Tuning EASY-Backfilling Queues

Jérôme Lelong, Valentin Reis, Denis Trystram

Tuning EASY-Backfilling Queues.

Jérôme Lelong, Valentin Reis, and Denis Trystram

Univ. Grenoble Alpes, CNRS, Inria, LIG, LJK, France
firstname.lastname@imag.fr

Abstract. EASY-Backfilling is a popular scheduling heuristic for allocating jobs in large scale High Performance Computing platforms. While its aggressive reservation mechanism is fast and prevents job starvation, it does not try to optimize any scheduling objective *per se*. We consider in this work the problem of tuning EASY using queue reordering policies. More precisely, we propose to tune the reordering using a simulation-based methodology. For a given system, we choose the policy in order to minimize the average waiting time. This methodology departs from the First-Come, First-Serve rule and introduces a risk on the maximum values of the waiting time, which we control using a queue thresholding mechanism. This new approach is evaluated through a comprehensive experimental campaign on five production logs. In particular, we show that the behavior of the systems under study is stable enough to learn a heuristic that generalizes in a *train/test* fashion. Indeed, the average waiting time can be reduced consistently (between 11% to 42% for the logs used) compared to EASY, with almost no increase in maximum waiting times. This work departs from previous learning-based approaches and shows that scheduling heuristics for HPC can be learned directly in a policy space.

1 Introduction

The main challenge of the High Performance Computing community (HPC) is to build extreme scale platforms that can be efficiently exploited. The number of processors on such platforms will drastically increase and more processing capabilities will obviously lead to more data produced [10]. Moreover, new computing systems are expected to run more flexible workloads. Seldom supported by the existing managing resource systems, the future schedulers should take advantage of this flexibility to optimize the performance of the system. The extreme scale generates a huge amount of data at run-time. Collecting relevant information is a prerequisite for determining efficient allocations.

The resources of such platforms are usually subject to competition by many users submitting their jobs. Parallel job scheduling is a crucial problem to address for a better use of the resources. Efficient scheduling of parallel jobs is a challenging task which promises great improvements in various directions, including improved machine utilization, energy efficiency, throughput and response time. The scheduling problems are not only computationally hard, but in practice they are also plagued with uncertainty as many parameters of the problem

are unknown while taking decisions. As a consequence, the actual production platforms currently rely on very basic heuristics based on queues of submitted jobs ordered in various ways. The most used heuristic is the well-known EASY-backfilling policy [24, 20]. While EASY is simple, fast to execute and prevents starvation, it does not fare especially well with respect to cumulative cost metrics such as the average waiting time of the jobs. Therefore, many HPC code developers and system administrators intend to tune this heuristic by reordering either the *primary* queue or the *backfilling* queue. Since such reordering of job queues may introduce starvation in the scheduling, this results in a dilemma between the average and maximal costs. In order to solve this dilemma, we introduce a thresholding mechanism that can effectively manage the risk of reaching too large objective values. This issue is further complicated by the dependency of the relative scheduling performances on system characteristics and workload profiles. We propose in this work to use simulations in order to choose queue reordering policies. Finally, we study the empirical generalization and stability of this methodology and open the door for further learning-based approaches.

The rest of the paper is organized as follows: Section 2 reviews existing resource management approaches from the literature. Section 3 describes the context and states the problem. Section 4 describes an experimental setup that is essential to the discussion. Section 5 introduce our approach, illustrating the discussion with results from the KTH-SP2 trace. Section 6 describes the thresholding mechanism used. Section 7 validates this approach using a comprehensive experimental campaign on 5 logs from the Parallel Workload Archive [14].

2 Related Works

This section presents current solutions to the scheduling problem and the current direction taken by the field.

2.1 Scheduling heuristics in HPC platforms

While parallel job scheduling is a well studied theoretical problem [19], 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 [24] and Conservative [21] Backfilling.

While Conservative Backfilling offers many advantages [25], it has a significant computational overhead, perhaps explaining why most of the machines of the top500 ranking [3] 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 [2]), this is rarely done without fine tunings by system administrators.

Several works explore how to tune EASY by reordering waiting and/or backfilling queues [29], sometimes even in a randomized manner [23], as well as some implementations [17]. However, as successful as they may be, these works do not address the dependency [5] 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 [27] 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 [30]. 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 [31] for instance).

Several works use this method in the context of HPC, in particular [29, 16], hoping that better job runtime estimations should improve the scheduling [9]. Some algorithms estimate runtime distributions model and choose jobs using probabilistic integration procedures [22].

However, these works do not address the duality between the cumulative and maximal scheduling costs, as mentionned in [16].

While these previous works intend to estimate uncertain parameters, we consider in this paper a more pragmatic approach, which is to directly learn a good scheduling policy from a given policy space.

3 Problem Setting

This section describes the generic platform model used in this paper. It recalls the EASY heuristic and defines two scheduling cost metrics to be minimized. Finally, it motivates and introduces the problem statement of this paper.

3.1 System Description

The problem addressed in this paper is the one faced by Resource and Job Management Systems (RJMS) such as SLURM [2], PBS [1] and OAR [7] and more recently by Flux [4].

The crucial part of these softwares 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 [24]. This strategy has no worst case guarantee beyond the absence of starvation (i.e. every job will be scheduled at some moment).

The EASY heuristic uses a job queue to perform job starting/reservation (the *primary* queue) and job *backfilling* (the *backfilling* queue). These queues can be dissociated and the heuristic can be parametrized via both a primary policy and a backfilling policy. This is typically done by ordering both queues in an identical manner using job attributes. In the following, we denote by EASY-P_R-P_B the scheduling policy that starts jobs and does the reservation according to policy P_R and backfills according to policy P_B. For the sake of completeness, Algorithm 1 describes the EASY-P_R-P_B heuristic.

