithm 1: EASY(P, Θ) policy

Input: Queue Q of waiting jobs.
Output: None (calls to $Start()$)

Sort Q according to P_R
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
for job j in Q do **do**
  if j can be started given the current system use. **then**
    Pop j from Q
    $Start(j)$
  else
    Reserve j at the earliest time possible according to the estimated running times of the currently running jobs.
    Backfilling jobs
    for job j' in $Q \setminus \{j\}$ **do**
      if j' can be started without delaying the reservation on j. **then**
        Pop j' from Q
        $Start(j')$
      end if
    end for
    break
  end if
end for

We further assume that there is a certain regularity among periods, *i.e.* that the distributions underlying the jobs submitted do not radically differ between consecutive periods. This assumption is validated in the study presented in [6]. It further entails that the behavior of a policy on the previous periods reflects its behavior on the current one, so that the selection of a policy can be based on its behavior on previous periods. However, there may be some variability between different periods for certain cost metrics [13]. This is illustrated in Figure 6 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 the 7 policies considered. This indicates that when when the cost metric is averaged over different periods, there is a tradeoff 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 of course reminiscent of reinforcement learning. 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 **??** can bypass dimensionality issues via function approximation. However, these methods rely on large amounts of data, and we are studying online (often called *on-policy* methods in reinforcement learning) methods. 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 ($0 \leq t < T$), and P_i ($1 \leq i \leq 10$) one reordering policy (defined in Section 4.1). The cost of policy P_i during the period Δ_t is defined as:

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

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:

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

$\lambda \in [0, 1]$ is a decay parameter that can be used to privilege recent history (*i.e.* recent periods). One then selects the policy P that minimizes the above cost for the period Δ_T:

$$P = \text{argmin}_{P_i, 1 \leq i \leq 10} w(\rightarrow T; P_i) \tag{5}$$

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 submission period.

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

In order to simulate how the simulation strategy for selecting policies would work on non simulated traces, we randomly introduce noise in the estimate of the waiting time defined by Eq. 3 by rescaling it, either down or up, for each job:

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

where ρ is uniformly sampled in the interval $[0.85, 1.15]$, thus adding a +/-15% noise on the waiting time estimated by the simulator. The overall cost is then defined in the same way as before, leading to:

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

As before (see Eq. 5), the policy that minimizes the above cost is selected for the period Δ_T.

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.

We thus 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 [34] and bandit problems [7], to tradeoff 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 Δ_t. We define the cost of policy P_i during the period Δ_t as:

$$w_\epsilon(t;P_i) = \begin{cases} \sum_{j=1}^{l(\Delta_t)} \mathbf{Wait}_j^{P_i} & \text{if } P_i \text{ used during } \Delta_t \\ 0 & \text{otherwise} \end{cases}$$

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 Δ_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 Δ_t, as well as disregard jobs that were submitted during Δ_t but finish later.

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

$$w(\rightarrow T;P_i) = \frac{1}{\sum_{t=0}^{T-1} \mathbb{1}(P_i,t) l(\Delta_t)} \sum_{t=0}^{T-1} \mathbb{1}(P_i,t) \lambda^{T-1-t} w_\epsilon(t;P_i) \tag{8}$$

where $\mathbb{1}(P_i,t) = 1$ if P_i is the policy used during the period Δ_t and 0 otherwise. The normalization by $\sum_{t=0}^{T-1} \mathbb{1}(P_i,t) l(\Delta_t)$ is necessary here to ensure that policies remain comparable over time, as 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 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):
$$P = \text{argmin}_{P_i, 1 \le i \le 10} w(\rightarrow T;P_i) \tag{9}$$
- With probability ϵ, select a policy P_i, $1 \le i \le 10$, at random (exploration mode).

Several studies propose to decay ϵ over time, the exploration being less important once the estimates for the different policies are reliable [30]. 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.

6 EXPERIMENTS

This section compares the different approaches to policy selection via a comprehensive experimental campaign. Subsection 6.1 outlines the experimental protocol used and Subsection 6.2 contains the experimental results and their discussion.

6.1 Experimental Protocol

The Evaluation of computer systems performance is an intricate task[14]. 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 and provides a replicable workflow for the interested reader.

