Learning to control large-scale parallel platforms

Valentin Reis

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 28 septembre 2018

Présentée par

Valentin Reis

Thèse dirigée par **Denis TRYSTRAM**
et codirigée par **Jérôme LELONG**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'École Doctorale **MSTII**

Apprentissage pour le contrôle de plateformes parallèles à large échelle

Learning to control large-scale parallel platforms

Thèse soutenue publiquement le **28 septembre 2018**,
devant le jury composé de :

Michela TAUFER
Professeur, Université du Delaware, États-Unis, Rapporteur
Alfredo GOLDMAN
Professeur, IME, Université de Sao Paulo, Brésil, Rapporteur
Nguyen Kim THANG
Maître de conférences, IBISC, Université Evry Val d'Essonne, France, Examinateur
Émilie KAUFMANN
Chargé de recherche CNRS, CRIStAL, Université de Lille, France, Examinateur
Arnaud LEGRAND
Directeur de recherche CNRS, LIG, France, Président
Jérôme LELONG
Maître de conférences, LIG, Univ. Grenoble Alpes, France, Co-Directeur de thèse
Denis TRYSTRAM
Professeur des universités, LIG, Grenoble INP, France, Directeur de thèse

Abstract

Providing the computational infrastructure needed to solve complex problems arising in modern society is a strategic challenge. Organisations usually address this problem by building extreme-scale parallel and distributed platforms. High Performance Computing (HPC) vendors race for more computing power and storage capacity, leading to sophisticated specific Petascale platforms, soon to be Exascale platforms. These systems are centrally managed using dedicated software solutions called Resource and Job Management Systems (RJMS). A crucial problem addressed by this software layer is the job scheduling problem, where the RJMS chooses when and on which resources computational tasks will be executed. This manuscript provides ways to adress this scheduling problem. No two platforms are identical. Indeed, the infrastructure, user behavior and organization's goals all change from one system to the other. We therefore argue that scheduling policies should be adaptive to the system's behavior. In this manuscript, we provide multiple ways to achieve this adaptivity. Through an experimental approach, we study various trade-offs between the complexity of the approach, the potential gain, and the risks taken.

Résumé

Fournir les infrastructures de calcul nécessaires à la résolution des problèmes complexes de la société moderne constitue un défi stratégique. Les organisations y répondent classiquement en mettant en place de larges infrastructures de calcul parallèle et distribué. Les vendeurs de syst'emes de Calcul Hautes Performances sont incités par la compétition à produire toujours plus de puissance de calcul et de stockage, ce qui mène à des plateformes "Petascale" spécifiques et sophistiquées, et bientôt à des machines "Exascale". Ces systèmes sont gérés de manière centralisée à l'aide de solutions logicielles de gestion de jobs et de ressources dédiées. Un problème crucial auquel répondent ces logiciels est le problème d'ordonnancement, pour lequel le gestionnaire de ressources doit choisir quand, et sur quelles ressources exécuter quelle tache calculatoire. Cette thèse fournit des solutions à ce problème. Toutes les plateformes sont différentes. En effet, leur infrastructure, le comportement de leurs utilisateurs et les objectifs de l'organisation hôte varient. Nous soutenons donc que les politiques d'ordonnancement doivent s'adapter au comportement des systèmes. Dans ce manuscrit, nous présentons plusieurs manières d'obtenir cette capacité d'adaptation. A travers une approche expérimentale, nous étudions plusieurs compromis entre la complexité de l'approche, le gain potentiel, et les risques pris.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisors, Professor Denis Trystram and Associate Professor Jérome Lelong, for their time and uninterrupted trust. In particular, my research interest is entirely due to Denis, whose encounter was a turning point in my academic path. My sincere thanks also go to Professor Eric Gaussier for his guidance and the research opportunities he helped create.

I warmly thank all my co-authors and my fantastic colleagues at IMAG and Inria whose company made these years in Grenoble worthwhile. Lastly, I thank my friends and family for their unfailing support. Various technical and grant acknowledgements follow.

I 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.

Experiments presented in this thesis 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[1]. Access to the experimental machine(s) used in this thesis was gracefully granted by research teams from LIG[2] and Inria[3].

This work has been partially supported by the LabEx PERSYVAL-Lab(ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

[1]https://www.grid5000.fr
[2]http://www.liglab.fr
[3]http://www.inria.fr

Contents

1 Introduction

The applications developed by institutional and industrial research teams have constantly growing computational needs. A solution to provide for these needs is to run computational jobs inside HPC centers, which allow to compute at a greatly increased scale by pooling hardware and human resources. High Performance Computing (HPC) vendors race for more computing power and storage capacity, leading to sophisticated specific Petascale platforms, soon to be Exascale platforms.

The resulting management of such huge hardware resources comes with significant challenges. Some of these challenges are technical and others are administrative. Resource and Job Management Systems (RJMS) such as OAR[**oar**], SLURM[**slurm**] or more recently Flux[**flux**] provide integrated solutions to these problems. Notably, one has to appropriately let users access parts of the machine at specific times, with specific computational environments that suit their needs, all the while enforcing usage, security, and performance policies. Moreover, the execution of the jobs themselves requires some orchestration. The RJMS will sometimes be responsible for providing the input data and retrieving results, and is most often responsible for wrapping jobs in MPI [**mpi**] implementations.

A crucial problem adressed by this software layer is the parallel job scheduling problem, where the RJMS determines when and on which resources computational tasks will be executed. This is a dynamic process, where users may submit jobs into a waiting queue at any time. When free resources become available, the scheduler may decide to allocate a job from the waiting queue. Job submissions come most of the time with an associated resource request and a maximal time request (also called a *walltime*). Schedulers usually try to minimize user-centric quantities such as the average job waiting time, or the average slow-down. The scheduling problem this thesis deals with and these objective metrics are precisely defined in the next chapter.

Sophisticated solvers are not used in practice because of the on-line nature of the problem, the uncertainties inherent to the 'anytime' submission model, and the uncertain running times and the computational hardness of scheduling problems. Indeed, scheduling policies operate under a strong response-time constraint, as decisions must be taken in a real-time fashion in order not to waste system time This

problem is therefore solved using "heuristic" policies, which boil down to variants of priority list scheduling.