This paper makes use of 7 classical queue reordering policies that are presented below:

- FCFS: First-Come First-Serve, which is the widely used default policy [24].
- LCFS: Last-Come First-Serve.
- LPF: Longest estimated Processing time $\widetilde{p_j}$ First.
- SPF: Smallest estimated Processing time $\widetilde{p_j}$ First [25].
- LQF: Largest resource requirement q_j First.
- SQF: Smallest resource requirement q_j First.

Algorithm 1 EASY-P_R-P_B policy

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

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

– EXP: Largest Expansion Factor First [25], where the expansion factor is defined as follows:

$$\frac{wait_j + \widetilde{p_j}}{\widetilde{p_j}} \tag{1}$$

where $wait_j$ is the waiting time until now of job j.

This search set is taken to maximize semantic diversity, without passing judgement on which policy should be the best for a particular objective.

3.3 Scheduling metric

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 reference.

$$\mathbf{Wait}_j = start_j - r_j \tag{2}$$

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**). In the following, we will also use the maximal version of this cost metric which we denote by **MaxWait**, a.k.a the maximal value of the waiting time of all the jobs from a scheduling run.

3.4 Problem Description

There are in the authors' view two main difficulties when effectively tuning the EASY heuristic. Each of these two issues are illustrated below by a dedicated scheduling experiment.

Table 1. AvgWait performance of EASY-EXP-EXP and EASY-SQF-SQF on the original CTC-SP2 and SDSC-SP2 traces, in seconds.

	CTC-SP2	SDSC-SP2
EASY-EXP-EXP	3074	6765
EASY-SQF-SQF	2090	11234

First, the relative performance of EASY policies is sensitive to the context [5, 25]. Table 1 illustrates this effect by comparing the AvgWait of two different queue ordering policies on the logs of two different workloads from the Parallel Workload Archive. The results suggest that there is no "one size fits all" choice of primary and backfilling queue policies. In such a situation, tuning EASY must be done locally for each HPC system. This can be done via simulation, taking care that the results *generalize* to the future.

Table 2. AvgWait and MaxWait performance of EASY-SPF-SPF and EASY-FCFS-FCFS on the original CTC-SP2 trace, in seconds.

	EASY-SPF-SPF	EASY-FCFS-FCFS
AvgWait	2784	3974
MaxWait	661280	176090

Second, starvation may occur when changing the EASY queue policy away from FCFS. This issue concerns the method used to measure the objective. Most systems use a variant of the EASY- FCFS-FCFS policy, where the FCFS policy is used both for primary and backfilling queues. The main advantage of this choice is that it controls the *starvation risk* by greedily minimizing the maximum values of the job waiting times. Indeed, a job might be indefinitely delayed when not starting jobs in the FCFS order. This effect was pointed out in some related works [29, 16] that optimize the average cost by removing the FCFS constraint. Table 2 illustrates this effect by reporting the AvgWait and MaxWait of the EASY-SPF-SPF and EASY-FCFS-FCFS strategies on the CTC-SP2 trace.

In this paper, we would like to study the following question: **How to leverage workload data in order to improve cumulative cost metrics while controlling their maximum values?**

In order to answer this question, we investigate the use of simulation to tune EASY-P_R-P_B by reordering its two queues. The first conclusion is that reordering the primary queue is more beneficial than simply reordering the backfilling queue. However, this introduces a risk on the maximum values of the objective, which we control by hybridizing FCFS and the reordering policy via a thresholding mechanism. Finally, we show that the experimental performance of the thresholded heuristics generalizes well to unseen data.

4 Experimental Protocol

This section motivates the statistical approach used to measure performance and describes the simulation method.

4.1 Statistical approach

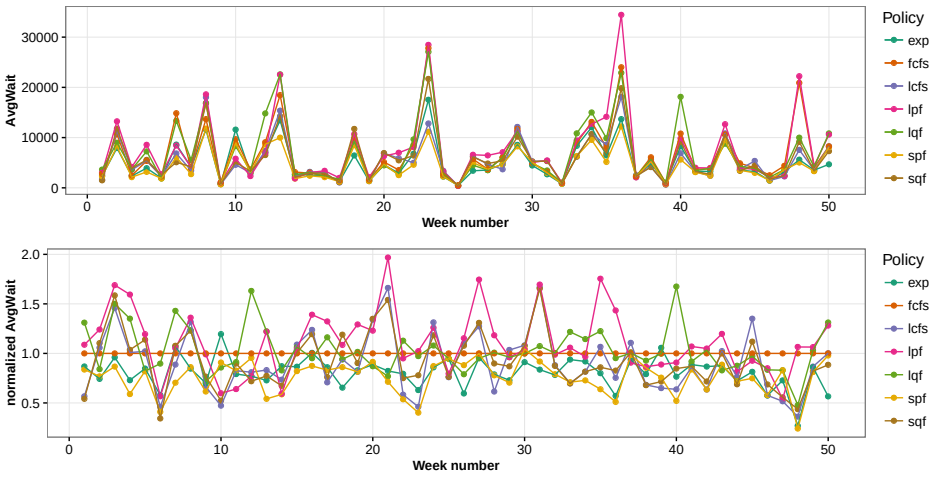

Fig. 1. AvgWait obtained for the 7 main queue policies with **FCFS** backfilling for 150 generated weeks on the KTH-SP2 trace. First, in absolute value, and then normalized with respect to EASY-FCFS-FCFS.