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	163840	68k	9 Months
CEA-Curie	2012	80,640	312k	3 Months
Unilu-Gaia	2014	2,004	51k	4 Months

6.1.1 Traces. The experimental campaign used in this work uses a mixture of small and large systems from different periods. Table 1 presents the 7 logs, which can all be obtained from the Parallel Workload Archive [16]. We use the 'cleaned'[1] version of the logs as per the archive. We impose an additional filtering step to the workloads[2] in order to clean singular data. More precisely, we apply the following modifications:

(1) If the number of allocated or requested cores of a job exceeds the size of the machine, we remove the job.
(2) If the number of allocated or requested cores is negative, we use the available positive value as the request. If both are negative, we remove the job.
(3) If the runtime or submission time is negative, we remove the job.

6.1.2 Simulator. The choice of a simulator is a critical part of experimental validation that raises a tradeoff 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-morted logs. This is the approach used by high-fidelity batch scheduling simulators such as Batsim [12]. In the present work, the focus is on studying the EASY-Backfilling mechanism in itself, without adressing the allocation problem. Moreover, the experimental protocol used here requires many simulation runs. Therefore, we set the precision/runtime tradeoff 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. We develop a lightweight simulator[24] and an accompanying multi-armed bandit library[23]. 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 point of comparison, the Batsim simulator with no network communication modeling and no resource contiguity takes more than 7 hours to replay the same trace on the same hardware. The complete simulation campaign presented below is executed on a single Dell PowerEdge R730 in 22 hours, including input preparation and analysis code.

6.1.3 Resampling. To carry our experiments, we need to generate many trace samples. More precisely, we want to sample the job submission process, which is achieved by splitting the traces in weeks, randomly shuffling them and picking the jobs submitted in the selected weeks. 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, we can reasonably assume that it evolves under its stationary distribution. Thus, computing an average over the sampled traces at a given time can be related to a time average of an underlying Markov process, which converges to the true quantity thanks to the ergodic theorem. Moreover, the job submission process has no long range correlation making it look like an independent and identically distributed process when considering sufficiently time spaced weeks.

6.1.4 Experimental campaign.

6.1.5 Replicability. Readers interested in replicating the experiments (or a part thereof) are invited to peruse to the 'artifacts description' appendix of this paper. We provide to persistent archives containing the simulator, the analysis code and an automated workflow that replicates all experiments and figures present in this paper.

6.2 Results

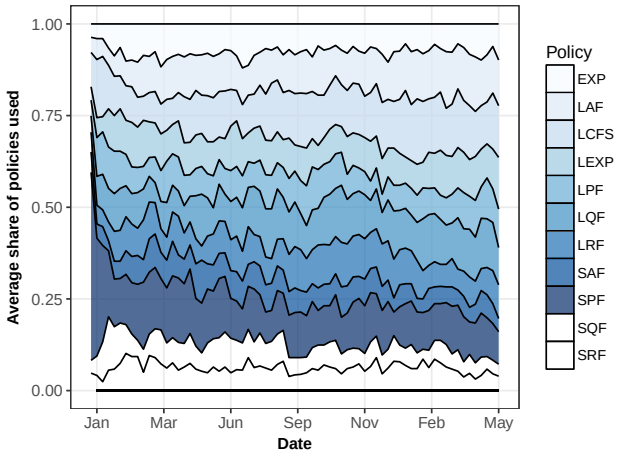

Figure 5: Share of the policies chosen by Epsilon-Greedy as a function of time. The average choice is obtained by resampling the original trace 100 times and aggregating by date.

[1] http://www.cs.huji.ac.il/labs/parallel/workload/logs.html#clean
[2] This filtering step is available in the reproducible workflow[25] as shell script misc/strong_filter

