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),* 9 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Plan :

- Context - message : les machines sont de + en + complexes, les systemes de management restent rudimentaires face a la complexite croissante de l'architecture et des applications.
- les habitudes et usages existants : justification du FCFS-EASY-BF, developper rapidement ce qui existe
- ce que l'on propose

Providing the computing infrastuctures needed to solve actual complex problems arising in the various fields of the modern society (including climate change, health, green energy or security) is a strategic challenge. In particular, High Performance Computing (HPC) systems are evolving to extreme-scale parallel and distributed platforms. The race for always 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. However, there is still a long way to go and many scientific and technical problems to solve by the system management software in order to adapt to such large-scale evolution and the increasing complexity in both architecture and applications. The existing job and resource management sofwares allow to run tens thousands jobs on hundreds thousands cores. They are based on robust and rather simple policies.

The management generates a huge amount of data. During the last few years, there was an explosion of the number of of works at the interface of HPC and bigdata, dealing with learning algorithms (including the authors of this paper [19]). Most of these studies target the process to determine more or less accurately the value of some specific key parameters, with the idea of learning better estimates should improve the performances of the resource manager.

Our idea within this work is to learn how the management system behave within the classical framework of of selecting jobs of the submission queue under EASY-backfilling. Contrarily to most existing studies which consider only one execution trace, we extend the learning on a multiple trace basis. Indeed, learning for a specific trace leads to limited results, customized for one trace on a given platform (what happens if we target another platform?). Moreover, this is usually inefficient to deal with extreme unpredictable events.

1.1 Contributions

2 RELATED WORKS

This section presents related works in the area of scheduling for batch scheduling.

2.1 Scheduling heuristics in HPC platforms

While parallel job scheduling is a well studied theoretical problem [21], the practical ramifications, varying hypotheses, and inherent uncertainty of the problem in HPC have driven practitioners and researchers alike to use and study simple heuristics. The two most popular heuristics for HPC platforms are EASY [29] and Conservative [22] Backfilling.

While Conservative Backfilling offers many advantages [30], it has a significant computational overhead, perhaps explaining why

Conference'17, Washington, DC, USA

DOI: 10.1145/nnnnnnn.nnnnnnn

most of the machines of the top500 ranking [5] still use at the time of this publication a variant of EASY Backfilling.

2.2 EASY

There is a large body of work seeking to improve EASY. Indeed, while the heuristic is used by various resource and job management softwares (most notably SLURM [4]), this is rarely done without fine tunings by system administrators.

Several works explore how to tune EASY by reordering waiting and/or backfilling queues [33], sometimes even in a randomized manner [24], as well as some implementations [20]. However, as successful as they may be, these works do not address the dependency [7] of scheduling metrics on the workload. Indeed these studies most often report *post-hoc* performance since they compare algorithms after the workload is known.

The dynP scheduler [31] proposes a systematic method to tuning these queues, although it requires simulated scheduling runs at decision time and therefore costs much more than the natural execution of EASY.

2.3 Data-aware resource management

There is a recent focus on leveraging the high amount of data available in large scale computing systems in order to improve their behavior. Some works use collaborative filtering to colocate tasks in clouds by estimating application interference [34]. Others are closer to the application level and use binary classification to distinguish benign memory faults from application errors in order to execute recovery algorithms (see [35] for instance).

Several works take a similar approach in the context of HPC, in particular [19, 33], hoping that better job runtime estimations should improve the scheduling [13]. Some algorithms estimate runtime distributions model and choose jobs using probabilistic integration procedures [23].

However, these works do not address the duality between the cumulative and maximal scheduling costs, as mentionned in [19].

A recent work [28] proposes to help users specify better memory requirements by applying supervised learning techniques to job metadata.

While these previous works intend to estimate uncertain parameters, they lack in validating the impact on the system. We consider in this paper a more pragmatic approach, which is to directly learn a scheduling policy from a given seach space.

2.4 Multi-Armed Bandits

A multi-armed bandit (MAB) problem is a sequential allocation problem with partially observable rewards. At every round, an action (an arm) 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 number of rounds. There are a number 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 [32] for the original work on the stochastic case and [8] for the UCB family of algorithms. The work of [10] is the earliest known work to us for the adversarial case see [9] for the EXP family of algorithms. We refer to the review [11] 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 constraits such as the fact that the rewards should be contained in a range. The[8] heuristic called Epsilon-Greedy is known to achieve good results in practice, and does not have this requirement.