Most RJMS implement a default heuristic called EASY-Backfilling. This default heuristic is usually customized by system administrators and researchers. This customization is not trivial, but it is well known that if properly applied, it can result in great improvements with respect to user metrics.

This need for adapting the heuristic exists because the infrastructure, user behavior and organization's goals all change from one system to another. As a consequence, there exist many scheduling policies in the literature, almost as many as existing systems. To be clear, all these differences mean that a scheduling policy that works well for a given system can essentially be useless for another one.

This state of affairs is the main motivation for this thesis. In this work, we study two ways to automate this adaptivity by using feedback from the system. This study is performed via an experimental approach workload traces from various machines and a scheduling simulator. We provide experimental results for these two approaches, which are respectively based on supervised learning and multi-armed bandits, that are conclusive in supporting the claim that adaptivity can be automated.

Chapter **??** presents the problem in detail, including the simulation model we will use and the classical heuristic solutions the rest of the thesis is based on.

Chapter **??** presents the state of the art in resource and job management for HPC. It gives an overview of the family of learning methods that will be used in this manuscript and discusses the available simulation methods for these systems.

Chapter **??** presents the first method we study, which uses supervised learning to augment the scheduling heuristic. We show how an online regression method can be used to integrate system feedback into the scheduling heuristic. The results from this work are encouraging, as the methods consistently reduce waiting times compared to the vanilla EASY-Backfilling algorithm by 5 to 86 percent on 6 workload traces. In this sense, this approach achieves the adaptation we were looking for. However, the approach is reliant heavy use of simulations to nail down the appropriate search space. This is the common theme of hyper-parameter selection that arises in applications of machine learning techniques. Moreover, this approach reveals several drawbacks, such as having poor performance on some platforms compared to related works that were designed specifically designed with these very platforms in mind. As a final caveat, the method shows poor performance in terms of the higher statistical moments of the user metric it should optimize.

Chapter **??** takes a step back and considers the problem of designing a more expressive yet tractable search space for learning. More precisely, this chapter empirically evaluates the reach, learn-ability, and inherent risk of threshold-enabled queue-reordering policies. We show that using a limited set of queue reordering functions along with a threshold operation allows to decrease the average waiting time of jobs by up to 46.89% in a train/test fashion.

Chapter **??** evaluates a practical online approach for learning the scheduling policy in the discrete search space defined in Chapter **??**. More precisely, we study two approaches. One is a simulation-based learning step, and the other uses a multi-armed bandit algorithm for policy selection. This allows to reach between 11% to 60% improvement in the average waiting times of jobs in the case of simulation, and to reach between 8% to 48% using the bandit technique.

Chapter **??** concludes on the results of this thesis and outlines some areas for further investigation.

Background and State of the Art

2

This chapter presents the state of the art in resource and job management for HPC and discusses available simulation methods for these systems. It also gives some background and pointers on the kind of learning methods that will be used in this thesis.

2.1 State of the Art

2.1.1 Resource and Job Management Software

Resource and Job Management Systems (RJMS) provide integrated solutions to the management of high-performance computing machines. The available software have a varying range of properties. The SLURM[**SLURMdocSCHED**] workload manager is perhaps the most popular software, with some other centers using their in-house software [**Cobalt**]. Other notable software are OAR[**OAR**], PBS [**PBSdoc**], Torque [**torque**] and more recently Flux [**flux**].

2.1.2 Scheduling heuristics

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

Parallel job scheduling is an old studied theoretical problem [**Frachtenberg:2009:JSS:1692356**, **Feitelson:2004:PJS:2128864.2128865**] whose practical ramifications, varying hypotheses, and inherent uncertainty of the problem applied to the HPC field have driven practitioners to use simple heuristics (and researchers to study their behavior). The two most popular heuristics for HPC platforms are EASY [**easy**] and Conservative [**mualem_utilization_2001**] backfilling.

While Conservative Backfilling offers advantages such as improved predictibility and exposing job deadline to users, it has a significant computational overhead and is more complex which may explain why popular schedulers such as

SLURM [**SLURMdocSCHED**], MOAB [**etsion2005short**], LoadLeveler [**etsion2005short**], LSF [**etsion2005short**] rather use a variant of EASY-Backfilling. However, these software all allow to tune the basic EASY-Backfilling job ordering rule. The reason for this feature existing is that every system has different requirements and therefore priorities on which job to run first. Scheduling being a hard computational problem however, there is really no guarantee that this ordering rule is meaningful as far as the actual job execution is concerned.

2.1.3 Predictive approaches

There is a recent focus on leveraging the huge 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 interferences [**7516031**]. 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 [**fmodeling**] for instance).

Several works use a predictive approach in the context of HPC, in particular [**Tsafrir_easypp_2005**, **learningruntimes**], hoping that better job runtime estimations should improve the scheduling [**chiang_impact_2002**]. Some algorithms estimate runtime distributions model and choose jobs using probabilistic integration procedures [**Nissimov2008**].

Historically, job running time prediction has been first attempted [**gibbons**] by categorizing the jobs according to a predefined set of rules. Then, statistics based on job categories are used to generate a prediction. In this approach, called *Templates*, a partitioning into templates has to be provided by the job management system or the system administrator. It can be seen as an ancestor of tree-based regression models in which the binning has to be obtained through statistical analysis of the specific system and population and/or discussion with a domain expert. The technique was subsequently adapted [**gibbons-GA**] using a more automatic way (a genetic algorithm evolving template attributes) to generate the rules. These works used minimal, high-level information about jobs similar to what can be found in HPC logs.

There exist other works that use more specialized methods, but require the modeling of the jobs. For instance, Schopf *et al.* predict running time of applications by analyzing their functional structure [**Schopf99usingstochastic**]. Another example is the method developed by Mendes *et al.* which performs a static analysis of the applications [**MendesCompilation**].

A later survey [**ML-predictruntime-survey**] evaluates the use of more recent supervised learning tools. This work focuses on two scientific applications and uses in-depth information about both the jobs (*e.g.* input parameters) and the machines (*e.g.* disk speed). A closely related paper by Duan *et al.* [**CCGworkflow**] proposes an hybrid Bayesian-neural network approach to dynamically model and predict the running time of scientific applications. It uses in-depth information about jobs and their environment as well.