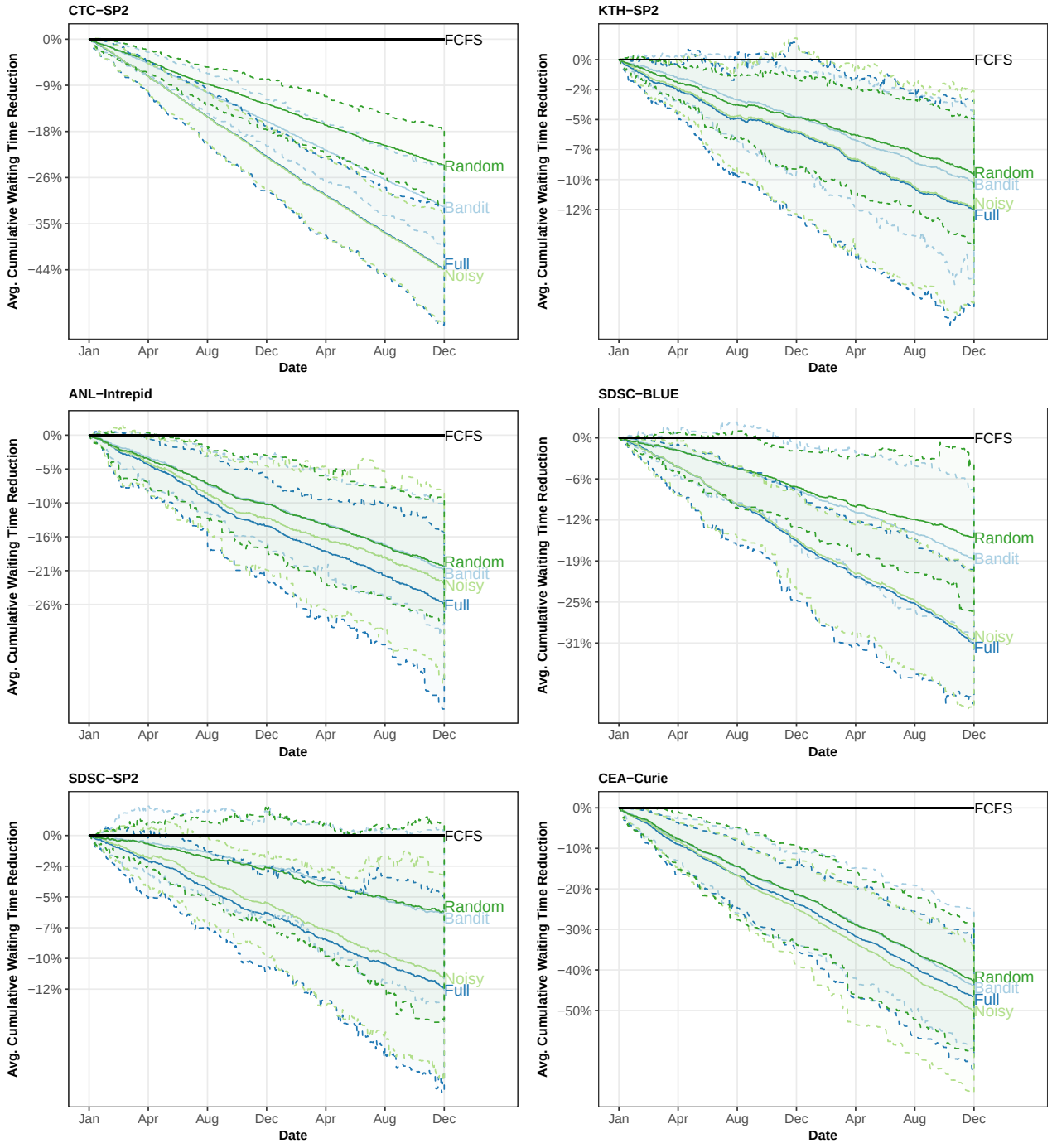

Figure 2: Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the FullFeedback, NoisyFeedback and EpsilonGreedy policies. The average is obtained by resampling the original trace 100 times. The dashed lines represent the 10th and 90th percentiles of the values across this resampling. Each figure is a different trace, and this figure is followed-up in figure 3 for the UniLu-Gaia log.

7 CONCLUSION

ACKNOWLEDGMENTS

Authors are sorted in alphabetical order. The authors would like to warmly thank Pierre Neyron for help with computational experiments and Arnaud Legrand for discussions about reproducibility. We gracefully thank the contributors of the Parallel Workloads Archive, Victor Hazlewood (SDSC SP2), Travis Earheart and