3 PROBLEM SETTING

This section presents the systems under study and the scheduling problem. It first introduces the EASY-Backfilling heuristic and gives (fix ugly sentence) the problem statement.

3.1 System Description

The problem addressed in this work is the core logic of Resource and Job Management Systems (RJMS) such as SLURM [4], PBS [3], OAR [12], Cobalt [1], and more recently Flux [14].

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, these softwares need 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 description and the notations of the characteriscs 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 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 [29]. 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.

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

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 can sometimes be dual in practice, making the problem bi-objective. Subsection 4.2 will outline our approach to managing this aspect.

3.4 Problem Statement

Our problem statement is: **How to tune EASY-Backfilling in an online manner?** introduce the difference in workloads, the duality of avg and max, and the online 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 [29].
(2) LCFS: Last-Come First-Serve.
(3) LPF: Longest estimated Processing time First.
(4) LQF: Largest resource requirement q_j First.
(5) LEXP: Largest Expansion Factor First [30], 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.
(6) SEXP: Smallest Expansion Factor First
(7) LRF: Largest Ratio $\frac{p_j}{q_j}$ First
(8) SRF: Smallest Ratio First
(9) LAF: Largest Area $p_j \times q_j$ 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.

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

5 ONLINE TUNING

We present here the strategies we have retained for selecting a policy. In the remainder, 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 length: $\Delta_0, \cdots, \Delta_T$, where T is the index of the current policy period; all periods have length Δ. At the beginning of each period, from the first period Δ_0 to the current period Δ_T, a new policy is selected. We rely here on three different strategies for this selection: complete simulation, noisy simulation, ϵ-greedy bandit exploration. All these strategies are applied online.

5.1 Online policy selection with complete simulation

Several simulators have been developed for "playing" reordering policies on a given set of jobs. Such simulations are interesting inasmuch as they provide an estimate of the cost of a given policy on a set of jobs. TO BE COMPLETED

We further assume here that there is a certain regularity among periods, *i.e.* that the jobs submitted between consecutive periods do not differ radically. This assumption underlies ??? and is of course all the more true than the policy period considered is not too long. It further entails that the behavior of a policy on the previous periods, in particular on the most recent ones, reflects its behavior on the current one. One can thus base the selection of a policy on its behavior on previous periods, as proposed below.

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

$$w_t(P_i) = \sum_{j=1}^{l(\Delta_t)} Wait_j^{P_i}$$

where $Wait_j^{P_i}$ denotes 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:

$$w(P_i) = \frac{1}{T} \sum_{t=0}^{T-1} \left(\frac{l(\Delta_t)}{\sum_{t=0}^{T-1} l(\Delta_t)}\right)^{\alpha} \lambda^{T-1-t} w_t(P_i) \tag{3}$$

where λ is a decay parameter that privileges recent periods. The fraction $l(\Delta_t)/\sum_{t=0}^{T-1} l(\Delta_t)$ allows to balance the periods according to their importance in terms of the number of jobs considered during the period, so as to penalize the influence of "small" periods wrt to "large" ones. The parameter $\alpha \in [0, 1]$ serves here to control this balance (for $\alpha = 0$ large and small periods are considered equal whereas for $\alpha = 1$ small periods are penalized).

5.2 Online policy selection with noisy rewards

When we have an noisy simulator.

5.3 Bandit policy selection

When we have only bandit feedback of the accurate simulator. We use the epsilon-greedy bandit2.

Algorithm 2: Epsilon-Greedy Bandit policy

Input: $0 < \epsilon < 1, K > 1$

1: **for** $t = 1, 2, \ldots$ **do**
2: Let i_t be the reordering heuristic with the highest current average reward.
3: With probability $1 - \epsilon$ use i_t and with probability ϵ use a random reordering heuristic.
4: **end for**

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[17]. 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 [18]. 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 [16]. 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[26] and an accompanying multi-armed bandit library[25]. 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

[1] http://www.cs.huji.ac.il/labs/parallel/workload/logs.html#clean
[2] This filtering step is available in the reproducible workflow[27] as shell script misc/strong_filter

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 used, the analysis code and the associated workflow necessary to replicate all experiments and figures present in this paper. All these parts's dependencies are automatically managed via Nix [15] and we invite the reader to replicate the results.