All these previous approaches assume jobs and their running times to be identically distributed and independent (*i.i.d.*), and therefore, they do not leverage dependencies between job submissions. A stochastic model [**hmm**] has been proposed for predicting job running time distributions. By opposition to the previous studies which only used job descriptions, this technique only relies on historical running time information. It treats successive running times of a given user as the observations of a Hidden Markov Model [**rabiner**], and hence it does not use the hypothesis that job submissions should be *i.i.d.*. A more recent study [**tsafrir**] also shows that the average of the last two running times of the job's user is a good predictor of the running time.

Tab. 2.1: Comparison of the various predictive approaches.

Work	Uses recent historical information	Black box
[**gibbons**]	No	Yes
[**Schopf99usingstochastic**]	No	No
[**MendesCompilation**]	No	No
[**ML-predictruntime-survey**]	No	Yes
[**CCGworkflow**]	No	Yes
[**hmm**]	Yes	Yes

2.1.4 Adaptive job scheduling

Several works explore how to tune EASY by reordering waiting and/or backfilling queues [**Tsafrir_easypp_2005**], sometimes even in a randomized manner [**1592720**], as well as some implementations [**Jackson2001**]. However, as successful as they may be, these works do not address the dependency [**variability**] 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 [**streit_selftuning_2002**] scheduler 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.

2.1.5 HPC System Simulation

Simulating a computing infrastructure is never an easy task. Node characteristics in terms of memory, CPU, disk capacity and throughput all have an impact on application performance that has to be modeled. Network effects are also complex and applications may interfer with each other's use of the network. These complexities motivated the development of specialized simulation software for the hardware layer such as SimGrid [**casanova:hal-01017319**]. There also exist Job Scheduling simulators, such as [**batsim**]. These tools are born from the vision of high-fidelity simulation of the underlying systems. In this manuscript, we'll have very high simulation runtime constraints, which will force us to rely on a much coarser platform model.

The next section presents the problem in detail, including the simulation model and the classical heuristic solutions in use on actual systems.

2.2 Background.

2.2.1 Supervised Learning

A Supervised learning problem is a prediction problem where one observes so-called labeled instances, which consist of some points in a known set X along with labels, which resides in a known set Y. Each label is associated with a point. The goal of supervised learning is to appropriately choose a label for new, unlabeled instances (a point in X), in a way that is coherent with previously observed labeled instances. This notion of coherence usually leads to formulate an optimization problem, whose form and algorithmic resolution is in most cases considered a full-fledged research question.

There are a number of theoretical frameworks and solution strategies for such problem statement. Among the most used settings in practice are Statistical Learning [**friedman2001elements**] and Online Learning [**perditionburningflames**], although there exist others such as the Probably Approximately Correct [**feldman2014probably**] framework, the Robust approach [**xu2009robust**], and more.

When a theoretical setting is chosen to guide the design of the problem, optimization algorithms are readily available for some useful problem classes. This is the case of traditional problem classes such as regression (When Y is the set of real numbers) or binary classification (When Y is a set with two values). These problem are indeed known to be tractable even with millions of labeled instances, and the formulations

arising from the various frameworks are often similar or identical. Often, these frameworks suggest modeling the task with a convex problem [**bubeck2015convex**], which are most times easily solved.

Even with this restriction, there are usually a good number of design questions that require careful analysis. What should be the convex problem formulation for a specific application? The practicioner has to choose a loss function and perhaps some constraints, and this is not an easy task. These considerations will be made more concrete in Chapter **??**, which will use a regression model to predict the running time of HPC jobs.

2.2.2 Multi-Armed Bandits

A Multi-Armed Bandit (MAB) is a sequential allocation problem with partially observable rewards [**bubnow**]. At every round an action, called an "arm", must be chosen from a fixed set and the corresponding reward is observed. The goal of a MAB algorithm is to maximize the total reward obtained in a successive number of rounds. This is essentially achieved using a combination of explorative actions, which help to estimate the quality of each arm, and of exploitative actions, which leverage the current estimates by using a good arm. This problem can be addressed using various hypotheses. Most common are the (original) stochastic setting and the adversarial setting of regret minimization. See [**thompson**] for the original work on the stochastic setting and [**Auer2002**] for the ”Upper Confidence Bound“ family of algorithms. To the best of our knowledge, [**banos**] is the earliest for the adversarial setting; see [**nonstoch**] for the “Exponential-weight” family of algorithms. We refer to [**bubnow**] for a comprehensive review of the field. While these algorithms bound the cumulative difference in loss to the best arm (the regret), they have functional constraints such as the fact that the rewards should be contained in a range. The simple heuristic called *Epsilon-Greedy* introduced in [**Auer2002**] does not have this requirement. Chapter **??** will use this simple MAB approach to select a scheduling policy.

3

Problem Description

The object of study of this thesis is the problem of allocating resources in High Performance Computing (HPC) platforms. The core resource management question is the following: How to share the machine? By the time of writing this manuscript, most (if not all) existing systems achieve this using a job reservation queue. In this framework, a user may submit computational jobs at any point in the operation of the system. A job consists of a resource request, which may include requirements for nodes such as CPU power, memory, or network bandwidth, a time duration, and a script which executes the computational task at hand. These jobs are then queued for allocation on the platform until the resource request can be satistfied. Jobs from all users compete for the same resources, and a specialized software layer manages this aspect.

3.1 Resource and Job Management Systems

This software layer, called the Resource and Job Management System (RJMS) and often refered to as a Batch Scheduler, is responsible for a number of administrative operations. These software are responsible for automating the administration and monitoring tasks which are needed for the ”job“ abstraction to be realized on the hardware. In most systems, jobs are given exclusive access to the computing resources they require during all their duration. The RJMS deals with the user credential, filesystem mounting, network and data providing aspect of jobs. However, the crucial part of RJMS is the scheduling algorithm that determines where and when the submitted jobs are executed. Figure **??** illustrates the role that the RJMS plays in a HPC system.