The experimental approach used in this paper is statistical by nature. Figure 1 shows how the AvgWaits of the 7 primary policies used along with FCFS backfilling evolves during the first 150 weeks of the "cleaned"[1] KTH-SP2 trace from the Parallel Workloads Archive. The variability [5, 15] of cost metrics and their sensitivity to small changes in the workload logs [28] have been thoroughly studied in the literature. Our approach to measuring performance without reporting noise from workload flurries [28] is to aggregate the cost metric on a large number of generated logs. In this way, we can report the variability along with the average values. The trace generation approach of this paper follows in part the methodology of [12]: We design a trace resampler in order to generate week-long workload logs from an original dataset. The resampling technique used is simplistic in nature: for each system user, a random week of job submissions from the original trace is used. This approach is combinatorially sufficient to generate infinitely many logs while preserving the natural dependency of the

[1] See the Parallel Workloads Archive [14] for details.

workload on the weekly period and the variability in load. On the downside, the seasonal effect and the dependency between users are lost. Moreover, there is no user model or other feedback loop in the simulations. In all experiments, the performance of every policy is evaluated by averaging the cost values over 250 generated weeks.

4.2 Simulation method and testbed

While high quality simulators like SimGrid [8] are available in practice, this paper focuses on backfilling behavior and does not need to use such advanced tools. This is motivated by the fact that one needs to use a high-performance approach to simulation in order to perform the high number of scheduling runs necessary for this study (the total number of week-long simulations in this paper is of the order of 10^6). Therefore, experiments are run with a specially written lightweight backfilling scheduler. Since there is a need for both speed of execution and generality of application, our scheduler simulator discards all topological information from the original machines. Using this simulator, a week of EASY backfilling can be replayed in under a tenth of a second for the KTH-SP2 machine, the I/O operations (reading and writing a swf file) included. All simulations are performed on a Dell PowerEdge T630 machine with 2x Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz/14 cores (28 cores/node), and 260 GB of RAM. We use a minimalistic approach to reproducible research [26] and provide a snapshot of the work that includes a build system that runs the experiments using the *zymake* [6] minimalistic workflow system. The archive includes our simulator and a nix [11] file that describes the dependencies.

5 Primary and Backfilling queues

This section presents a dedicated experimental campaign that uses the KTH-SP2 trace in order to illustrate the contradictory effect of average and maximum cost.

5.1 Maximum and Average Cost

Figure 2 and 3 show a bi-objective view of the *post-hoc* optimization problem of choosing a primary and backfilling policy among all 49 possible combinations (7 policies for the primary queue and 7 for the backfilling queue). The two objectives are the cumulative and maximal costs. In order to obtain a truthful overview of the variability, we use a sample size of 250 weeks and all values are recentered on the performance of EASY-FCFS-FCFS for that particular week. Figure 2 and 3 vary in terms of y axis. In Figure 2, the y axis is the maximum MaxWait over simulated week, i.e. the highest waiting time of any job on all the simulated weeks. In Figure 3, the y axis is the average MaxWait over the 250 weeks. The average value reported is the mean average cost over individual weeks, which

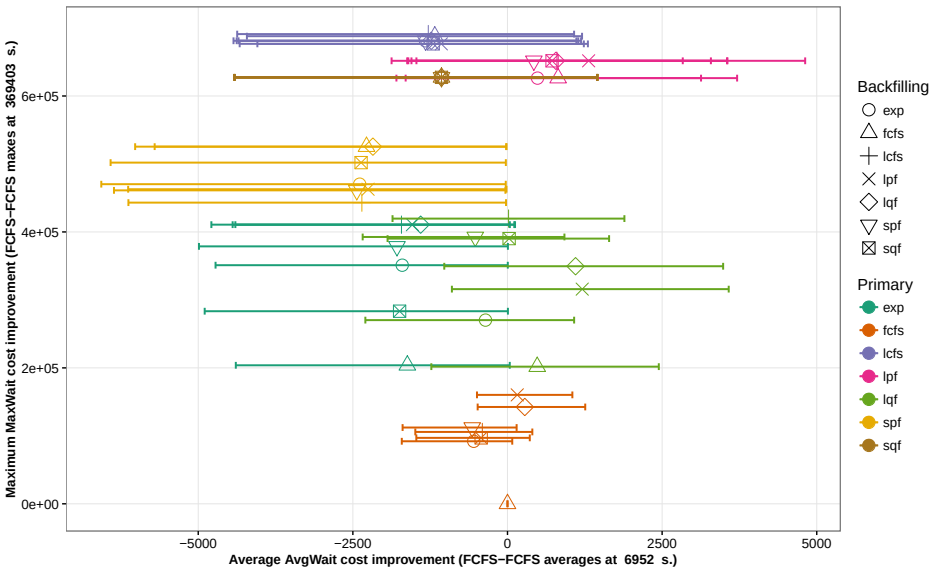

Fig. 2. Maximum and average waiting time cost of the 49 heuristics generated by using the 7 possible policies as primary and backfilling ordering averaged over 250 resampled weeks. All values are relative to the value obtained by the **EASY** primary queue policy with **EASY** backfilling. The maximum MaxWait value reported is the maximum waiting time of all jobs in the 250 weeks. The average AvgWait value is the mean of the weekly waiting time averages, and the range indicates the first and last decile of the samples.

allows for displaying deciles in both directions. Note that Figure 2 is a more aggressive way of reporting this value. There are two main observations.