6.2 Results

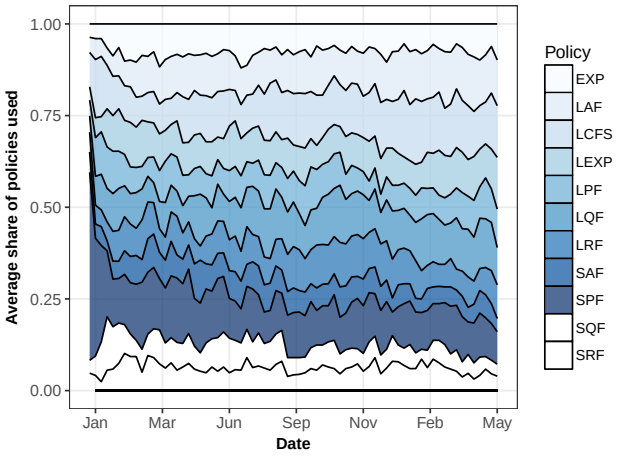

Figure 5: Share of the policies chosen by Epsilon-Greedy as a function of time. The average choice is obtained by resampling the original trace 100 times and aggregating by date.

7 CONCLUSION

ACKNOWLEDGMENTS

Authors are sorted in alphabetical order. We warmly thank Pierre Neyron for help with the experiments. 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[3]. Access to the experimental machine(s) used in this paper was gracefully granted by research teams from LIG[4] and Inria[5].

REFERENCES

[1] Cobalt Online documentation. http://trac.mcs.anl.gov/projects/cobalt/, accessed 03/23/17. (????).
[2] ISC license. https://opensource.org/licenses/ISC, accessed 03/23/17. (????).
[3] PBS Pro 13.0 administrator's guide. http://www.pbsworks.com/pdfs/PBSAdminGuide13.0.pdf. (????).
[4] SLURM Online documentation. http://slurm.schedmd.com/sched_config.html. (????).
[5] TOP500 Online Ranking. https://www.top500.org/. (????).
[6] Zenodo. http://zenodo.org/, accessed 03/23/17. (????).
[7] 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
[8] 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
[9] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. 2002. The Non-Stochastic Multi-Armed Bandit Problem. (2002).
[10] Author(s) Alfredo Banos, Alfredo Baros, San Fernando, and Valley State College. On Pseudo-Games, The. *Annals of Mathematical Statistics* (????), 1932–1945.
[11] 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
[12] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. 2005. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, Vol. 2. IEEE, 776–783.
[13] 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
[14] M. Grondona D. Lipari B. Springmeyer D. Ahn, J. Garlick and M. Schulz. Flux: a next-generation resource management framework for large HPC centers. In *third IEEE Internat. Conf. on Parallel processing Workshops, 9-17, 2014.*
[15] Eelco Dolstra. 2006. *The Purely Functional Software Deployment Model.* Ph.D. Dissertation. Universiteit Utrecht. http://www.cs.uu.nl/~
[16] 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
[17] Dror G. Feitelson. 2015. *Workload Modeling for Computer Systems Performance Evaluation* (1st ed.). Cambridge University Press, New York, NY, USA.

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

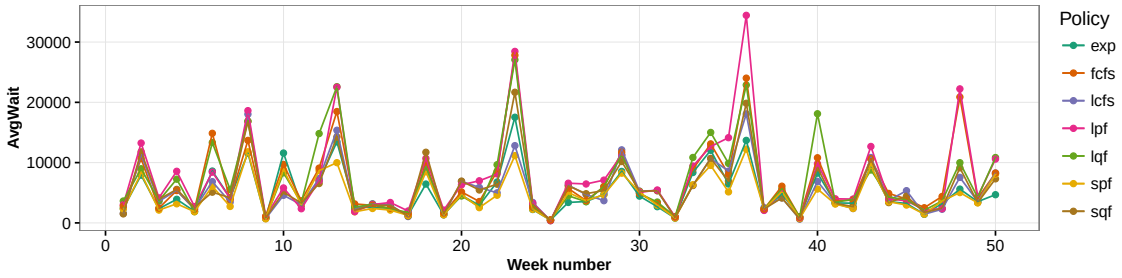

Figure 1: Variability in the weekly average waiting time in the KTH-SP2 trace (pre-processing described in subsection 6.1.1) for a few policies.

[18] 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

[19] 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

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

[21] Joseph YT Leung. 2004. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press.