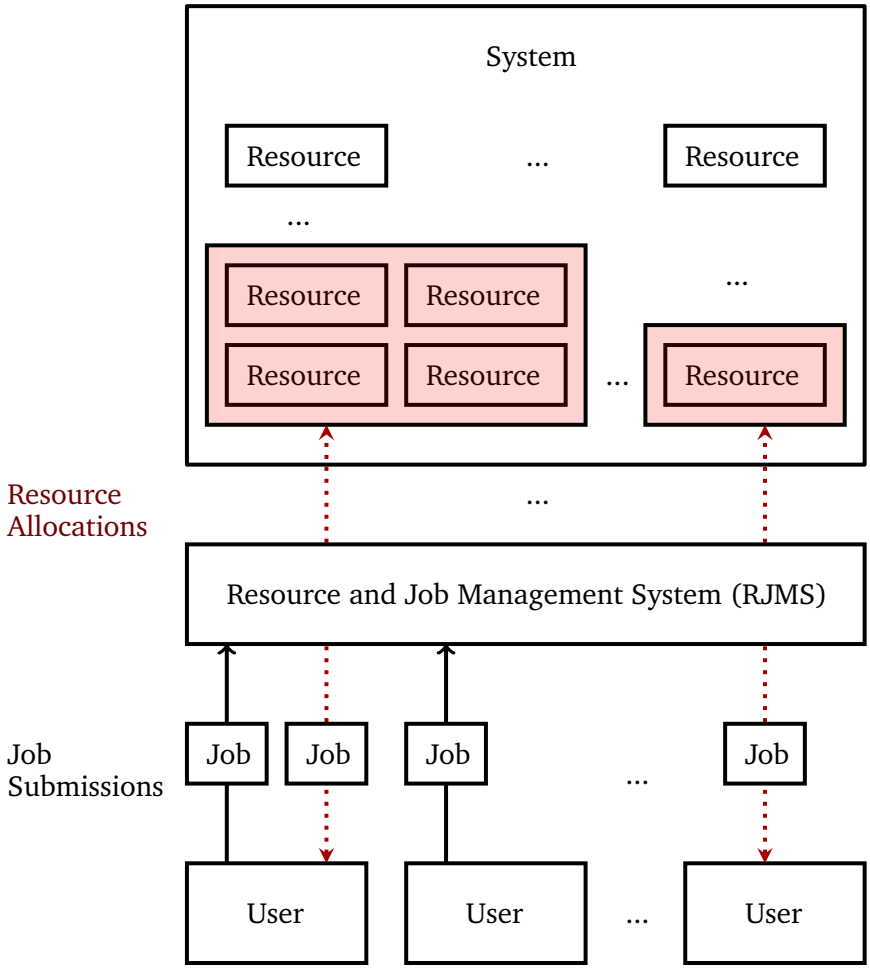


Fig. 3.1: Role of the Resource and Job Management System(RJMS). The RJMS manages the platform by giving users restricted access to some resource(s) during some amount of time. Both the resource request and allocated time are the result of the RJMS's decision, which is based on the Job request.

.

The scheduling 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 (also called walltime) and some resource requirements (number and type of processing units). Then, the Resource and Job Management System (RJMS) drives the search for the resources required to execute this job. Figure **??** illustrates the job submission process. Finally, the tasks of the job are assigned to the chosen nodes. In this thesis, we study this scheduling problem from an experimental perspective.

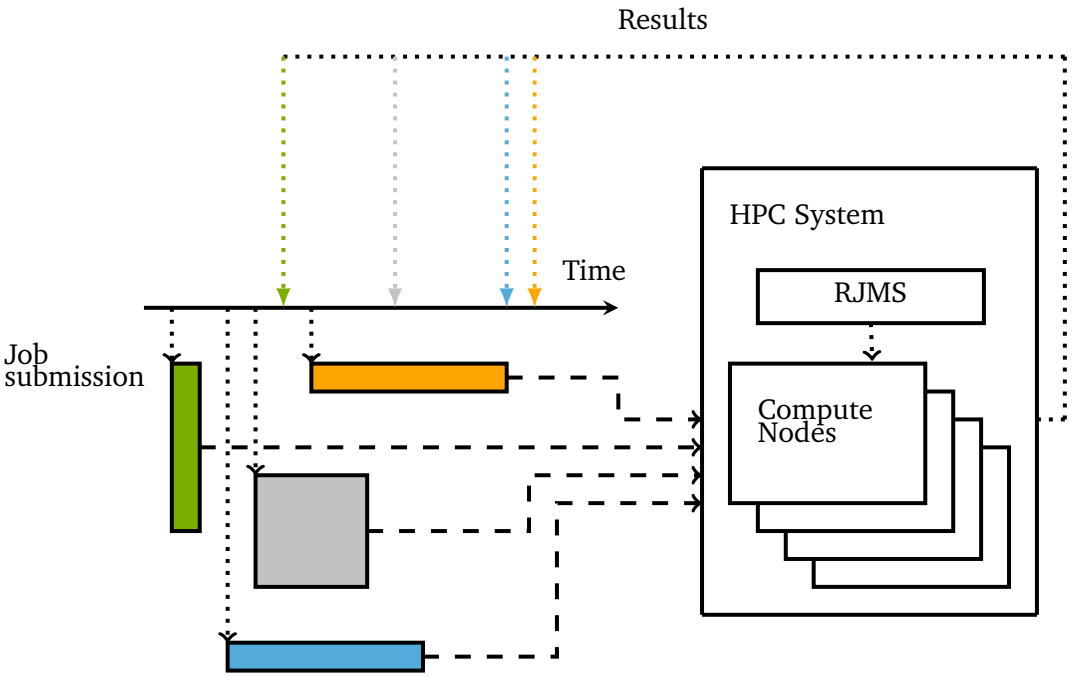


Fig. 3.2: The job submission process. Jobs are submitted anytime by users and scheduled by the RJMS, then executed.

3.2 The Scheduling Problem

We are interested in this thesis in scheduling jobs in HPC platforms. Application developers or users submit their jobs in a centralized waiting queue. The job management system aims at determining a suitable allocation for the jobs, which all compete against each other for the available computing resources. In most HPC centers, these users are requested to provide some information about their applications in order to help the system to take better decisions. In particular, it is expected that they give an estimation of the running time of their jobs.

In the classical case, the management software needs to execute a set of concurrent parallel jobs with rigid (known and fixed) resource requirements on an HPC platform represented by a pool of m identical resources (Usually CPUs). 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 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 (also called *processing time*)
- Requested running time $\widetilde{p_j}$ (sometimes called *walltime*), which is in general an upper bound of p_j.