First, it seems possible to improve the AvgWait on this machine as far as to reduce it of 30% *in hindsight* compared to the EASY-FCFS-FCFS baseline. However, such AvgWait improvements seem to entail an increase in MaxWait. Expectedly, the EASY-FCFS-FCFS heuristic has a good MaxWait behavior.

Second, there seems to be regularities in the performance's behavior: The main factor certainly come from the primary queue policy, while the importance of the backfilling policy varies depending on the primary policy. It appears that some policies such as SQF do not lead to many backfilling decisions, while others like LQF encourage frequent backfilling. Additionally, there are some backfilling policies, such as SPF and ExpFact that systematically outperform the others.

5.2 Comparing Backfilling Policies

It is an interesting question to ask whether some backfilling policies are consistently better than others regardless of primary scheduling policies. As Figure 4 shows, the AvgWait performance of all backfilling policies relative to EASY-FCFS-FCFS presents roughly the same relative performance for each primary

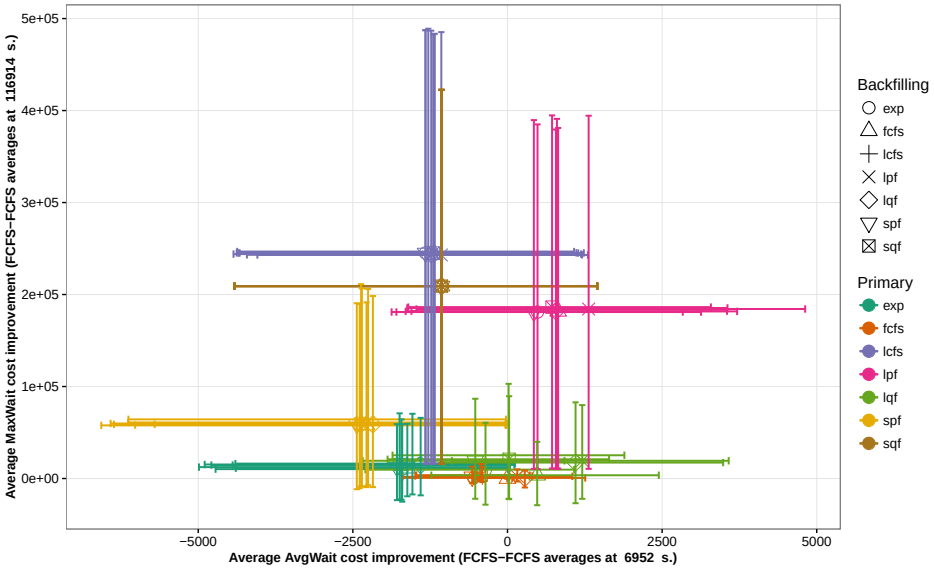

Fig. 3. Maximum and average waiting time cost of the 49 heuristics generated by using the 7 possible policies as primary and backfilling ordering averaged over 250 resampled weeks. All values are relative to the value obtained by the **EASY** primary queue policy with **EASY** backfilling. The average MaxWait value reported is the average of the maximum waiting time over 250 weeks. The average AvgWait value is as in Figure 2 the mean of the weekly waiting time averages, and the range indicates the first and last decile of the samples, both in x and y scale.

queue policy. Namely, for this machine the SPF backfilling policy was always the best from our search space in hindsight. We do not elaborate on this aspect here. In the next section, we focus on the maximal costs incurred by the tuned heuristic.

6 Queue threshold

This section introduces control over the maximal costs using a thresholding mechanism.

6.1 Thresholding and risk

The future costs Wait_j of a waiting job j are lower-bounded at any time t by the value of the waiting time so far, $t - r_j$ [2]. A simple way to introduce robustness into the heuristic is therefore to force jobs with unusually high values of $t - r_j$

[2] Note that this is also valid for the more refined Average Bounded Slowdown [13] metric.

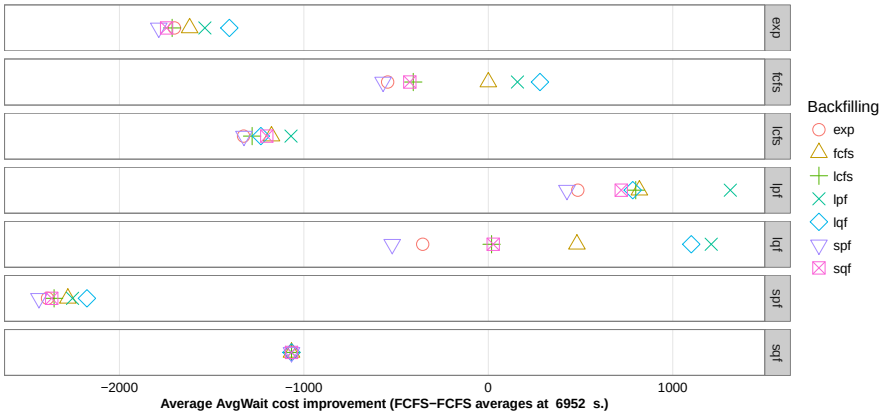

Fig. 4. Performance improvement over EASY-FCFS-FCFS of the 7 Backfilling policies conditioned on Primary policy.

ahead of the primary queue. One way to do this is to introduce a threshold parameter T and push jobs with $t - r_j > T$ immediately ahead of the primary queue after the primary queue sorting step (line 1 of Algorithm 1). If more than one job is in this situation, these jobs are ordered by submission time r_j at the head of the queue.