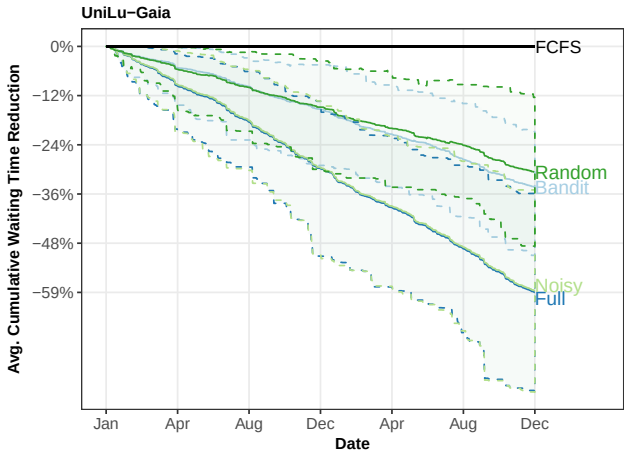

Figure 3: Follow-up from figure 2, plot for the UniLu-Gaia log.

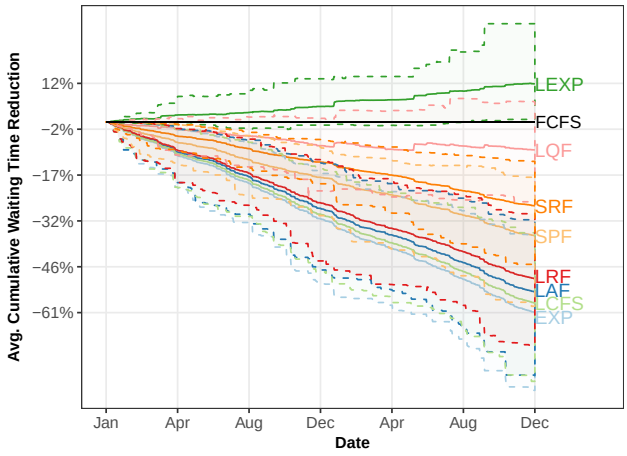

Figure 4: Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the policies P_i for $i = 1 \dots 10$ on the UniLu-Gaia trace. The average is obtained by resampling the original trace 100 times. The dashed lines represent the 10th and 90th percentiles of the values across this resampling.

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.

REFERENCES

[1] ISC license. https://opensource.org/licenses/ISC, accessed 03/23/17. (????).
[2] SLURM Online documentation. http://slurm.schedmd.com/sched_config.html. (????).

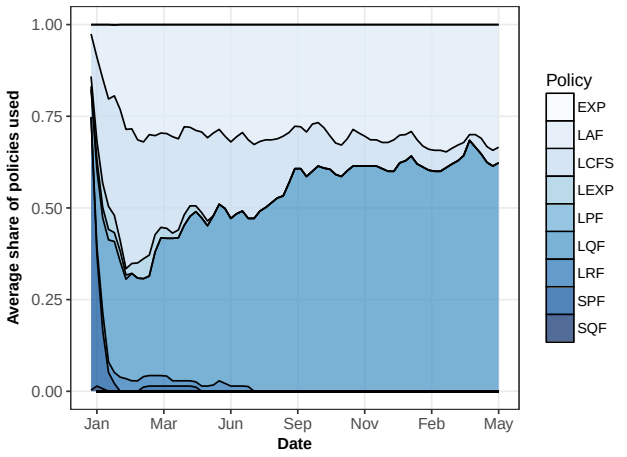

Figure 6: Share of the policies chosen by the Noisy policy. The average choice is obtained by resampling the original trace 100 times and aggregating by date.