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

$$\textbf{Wait}_j = \text{Start}_j - r_j \tag{3.1}$$

where Start_j is the starting time of job j on the system. Figure **??** illustrates this job model.

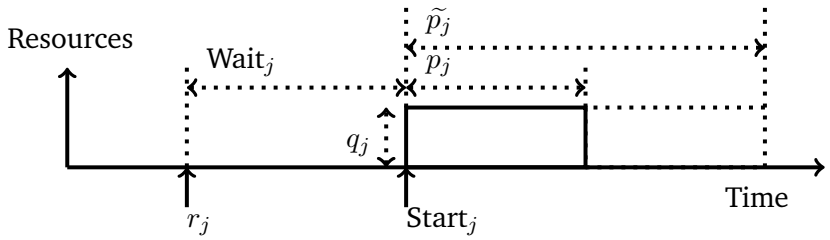
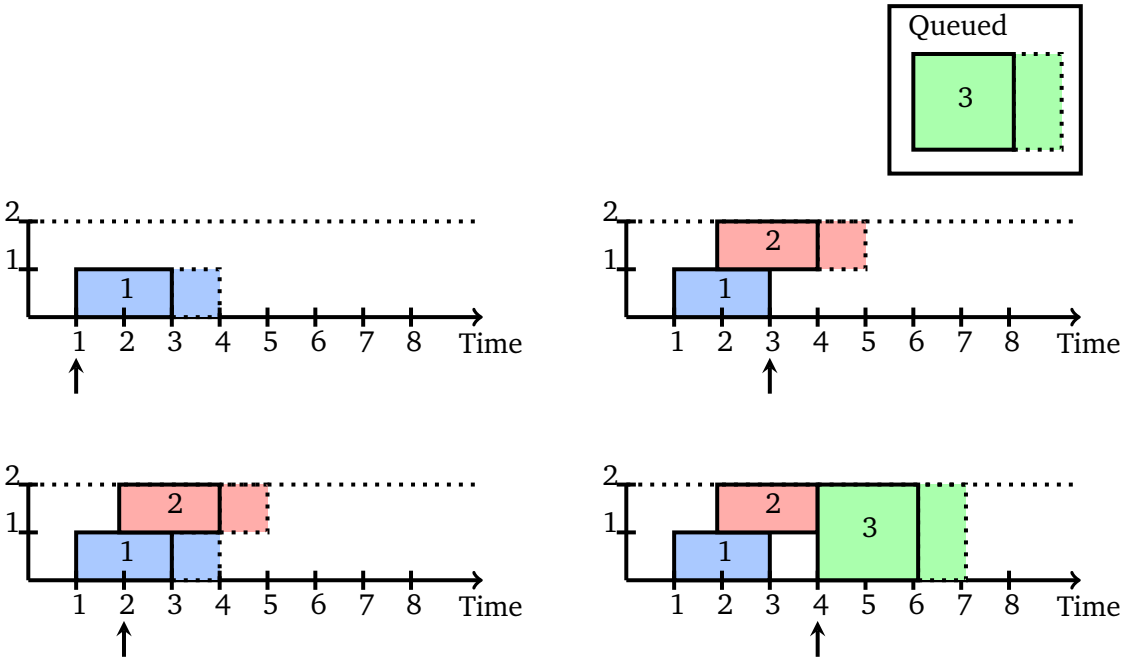


Fig. 3.3: Illustration of the job model.



j	p_j	$\widehat{p_j}$	q_j	r_j	$Start_j$
1	2	3	1	1	1
2	2	3	1	2	2
3	2	3	2	3	4

Fig. 3.4: Illustration of the scheduling process (time steps 1 to 4) using a simple 'first-come, first serve' policy. This policy waits for space to be available for the oldest job in the waiting queue, and starts it. The characteristics of the three job submissions are given in the table above.

3.3 Scheduling Objectives

There are various goals when scheduling jobs on HPC platforms. The first requirement is usually that jobs should not *starve*. That is, it should be impossible for a job's starting date to be indefinitely delayed. Moreover, a system administrator may use one or multiple cost metric(s) to measure the performance of the system.

The *stretch* of a job j is a popular tool:

$$\textbf{Stretch}_j = \frac{\textbf{Wait}_j + p_j}{p_j} \tag{3.2}$$

The stretch metric is usually considered as its more refined version called the *bounded slowdown*:

$$\textbf{Bsld}_j = \max\left(\frac{\textbf{Wait}_j + p_j}{\max(p_j, \tau)}, 1\right) \tag{3.3}$$

where τ is a threshold used to reduce the impact of small jobs on the metric [**feitelson1998metrics**].

Like other cost metrics, both waiting time and bounded slowdown are usually considered in their *cumulative* version, which means that the job scheduler should seek to minimize the average metric over a period of time where N jobs were submitted:

$$\frac{1}{N}\sum_{j=1}^{j=N}\mathbf{Wait}_j, \quad \frac{1}{N}\sum_{j=1}^{j=N}\mathbf{Bsld}_j \tag{3.4}$$

Other objective metrics besides the average view also have their place. The maximum metrics among N jobs that were submitted during a time period are also worthy of interest:

$$\max_{j=1\ldots N}\mathbf{Wait}_j, \quad \max_{j=1\ldots N}\mathbf{Bsld}_j \tag{3.5}$$

Indeed, this metric is a way to measure how well the *no starvation* constraint is enforced in practice by the scheduler. Unfortunately, this criteria is often antagonistic with the average waiting time.

3.4 Resampling

The previously introduced scheduling metrics exhibit a high variability when calculated on real workload traces. This is a well studied phenomenon [**feitelson2001metrics**, **frachtenberg2005pitfalls**]. While the parallel job scheduling community has traditionally focused on using generative models [**feitbook**] in order to drive sample sizes up, there is a recent focus on the use of resampling tools [**feitresampling**]. In this manuscript, we will make use of two different resampling techniques, detailed in Chapter **??** and Chapter **??**.

4 Runtime Prediction

This section presents a first solution based on a predictive approach. The main idea behind this approach is to attempt to reduce the uncertainty in the input parameters of the scheduling problem before applying the EASY-Backfilling scheduling heuristic. The EASY-Backfilling heuristic therefore will have to be modified in consequence.