Figure 5 illustrates the effect on 7 possible heuristics on the KTH-SP2 system with $T = 20$ hours. The heuristics search space is dimished by fixing the backfilling policy to SPF (see Subsection 5.2) for pure visual reasons and exhaustive treatment is delayed to Section 7. The threshold is reported as a horizontal line on the figure. The MaxWait is greatly reduced, while all AvgWait values are (perhaps expectedly) moved torwards EASY-FCFS-FCFS. This mechanism seems to be a hopeful candidate for tuning the queue policies while controlling the waiting time of rogue jobs.

The next section gives a glimpse of the behavior of generalization in this framework.

7 Experimental Validation

This section presents a systematic study of EASY-P_R-P_B tuning.

7.1 Generalization protocol

The goal of the experimental campaign is to study how the performance of different heuristics generalize empirically. That is to say, can EASY Backfilling be tuned on specific workload data? We follow the most simple protocol for assessing learnability:

The initial workload is split at temporal midpoint in two parts, the *training* and *testing* logs. Each of these are used to resample weeks. For each HPC log

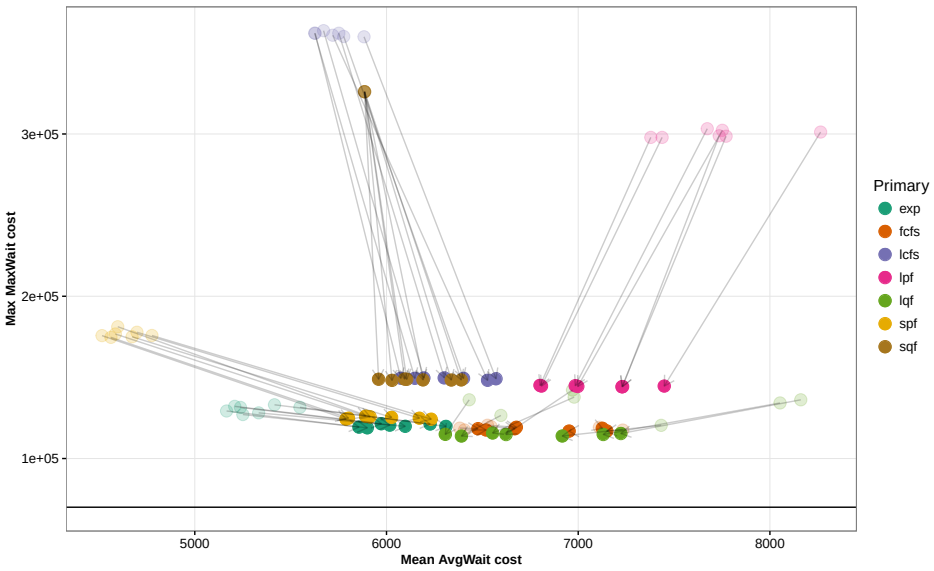

Fig. 5. Maximum and average waiting time cost of the 7 heuristics generated by using the 7 possible thresholded primary policies with SPF backfilling averaged over 250 resampled weeks. The threshold T is chosen at a value of 20 hours. All values are relative to the value obtained by the **EASY** primary queue policy with **EASY** backfilling. The average MaxWait value reported is the maximum waiting time of all jobs in the 250 weeks. The average AvgWait value is as in Figure 2 the mean of the weekly waiting time averages. Semi-transparent points represent the performance of the un-thresholded policies.

from the Parallel Workload archive used in the experiment, this process results in two databases of 250 weeks each. The experimental campaign will consist in running simulations on the training weeks, selecting the best performing policy (tuning the heuristic), and evaluating the performance of this policy on the testing weeks. The search space for EASY-P_R-P_B will be the set of dimension 49 composed by the choice of 7 policies as Primary reordering policy and 7 policies as Backfilling reordering policy.

This simple approach to measuring performance generalization corresponds to the situation where a system administrator having retained usage logs from a HPC center must choose a scheduling policy for the next period.

7.2 Workload logs

Table 3 outlines the five workload logs from the Parallel Workloads Archive [14] used in the experiments. These logs cover both older and more recent machines of varying size and length. The logs are subject to pre-filtering. The filtering step excludes jobs with $\widetilde{p_j} < p_j$ and jobs whose "requested_cores" and "allocated_cores" fields exceed the size of the machine.

Table 3. Workload logs used in the simulations.

Name	Year	# CPUs	# 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
CEA-Curie	2012	80,640	312k	3 Months