[3] TOP500 Online Ranking. https://www.top500.org/. (????).
[4] Zenodo. http://zenodo.org/, accessed 03/23/17. (????).
[5] Kento Aida. 2000. 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)*. Springer-Verlag, London, UK, UK, 1–17. http://dl.acm.org/citation.cfm?id=646381.689680
[6] Anonymous. *(omitted due to double-blind review).*
[7] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2 (2002), 235–256. DOI: https://doi.org/10.1023/A:1013689704352
[8] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2002. The Non-Stochastic Multi-Armed Bandit Problem. (2002).
[9] Author(s) Alfredo Banos, Alfredo Baros, San Fernando, and Valley State College. On Pseudo-Games, The. *Annals of Mathematical Statistics* (????), 1932–1945.
[10] Sébastien Bubeck and Nicolò Cesa-Bianchi. 2012. Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. *Foundations and Trends® in Machine Learning* 5, 1 (2012), 1–122. DOI: https://doi.org/10.1561/2200000024
[11] Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K. Vernon. 2002. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In *Job Scheduling Strategies for Parallel Processing*, Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn (Eds.). Number 2537 in Lecture Notes in Computer Science. Springer Berlin Heidelberg. DOI: https://doi.org/10.1007/3-540-36180-4_7
[12] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. 2016. Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator. In *20th Workshop on Job Scheduling Strategies for Parallel Processing*. Chicago, United States. https://hal.archives-ouvertes.fr/hal-01333471
[13] Dror G Feitelson. 2001. Metrics for parallel job scheduling and their convergence. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 188–205.
[14] Dror G. Feitelson. 2015. *Workload Modeling for Computer Systems Performance Evaluation* (1st ed.). Cambridge University Press, New York, NY, USA.
[15] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. 2005. Parallel Job Scheduling — a Status Report. In *Proceedings of the 10th International Conference on Job Scheduling Strategies for Parallel Processing (JSSPP'04)*. Springer-Verlag, Berlin, Heidelberg, 1–16. DOI: https://doi.org/10.1007/11407522_1
[16] Dror G. Feitelson, Dan Tsafrir, and David Krakov. 2014. Experience with using the Parallel Workloads Archive. *J. Parallel and Distrib. Comput.* 74, 10 (2014), 2967 – 2982. DOI: https://doi.org/10.1016/j.jpdc.2014.06.013
[17] Eitan Frachtenberg and Uwe Schwiegelshohn (Eds.). 2009. *Job Scheduling Strategies for Parallel Processing: 14th International Workshop, JSSPP 2009, Rome, Italy, May 29, 2009. Revised Papers.* Springer-Verlag, Berlin, Heidelberg.
[18] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. 2015. 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)*. ACM, New York, NY, USA, Article 64, 10 pages. DOI: https://doi.org/10.1145/2807591.2807646

[19] David Jackson, Quinn Snell, and Mark Clement. 2001. Core algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing*. Springer.

[20] Ahuva W. Mu'alem and Dror G. Feitelson. 2001. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Trans. Parallel Distrib. Syst.* 12, 6 (June 2001), 529–543. DOI: https://doi.org/10.1109/71.932708

[21] Avi Nissimov and Dror G. Feitelson. 2008. *Probabilistic Backfilling*. Springer Berlin Heidelberg, Berlin, Heidelberg, 102–115. DOI: https://doi.org/10.1007/978-3-540-78699-3_6

[22] D. Perkovic and P. J. Keleher. 2000. Randomization, Speculation, and Adaptation in Batch Schedulers. In *Supercomputing, ACM/IEEE 2000 Conference*. 7–7. DOI: https://doi.org/10.1109/SC.2000.10041

[23] Valentin Reis. 2017. freuk/obandit: Zenodo Release. (March 2017). DOI: https://doi.org/10.5281/zenodo.437875

[24] Valentin Reis. 2017. freuk/ocst: Bandit policy selection paper release. (March 2017). DOI: https://doi.org/10.5281/zenodo.437873

[25] Valentin Reis. 2017. **TODO**. (March 2017). DOI: https://doi.org/10.5281/zenodo.437875

[26] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. 1996. The EASY - LoadLeveler API Project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96)*. Springer-Verlag, London, UK, 41–47. http://dl.acm.org/citation.cfm?id=646377.689506

[27] Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P Sadayappan. 2002. Characterization of backfilling strategies for parallel job scheduling. In *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE, 514–519.

[28] A. Streit. 2002. The self-tuning dynP job-scheduler. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*. DOI: https://doi.org/10.1109/IPDPS.2002.1015662

[29] William R. Thompson. 1933. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* 25, 3/4 (1933), 285–294. http://www.jstor.org/stable/2332286

[30] Michel Tokic. 2010. Adaptive ϵ-greedy Exploration in Reinforcement Learning Based on Value Differences. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*. 203–210.

[31] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. 2005. Backfilling using runtime predictions rather than user estimates. *School of Computer Science and Engineering, Hebrew University of Jerusalem, Tech. Rep. TR* 5 (2005).

[32] Y. Ukidave, X. Li, and D. Kaeli. 2016. Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 353–362. DOI: https://doi.org/10.1109/IPDPS.2016.73

[33] A. Vishnu, H. v. Dam, N. R. Tallent, D. J. Kerbyson, and A. Hoisie. 2016. Fault Modeling of Extreme Scale Applications Using Machine Learning. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 222–231. DOI: https://doi.org/10.1109/IPDPS.2016.111

[34] Christopher John Cornish Hellaby Watkins. 1989. *Learning from Delayed Rewards*. Ph.D. Dissertation. King's College.