The objective of this section is to show by some simulations on actual data that using good running time predictions significantly improves the scheduling performances. First, let us clarify the vocabulary: a job is *over-predicted* if its predicted running time is greater than the actual running time. Similarly, a job is under-predicted when the predicted running time is lower than the actual running time.

Table **??** reports the comparison of simulations based on the testbed detailed in Section **??**. Each log runs with EASY, first with the original user's requested running times, then with the actual running times (as if the users were entirely clairvoyant). The metric used for comparison, AVEbsld is described in Subsection **??**.

The results reported in Table **??** emphasize that clairvoyant simulations are in average 27% better than simulations using the original requested running times. As running times are shorter in the clairvoyant case, more jobs can be backfilled and thus, the performances are improved. Taking into account actual running time values always improves the scheduling performances, thus accurate running time estimates are crucial for reaching good performances.

Tab. 4.1: AVEbsld performances of EASY (using requested times) and EASY-CLAIRVOYANT (using actual running times). Values between parentheses show the corresponding decrease in AVEbsld.

Log	EASY	EASY-CLAIRVOYANT
KTH-SP2	92.6	71.7 (22%)
CTC-SP2	49.6	37.2 (25%)
SDSC-SP2	87.9	70.5 (19%)
SDSC-BLUE	36.5	30.6 (16%)
Curie	202.1	69.9 (65%)
Metacentrum	97.6	81.7 (16%)

4.1 Prediction method

We first outline in this section the characteristics that the prediction method should have, prior to describing our approach in detail.

4.1.1 Rationale

Let us first argue that an approach based on machine learning for predicting the running times of jobs should have the following characteristics:

It should be based on minimal information. In hope that results extend well to new HPC systems and be usable in mainstream job management system, a learning-based system should prove its effectiveness on minimal job descriptions. In this light, one reasonable approach is to use information of the Standard Workload Format (*SWF*) [**workloadarchive**]. found at the Parralel Workload Archive.

It should work on-line. This is motivated by previous studies which emphasize that dependencies between job running times are so far from being *i.i.d.*[1] that *two successive running times* [**tsafrir**] are enough to predict running time with good accuracy. Therefore, an algorithm based on batches (which re-approximates the learned concept once every k jobs, where $k \gg 1$) should not be favored.

It should use both job description and system history Unlike previous studies that either assume jobs to be temporally independent or rely exclusively on running time locality [**hmm**], a holistic approach should leverage both job descriptions and temporal dependencies.

It should be robust to noise. Because HPC jobs can have an erratic behavior (*e.g.* they may fail or hang-up), the data one is relying on will be noisy. The learning algorithm should be robust to this noise and avoid overfitting.

It should accept an arbitrary loss function. Attempting to minimize the cumulative loss of an arbitrary function allows for a "declarative" statement of the harm incurred by a misprediction. This last aspect is motivated by the intuition that an inaccurate prediction of a job's running time would not harm the scheduling in an identical way depending on the job's characteristics and the direction of the error. This will be explored in detail in the next Subsection. Arbitrary loss functions generally pose computational limitations, and in practice convex loss functions are often used.

We now turn to the description of the prediction method.

[1] That is, *independently and identically distributed*.

4.2 Introduction

4.2.1 Context

Large scale high performance computing (HPC) platforms are becoming increasingly complex. There exists a broad range and variety of HPC architectures and platforms. As a consequence, the *job management system* (which is the middleware responsible for managing and scheduling jobs) should be adapted to deal with this complexity. Determining efficient allocation and scheduling strategies that can deal with complex systems and adapt to their evolutions is a strategic and difficult challenge.

More and more data are produced in such systems by monitoring the platform (CPU usage, I/O traffic, energy consumption, *etc.*), by the job management system (*i.e.*, the characteristics of the *jobs* to be executed and those which have already been executed) and by analytics at the application level (parameters, results and temporary results). All this data is most of the time ignored by the actual systems for scheduling jobs.

The technologies and methods studied in the field of *big data* (including Machine Learning) could and must be used for scheduling jobs in the new HPC platforms.

For instance, and this will be the focus of this chapter, the running time of a given job on a specific HPC platform is usually not known in advance and moreover, it depends on the context (characteristics of the other jobs, global load, *etc.*). In practice, most job management systems ask the users for an upper bound on the job running time, threatening to kill it if it exceeds this requested value. This leads to very bad estimates of the running times given by the users [**userestimate**]. A precise knowledge of the running times is even more important since the algorithms used in most of these systems assume that this value is perfectly known. Thus, the question of how to estimate the running times of jobs seems worth exploring.

Obviously, the job running time is not the only parameter impacted by uncertainties. We focus on it as a proof of concept in order to show that it is possible to improve the scheduling performances for popular FCFS-BF (First Come First Serve with Backfilling) batch scheduling policy. The analysis provided in this chapter can be extended easily to other scheduling policies.

4.2.2 Contributions

The main question addressed in this chapter is to determine to what extent predictions of the running times may help for obtaining a better schedule. For this purpose, we rely on on-line regression methods and consider a family of loss (or cost) functions that are used to learn the prediction model. Then, we show how to use the predictions obtained by improving popular scheduling algorithms. Finally, we perform an experimental evaluation of the proposed new algorithms using six actual traces. The results show an average gain of 28% compared to the classical EASY policy (with a maximum of 86%) and 11% in average compared to EASY++.

4.2.3 Content

Section **??** presents a method for estimating the running time of the jobs based on linear regression. Then, we present the studied scheduling algorithms and their adaptation to predictive techniques in Section **??**. Section **??** reports the simulations done with actual log data. They show that some combinations of prediction and scheduling algorithms improve the system performances.

4.2.4 A new prediction approach

A job j is represented by a vector[2] $\mathbf{x}_j \in \mathbb{R}^n$ where n is the number of features of the model. The features which we feed the algorithm with are described in Table **??**. These features are taken from various sources of information, such as the job's description ($\widetilde{p_j}$ and q_j). Others are taken from historical information (*e.g.* $p_{j-1}^{(k)}$) and some are taken from the current state of the system (*e.g.* Jobs Currently Running). Additionally, some features are taken from the environment (*e.g.* Time of the day).