7.3 Empirical generalization results

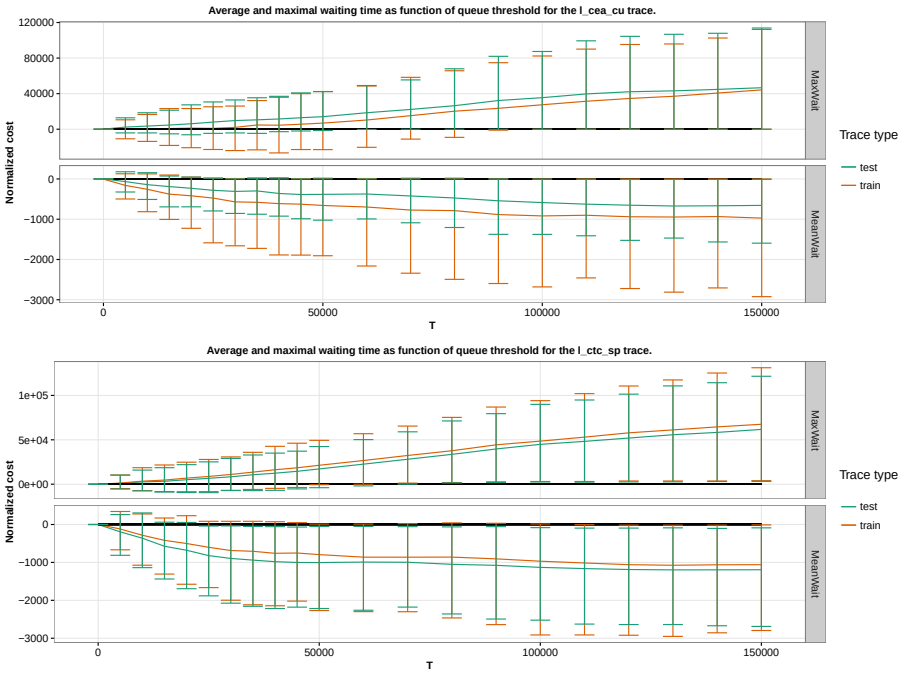

Fig. 6. AvgWait and MaxWait generalization of thresholded policies as affected by the queue threshold. The Value reported as "train" is that of the least costly heuristic among the 49 possible policy parametrizations averaged on the *training* logs. The Value reported as "test" is the averaged cost of the same heuristic on the *testing* logs. This figure is continued as Figure 7.

Figure 7 summarizes the behavior of the empirical generalization and risk of the waiting time with respect to the value of the threshold T. There is a fortunate effect in that the values from the lower parts of the graphs (the *AvgWait* cost) seem to decrease faster than values from the upper part (the *MaxWait* cost), which increases linearly with T.

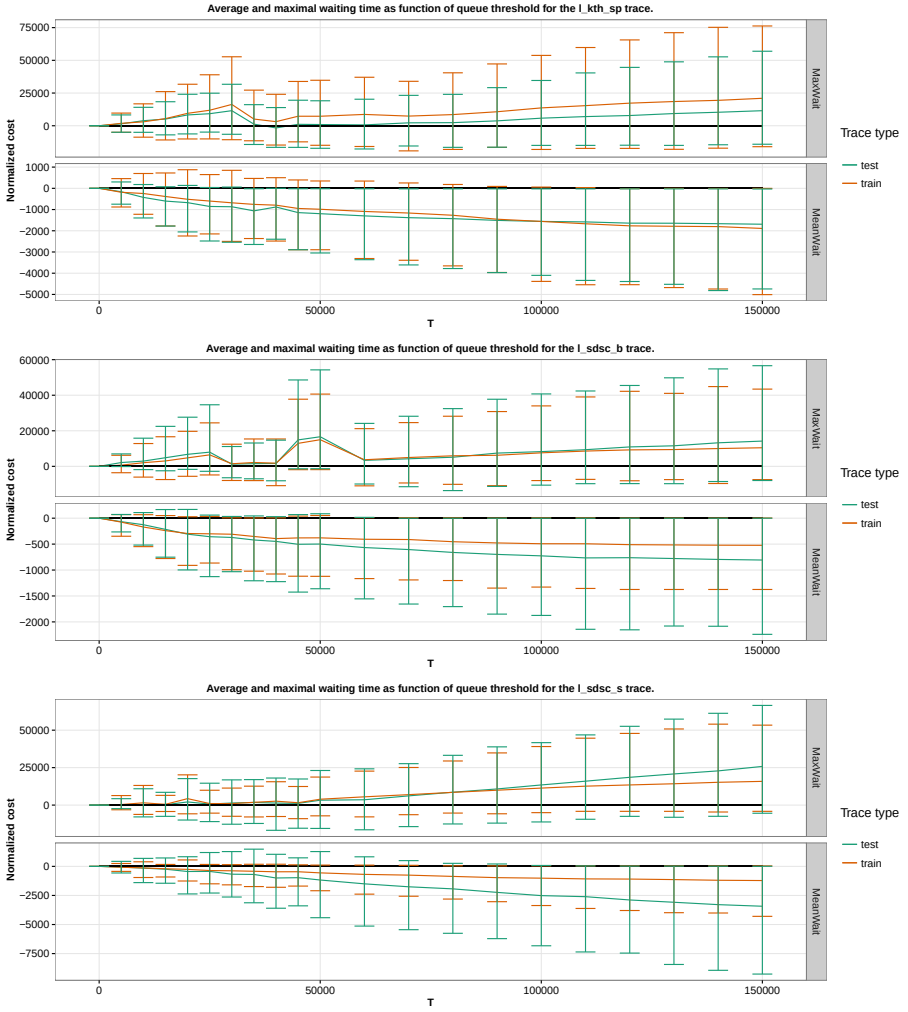

Fig. 7. Follow-up from Figure 7.

By using an aggressive appproach (no threshold), the AvgWait can be reduced until 80% to 65% compared to the EASY-FCFS-FCFS baseline. However, in that case the values of the MaxWait can jump as high as 250% that of the baseline.

By using a conservative approach (thresholding at 20 hours), the AvgWait can be reduced until 90% to 70% in expectation, while keeping the MaxWait increase under 175% of the baseline in all cases.

Figure 8 shows how the AvgWait of the 49 combination of queue and backfilling policies evolve from the training to the testing logs when we use this conservative threshold of 20 hours, with a higher sample size that was not per-

mitted by the previous experiment. This confirms the previous values and gives visual insight into the stability of the performance. Finally, we state the fact that while simplicity of exposure forces us to only deal with the waiting time, the results presented in this work are also valid for the more refined Average Bounded Slowdown [13].