[22] 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

[23] 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

[24] 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

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

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

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

[28] Eduardo R. Rodrigues, Renato L. F. Cunha, Marco A. S. Netto, and Michael Spriggs. 2016. Helping HPC Users Specify Job Memory Requirements via Machine Learning. In *Proceedings of the Third International Workshop on HPC User Support Tools (HUST '16)*. IEEE Press, Piscataway, NJ, USA, 6–13. DOI: https://doi.org/10.1109/HUST.2016.7

[29] 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

[30] 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.

[31] 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

[32] 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

[33] 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).

[34] 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

[35] 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

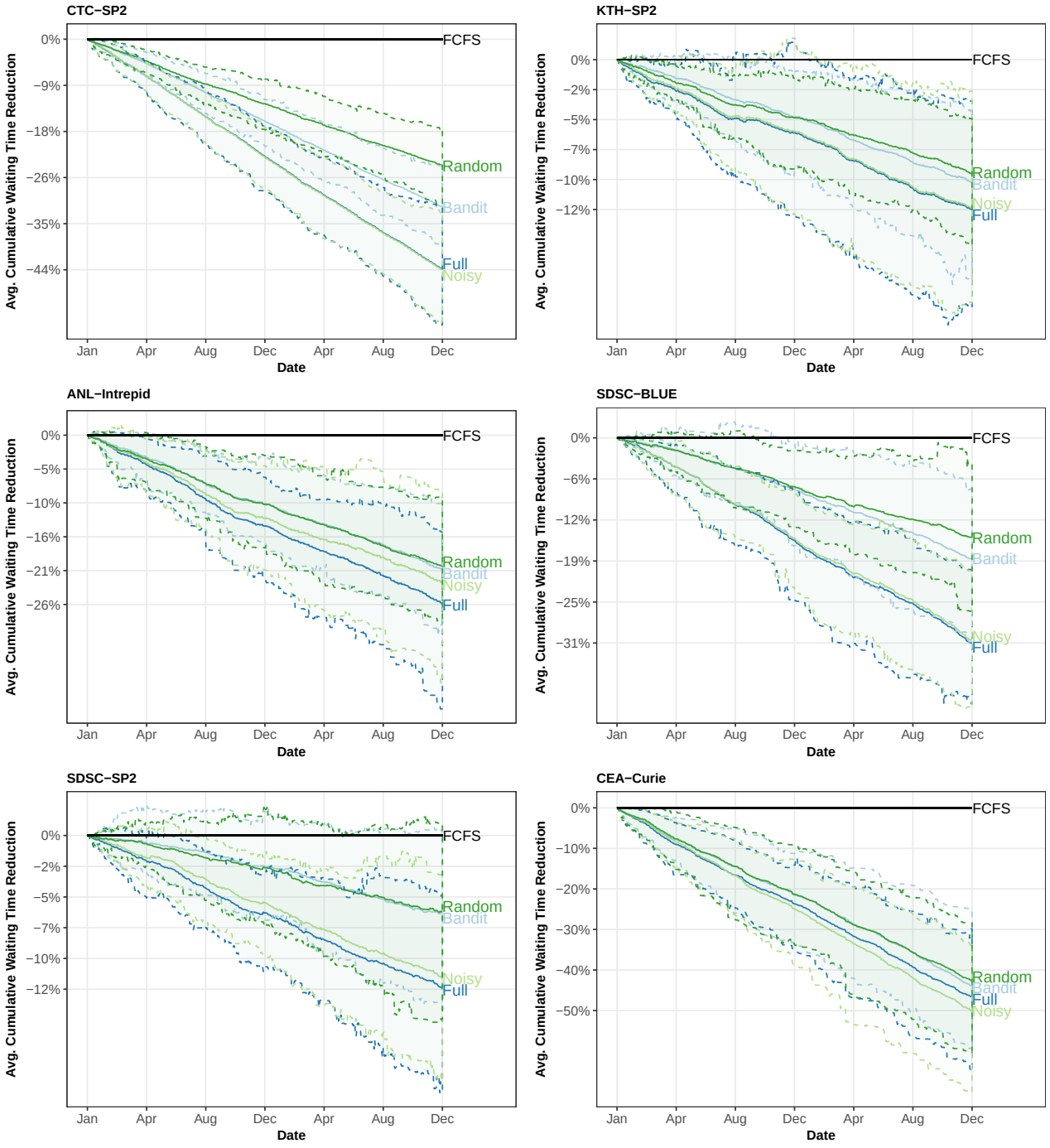

Figure 2: Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the FullFeedback, NoisyFeedback and EpsilonGreedy policies. The average is obtained by resampling the original trace 100 times. The dashed lines represent the 10th and 90th percentiles of the values across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