The prediction is achieved via a ℓ_2-regularized polynomial model. This choice is motivated by the availability of highly robust algorithms to fit on-line linear regression models [**nag**], even in the presence of an adversary scaling of the features. The ℓ_2 norm regularizer is used to prevent the quadratic model from overfitting and the polynomial representation (here of degree 2) to take into account dependencies between features. The regression function is:

$$f(\mathbf{w}, \mathbf{x}) = \mathbf{w}^\intercal \Phi(\mathbf{x}) \quad \mathbf{w} \in \mathbb{R}^{1+2n+\binom{n}{2}} \tag{4.1}$$

[2] As common in machine learning, we use bold letters to denote vectors and standard letters for scalars.

where the w_i are the parameters (to be learned) of the model, and Φ is a vector of basis functions:

$$\Phi(\mathbf{x}) = (1, x_1, \cdots, x_n, x_1x_2, \cdots, x_kx_l, \cdots, x_{n-1}x_n)^\intercal$$

Denoting the actual running time of job j by p_j, the cumulative loss for up to the N-th already-processed job is[3]:

$$\sum_{j=1}^{N} \mathfrak{L}(\mathbf{x}_j, f(\mathbf{x}_j), p_j)$$

where $\mathfrak{L}(\mathbf{x}, f(\mathbf{x}), y)$ is the loss function associated with predicting a value of $f(\mathbf{x})$ in the case where the actual running time is y. The regression problem finally takes the form:

$$\arg\min_{\mathbf{w}} \sum_{j=1}^{N} \mathfrak{L}(\mathbf{x}_j, \mathbf{w}^\intercal \Phi(\mathbf{x}_j), p_j) + \lambda ||\mathbf{w}||_2 \tag{4.2}$$

where λ is the regularization parameter. Once $\mathbf{w}$ has been learned, new running times are predicted through Equation (**??**).

The choice of the loss function is crucial and not straightforward here as the impact of a bad prediction on the running time varies from job to job as well as from the direction of the error (over- or under-prediction). Indeed, scheduling algorithms behave differently with respect to under-prediction and over-prediction: in the case of an under-prediction, a destructive effect on the planned schedule can happen, while an over-prediction never makes a planned schedule feasible but may imply unused CPUs. This suggests that one should rely on asymmetrical losses, that can be based on standard loss functions dedicated to either under- or over-prediction. Another source of complication with respect to prediction arises not just from the backfilling strategy, but from the scheduling problem itself. The tasks have a two-dimensional representation based on (q_j, p_j) and it is reasonable to expect that the difficulty of finding a good schedule should be dependent not only on the prediction error, but also on how it is distributed among jobs of different q and p. This suggests that the loss function should be weighted differently according to the jobs considered, leading to:

$$\mathfrak{L}(\mathbf{x}_j, f(\mathbf{x}_j), p_j) = \begin{cases} \gamma_j . L_u(f(\mathbf{x}_j) - p_j) & \text{if } f(\mathbf{x}_j) \geq p_j \\ \gamma_j . L_o(p_j - f(\mathbf{x}_j)) & \text{if } f(\mathbf{x}_j) < p_j \end{cases}$$

where L_u is the underprediction basis loss function, L_o is the overprediction basis loss function, and $\gamma_j > 0$ corresponds to the weight of job j.

[3]Note that one can also consider the k latest jobs or weigh differently the jobs to favor recent ones. These variants are straightforward to consider from the framework developed here.

Figure **??** shows an example of an asymmetrical loss function with a unit weight γ_j, using a squared loss basis for underprediction and a linear loss basis for overprediction.

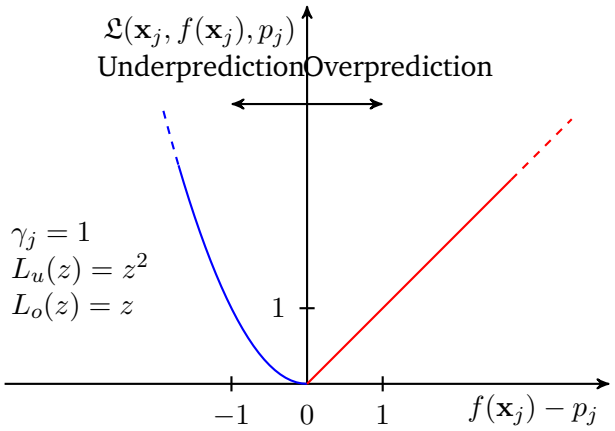


Fig. 4.1: Example Loss function $\mathfrak{L}$, plotted with respect to the difference of it's second and third parameters $f(\mathbf{x}_j) - p_j$ (the prediction error).

In this study, we consider two standard loss functions for under- and over-prediction, namely the squared loss ($L(z) = z^2$) and the linear loss ($L(z) = z$). It can be verified that all the possible combinations of these two loss functions in $\mathfrak{L}$ are continuous and convex (even though not differentiable everywhere) with respect to vector $\mathbf{w}$.

Choosing the weighting factor γ_j is not straightforward. On the one hand, a key property of backfilling algorithms is that jobs of small area (small p, small q) are easier to backfill. Underpredicting small jobs can therefore mean delaying a reservation, and one should therefore use a weighting factor which *decreases* with p and q. On the other hand, as we consider systems with no preemption, once a large (large p, large $q \approx m$) job is started, almost no CPUs/Nodes remain available. Thus, jobs in the queue are doomed to wait a long time. It follows that predicting jobs of large area correctly should be beneficial, and one should therefore use a weighting factor which *increases* with p and q. To account for these various elements, we explore four different possibilities along with a constant weighting factor, all of which are shown in Table **??**.

Finally, for each choice of two basis loss function L_u and L_o along with weighting scheme γ_j, the regression model (*i.e.* the vector $\mathbf{w}$) is learned on an on-line training/testing set by minimizing the cumulative loss through the Normalized Adaptive Gradient [**nag**] algorithm (NAG), a variant of the classical Stochastic Gradient Descent [**bottou**]. The NAG algorithm poses the advantage of being robust

to adversarial[4] scaling of the features. Robustness to feature scaling is a requirement of our problem because some features are difficult or impossible to normalize (*e.g.* BREAK TIME is unbounded). Section **??** describes the data sets retained as well as the values of the parameters considered for the search. Note that when $\gamma_j = 1$ and $L_u(z) = L_o(z) = z^2$, one is just considering a standard squared loss regression problem, learned in an on-line manner.