7.4 Generalization with T=20h

The final step is to study how the performance thresholded queue policies generalizes. Figure 8 shows how the performance of all various queue and backfilling policies evolve from *training* to *testing* logs when the threshold is set to an example value of 20 hours. While the values change from *training* to *testing* logs, the relative order of policies seems to be roughly conserved. This leaves hope for generalization. Moreover, it is possible from this figure to measure the improvement resulting from our methodology. We obtain AvgWait average diminutions of 21%, 11%, 36%, 42% and 29% respectively for the SDSC-BLUE, SDSC-SP2, CTC-SP2, CEA-CURIE, and KTH-SP2 machines. The approach does keep the average MaxWait in a reasonable range, and in fact the average testing AvgWait of the learned policy only surpasses that of EASY-FCFS-FCFS on the CEA-Curie trace, with a minor increase, the learned strategy's average MaxWait is of 88747 compared to a value of 86680 for the baseline.

8 Conclusion

This work leverages the fact that the performance of scheduling heuristics depends on the workload profile of the system. More precisely, we investigated the use of simulation to tune the EASY-Backfilling heuristic by reordering its two queues. The first conclusion is that reordering the primary queue is more beneficial than simply reordering the backfilling queue. However, this introduces a risk on the maximum values of the objective, which we control by hybridizing FCFS and the reordering policy via a thresholding mechanism. Finally, we showed that the experimental performance of the thresholded heuristics generalizes well. Therefore, this framework allows a system administrator to tune EASY using a simulator. Moreover, the attitude torwards risk in maximum values can be adapted via the threshold value. With a low threshold value, the increase in maximal cost is small but the learned policy does not take too much risk. It is possible to gain more by increasing the threshold, but this comes with an increase in the maximal cost. Two questions concerning the learning of EASY policies arise from this work.

First, the stability of other EASY heuristic classes remains unknown. The "simple" class of composed of 7 primary policies and 7 backfilling policies (cardinality 49) can generalize using thresholding. It is natural to ask whether it could be possible to learn using a larger set heuristics, such as parametrized queue policies or mixtures of reordering criterias. One could for instance consider the class of mixed policies that choose a job based on a linear combination of the 7

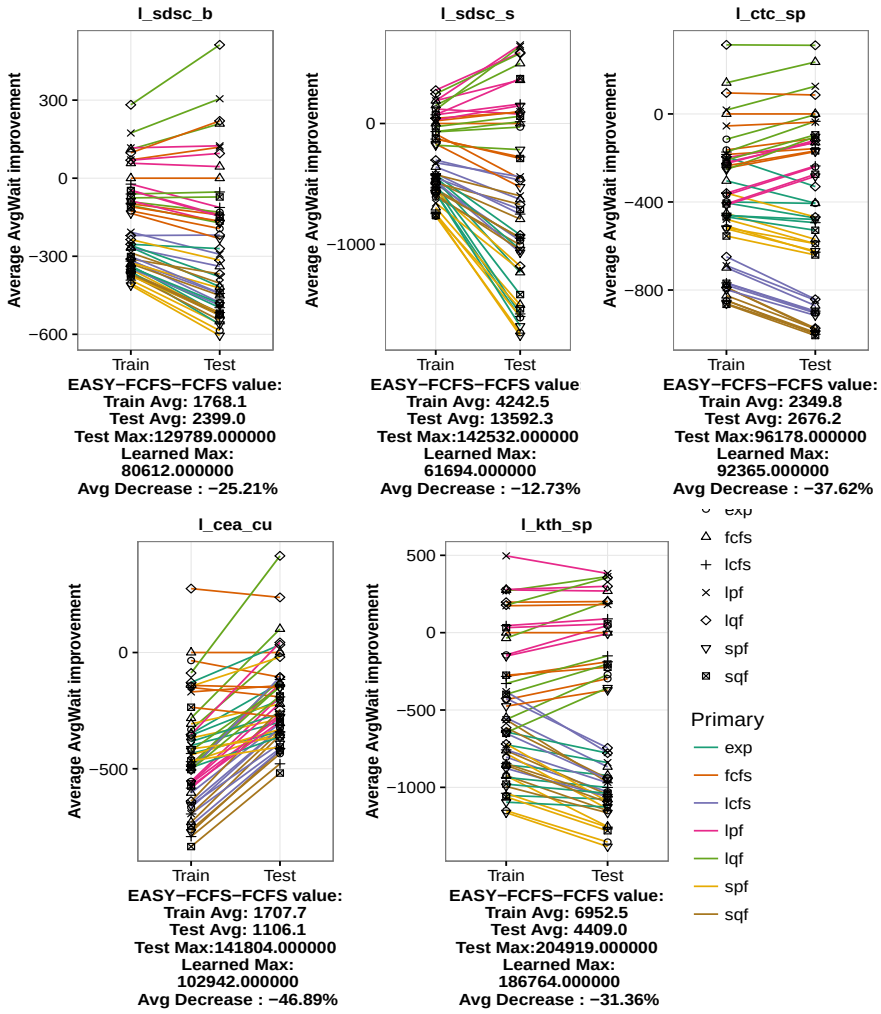

Fig. 8. AvgWait generalization of thresholded policies obtained by using a threshold value of 20 hours. Note that each plot has a different vertical y axis. The reported AvgWait and MaxWait values are averaged over 250 resampled weeks from the training or testing original logs, and we report the difference with the cost of EASY-FCFS-FCFS. The average of the baseline EASY-FCFS-FCFS is reported under the figure, along with the average MaxWait obtained by the best training policy on the testing logs (the "learned" policy).

criteria. A more ambitious endeavor is to ask whether it is possible to learn a contextual job ranking model [18] that performs well.