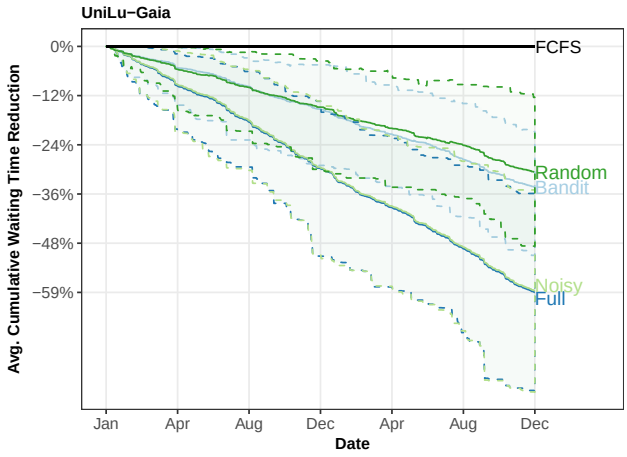

Figure 3: Follow-up from figure 2, plot for the UniLu-Gaia log.

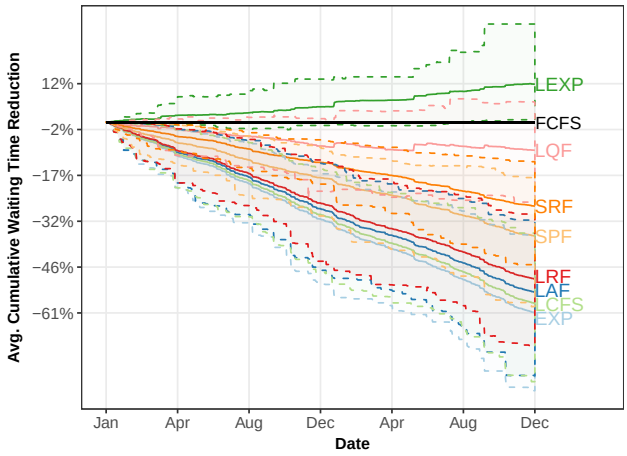

Figure 4: Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the policies P_i for $i = 1 \dots 10$ on the UniLu-Gaia trace. The average is obtained by resampling the original trace 100 times. The dashed lines represent the 10th and 90th percentiles of the values across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

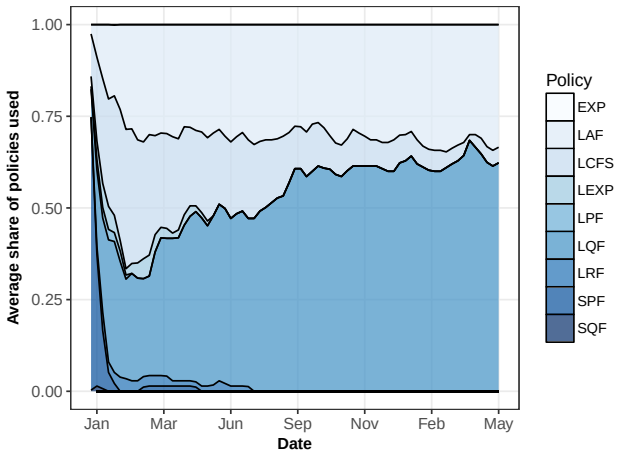

Figure 6: Share of the policies chosen by the Noisy policy. The average choice is obtained by resampling the original trace 100 times and aggregating by date.

ARTIFACTS DESCRIPTION

In this annex, we describe the computational artifacts associated with this work. We take a lightweight approach to replicability by packaging a 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 [2] license as a persistent Zenodo [6] archive at [**?**].

7.1 Packaging and Replicability

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 using the Nix [15] packaging system. 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.

Running the experiments can be done on any platform equipped with the nix workflow system by running:

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

7.2 Testbed

The experiments in this paper can run via in less than 24 hours on a large machine. 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 was designed to be executed on a single host.

7.3 Workflow

All experiments are tied together using Zymake [**?**], a minimalistic workflow system designed for computational experiments. 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 workfloww is composed of the following steps:

Preprocessing : 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 **??**.

Simulation : This step runs the lightweight ocaml backfilling simulator specially written for this work. This simulator is made available under the ISC [2] license both as a persistent Zenodo archive at [26] and as a git repository at [**?**].

Analysis : This step runs mostly R code that generates the figures presented in this paper.