8 ARTIFACT DESCRIPTION: [ONLINE (BANDIT) POLICY SELECTION FOR EASY-BACKFILLING]

8.1 Abstract

This appendix describes the computational artifacts associated with this work. We take a lightweight approach to replicability by packaging a declarative workflow that generates all the figures from this paper and making it available publicly. Moreover, questions regarding replication can be freely directed at the authors via e-mail.

All the code associated with this work is released under the [1] license as a persistent zenodo [4] archive at [**?**].

8.2 Description

There are various approaches for making computational experiments 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. This allows for replicating the experiments on clusters where virtualization is not available or kernels are too old for containers old and decreases 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.

8.2.1 Check-list (artifact meta information).

- **algorithm: Scheduling Simulation Workflow**
- **program: Ocaml, R, Bash, Zymake(all sources and dependencies included)**
- **compilation: Ocaml 4.02.3, GNU bash version 4.4.5(1)-release, R version 3.3.2**
- **transformations: Patched nix standard environment.**
- **binary: Most binaries are cached by Nix channels.**
- **data set: Traces from the parallel workload archive, included.**
- **run-time environment: Nixpkgs 17.03**
- **hardware: Dell PowerEdge R730**
- **output: Files**
- **experiment workflow: `zymakefile`, see below**
- **publicly available?: Open-Sourced under the ISC [1] license.**

8.2.2 Obtaining the code. **one-liner : replication**

Running the experiments can be done on any platform equipped with the nix package manager by running:

```
nix-build (fetchTarball ...)
```

8.2.3 Hardware. 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[3]. Access to the experimental machine(s) used in this paper was gracefully granted by research teams from LIG[4] and Inria[5].

The experiments in this paper can run via 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 757G of RAM. For this reason, the experimental campaign workflow was designed to be executed on a single host.

8.2.4 Software dependencies. All the software dependencies of our experiments are automatically managed by Nix.

8.2.5 Datasets. All data from this paper was obtained from the Swf Parallel Workloads Archive [16]. The data is cached in gzipped format in the persistent archive associated with this workflow, should the Parallel Workdloads Archive become unavailable.

8.3 Installation

The experiment dependencies are loaded into the environment by the `nix-build` command.

Nevertheless, the reader interested in using the tools outside the workflow can obtain an environment by running the `nix-shell` command:

```
nix-shell (fetchTarball ...)
```

Or alternatively, if the user is at the root of the extracted archive:

```
nix-shell -A banditSelection
```

This allows to use the following command-line tools:

- ocs: The scheduling simulator used in the experiments. Use `ocs --help` to see all available options.
- ocs-sampler: The workload resampler used in the experiments. Use `ocs-sampler --help` for usage.
- zymake: The zymake [**?**] tool. The workflow can be executed in the current environment in parallel via `zymake -l localhost zymakefile`.

8.4 Experiment workflow

All experiments are tied together using zymake [**?**], a minimalistic workflow system designed for computational experiments[6]. 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 : Data extraction from archives, filtering, 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 both as a persistent zenodo archive at [24] and as a git repository at [**?**].
- analysis : This step runs r code that generates the figures presented in this paper.

8.5 Evaluation and expected result

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

[3] https://www.grid5000.fr
[4] http://www.liglab.fr
[5] http://www.inria.fr

[6] the zymake system is also packaged by our nix expressions.

8.6 Experiment customization

Readers interested with modifying the experiments can obtain the zenodo archive at https://...

The experiments are replicated with the same procedure as before, this time invoking the build command with the local folder:

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

The file tree has the following structure:

```
root
├── zymakefile Zymake workflow
├── default.nix Nix packaging
├── pkgs/ Nix packaging
├── gz/ Data archive
│   ├── ANL-Intrepid.swf.gz
│   └── ...
├── misc/ Visualization and filtering code
├── ocst/ Simulator Code
└── paper/ This article.
```

Most simple workflow modifications (using different traces, changing the sample sizes) can be obtained by modifying the `zymakefile`. For modification

8.7 Notes

This artefact description was prepared according to the guidelines located at http://ctuning.org/ae/submission.html