9 Acknowledgements

Authors are listed in alphabetical order. We warmly thank Eric Gaussier and Frederic Wagner for discussions as well as Pierre Neyron and Bruno Breznik for their invaluable help with 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 of course Dror Feitelson. 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 Digitalis platform[3] of 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[4].

References

1. PBS Pro 13.0 administrator's guide. `http://www.pbsworks.com/pdfs/PBSAdminGuide13.0.pdf`
2. SLURM online documentation. `http://slurm.schedmd.com/sched_config.html`
3. TOP500 online ranking. `https://www.top500.org/`
4. Ahn, D.H., Garlick, J., Grondona, M., Lipari, D., Springmeyer, B., Schulz, M.: Flux: A next-generation resource management framework for large hpc centers. In: 2014 43rd International Conference on Parallel Processing Workshops. pp. 9–17 (Sept 2014)
5. Aida, K.: Effect of job size characteristics on job scheduling performance. In: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing. pp. 1–17. IPDPS '00/JSSPP '00, Springer-Verlag, London, UK, UK (2000), `http://dl.acm.org/citation.cfm?id=646381.689680`
6. Breck, E.: zymake: a computational workflow system for machine learning and natural language processing. In: Software Engineering, Testing, and Quality Assurance for Natural Language Processing. pp. 5–13. Association for Computational Linguistics (2008)
7. Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., Richard, O.: A batch scheduler with high level components. In: CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005. vol. 2, pp. 776–783. IEEE (2005)
8. Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. Journal of Parallel and Distributed Computing 74(10), 2899–2917 (Jun 2014), `http://hal.inria.fr/hal-01017319`
9. Chiang, S.H., Arpaci-Dusseau, A., Vernon, M.K.: The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) Job Scheduling Strategies for Parallel Processing. No. 2537 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jul 2002)

[3] http://digitalis.imag.fr
[4] https://www.grid5000.fr

10. DOE, A.r.: Synergistic challenges in data-intensive science and exascale computing (2013)
11. Dolstra, E., Visser, E., de Jonge, M.: Imposing a memory management discipline on software deployment. In: Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on. pp. 583–592. IEEE (2004)
12. Feitelson, D.G.: Resampling with Feedback — A New Paradigm of Using Workload Data for Performance Evaluation, pp. 3–21. Springer International Publishing, Cham (2016)
13. Feitelson, D.G., Rudolph, L.: Metrics and benchmarking for parallel job scheduling. In: Job Scheduling Strategies for Parallel Processing. pp. 1–24. Springer (1998)
14. Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. Journal of Parallel and Distributed Computing 74(10), 2967 – 2982 (2014), http://www.sciencedirect.com/science/article/pii/S0743731514001154
15. Frachtenberg, E., Feitelson, D.G.: Pitfalls in parallel job scheduling evaluation. In: Job Scheduling Strategies for Parallel Processing. pp. 257–282. Springer (2005)
16. Gaussier, E., Glesser, D., Reis, V., Trystram, D.: Improving backfilling by using machine learning to predict running times. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 641–6410. SC '15, ACM, New York, NY, USA (2015)
17. Jackson, D., Snell, Q., Clement, M.: Core algorithms of the Maui scheduler. In: Job Scheduling Strategies for Parallel Processing. Springer (2001)
18. Joachims, T.: Optimizing search engines using clickthrough data. In: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 133–142. ACM (2002)
19. Leung, J.Y.: Handbook of scheduling: algorithms, models, and performance analysis. CRC Press (2004)
20. Lifka, D.A.: The anl/ibm sp scheduling system. In: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing. pp. 295–303. IPPS '95, Springer-Verlag, London, UK, UK (1995), http://dl.acm.org/citation.cfm?id=646376.689366
21. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Trans. Parallel Distrib. Syst. 12(6), 529–543 (Jun 2001), http://dx.doi.org/10.1109/71.932708
22. Nissimov, A., Feitelson, D.G.: Probabilistic Backfilling, pp. 102–115. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-78699-3_6
23. Perkovic, D., Keleher, P.J.: Randomization, speculation, and adaptation in batch schedulers. In: Supercomputing, ACM/IEEE 2000 Conference. pp. 7–7 (Nov 2000)
24. Skovira, J., Chan, W., Zhou, H., Lifka, D.A.: The easy - loadleveler API project. In: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing. pp. 41–47. IPPS '96, Springer-Verlag, London, UK (1996), http://dl.acm.org/citation.cfm?id=646377.689506
25. Srinivasan, S., Kettimuthu, R., Subramani, V., Sadayappan, P.: Characterization of backfilling strategies for parallel job scheduling. In: Parallel Processing Workshops, 2002. Proceedings. International Conference on. pp. 514–519. IEEE (2002)
26. Stodden, V., Leisch, F., Peng, R.D.: Implementing reproducible research. CRC Press (2014)
27. Streit, A.: The self-tuning dynP job-scheduler. In: Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM (Apr 2002)

28. Tsafrir, D., Feitelson, D.G.: Instability in parallel job scheduling simulation: the role of workload flurries. In: Proceedings 20th IEEE International Parallel Distributed Processing Symposium. pp. 10 pp.– (Apr 2006)
29. Tsafrir, D., Etsion, Y., Feitelson, D.G.: Backfilling using runtime predictions rather than user estimates. School of Computer Science and Engineering, Hebrew University of Jerusalem, Tech. Rep. TR 5 (2005)
30. Ukidave, Y., Li, X., Kaeli, D.: Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 353–362 (May 2016)
31. Vishnu, A., v. Dam, H., Tallent, N.R., Kerbyson, D.J., Hoisie, A.: Fault modeling of extreme scale applications using machine learning. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 222–231 (May 2016)