4.3 Scheduling

4.3.1 The EASY algorithm

EASY is one of the most on-line version of backfilling algorithms. Backfilling algorithms are based on a priority queue where the jobs with the highest priorities are launched first. A job with a lower priority can run before a higher priority job if and only if it does not delay it. EASY is considered as an on-line algorithm since the decision to launch a job is only taken at the last moment. For instance as it is depicted in Figure **??**, the decision to launch job 2 is taken after the completion of job 1. Job 3 has been backfilled. In this figure, we can measure the importance of running times in backfilling algorithms. If the estimated running time of job 1 was much shorter, job 3 would not be backfilled.

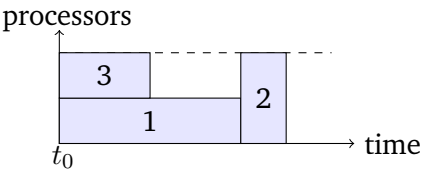


Fig. 4.2: Example of EASY for a queue of 3 jobs ordered according to their index. The decision to launch job 1 is taken at t_0. The second ready job (2) can not be executed since there are not enough CPUs/Nodes left. Thus, job 3 can also be launched at t_0. The decision to launch job 2 is finally taken when job 1 and 3 complete.

Tsafrir *et al.* proposed in [**tsafrir**] a slightly modified version of EASY, called EASY-SJBF, which performed better with running time predictions than the standard version. During the phase when the algorithm determines the candidate jobs to be backfilled, the jobs are sorted by increasing predicted running times instead of considering the FCFS order. They argue that this way, more jobs will be backfilled and thus, the overall performances will increase.

[4]The notion of *adversary* simply means that the scaling can be arbitrary. See [**plg**] for a comprehensive study.

4.3.2 Correction mechanism

In this section, we are interested in the following question: *what happens to the schedule when the running times are mispredicted?*

The case of over-predicted running times is easy to handle by backfilling since the situation is the same as without learning where the users over-estimate the requested running times. In case of under-predicted running times, there are two points to consider. First, we should determine a new prediction for the remaining execution time and second, we have to check whether the correction does not disturb too much the scheduling algorithm. Both points are detailed as follows.

First point. We prefer to update the running times by some simple rules instead of computing again a prediction by the learning scheme, which gave a wrong value. Obviously, these updated running times remain bounded by the requested running times. These values are given by a correction algorithm. We consider the three following ones:

- REQUESTED TIME – Set the new prediction value to be $\widehat{p_j}$ (the user requested running time);
- INCREMENTAL – Use the corrective technique from [**tsafrir**], *i.e.* increase at each correction by an fixed amount of time (1min, 5min, 15min, 30min, 1h, 2h, 5h, 10h, 20h, 50h, 100h);
- RECURSIVE DOUBLING – Increase the prediction value by the double of the elapsed running time.

Second point. Do backfilling variants handle the updated prediction? As the considered backfilling algorithms are on-line in nature, they adapt dynamically to the changes. However, notice that under-prediction with backfilling can lead to starvation. For instance, a large job will indefinitely wait for its required CPUs/Nodes if under-predicted shorter jobs are systematically backfilled before. They will be launched before the large one, leading to an unbounded delay.

4.3.3 Objective Functions

As it is commonly admitted [**FeitelsonBSLD**], the performances of scheduling algorithms are measured using the *bounded slowdown*, which is defined as follows for job j:

$$\text{bsld} = \max\Big(\frac{wait_j + p_j}{\max(p_j, \tau)}, 1\Big)$$

where $wait_j$ is the waiting time of job j (from the time it is released in the system and the time it starts its execution) and τ is a constant preventing small jobs to reach too large slowdown values. In the literature, τ is generally set to 10 seconds. We will use this value in the experiments.

One related objective function usually used for comparing performances is the average of bsld, defined as:

$$\text{AVEbsld} = \frac{1}{n}\sum_j \max\Big(\frac{wait_j + p_j}{\max(p_j, \tau)}, 1\Big)$$

In this chapter, all scheduling performances are measured with this objective function.

4.4 Experiments

4.4.1 Experiment objectives

The simulations we conducted aim at answering the following two questions:

1. Do the proposed predictive and corrective techniques improve existing scheduling algorithms?

2. Which prediction loss function, correction mechanism and backfilling variant work well together?

Previous studies have mainly focused on predicting running times, independently of the scheduling algorithms, using standard measures for the prediction error. In contrast, we aim here at predicting running times *for scheduling jobs with backfilling*, through a combination of appropriate loss functions, correction mechanisms and backfilling variants. The solutions we develop are thus closer to the problem of improving HPC systems. Moreover, identifying efficient combinations of prediction technique, correction mechanism and backfilling variants should provide insights

into the behavior of backfilling algorithms when running times are unsure. In the rest of the paper, we refer to such combinations as *heuristic triples*.

4.4.2 Description of the testbed

We make use in our study of a set of actual workload logs, described in Table **??**. All these workload logs but Metacentrum are extracted from the Parallel Workload Archive [**workloadarchive**]. Metacentrum is extracted from the personal website of Dalibor Klusáček[5]. They come from various HPC centers, correspond to highly different environments and have been selected for their high resource utilization, which challenges scheduling algorithms [**FeitelsonPitfalls**]. For each log, we run scheduling simulations using every possible heuristic triples: prediction technique, correction mechanism and backfilling variant.

All simulations are run using a fork of the open-source[6] batch scheduler simulator *pyss*.

All prediction techniques based on the different loss functions and weighting schemes introduced in Section **??** are considered here, in conjunction, for comparison purposes, with the actual running time p_j, denoted as CLAIRVOYANT, the user requested time $\widetilde{p_j}$, denoted as REQUESTED TIME and the average of the two previous running times for the jobs of user k, denoted as $AVE_2^{(k)}(p)$. For correction, we rely on the three correction techniques presented in Subsection **??**: REQUESTED TIME, INCREMENTAL and RECURSIVE DOUBLING. Lastly, we rely on the two backfilling algorithms presented in Subsection **??** , namely EASY and EASY-SJBF.

Notice that the case where REQUESTED TIME is used as prediction technique and EASY as the backfilling variant corresponds to the standard EASY backfilling algorithm. Similarly, the case where INCREMENTAL correction method is used with the $AVE_2^{(k)}(p)$ prediction technique and the EASY-SJBF Backfilling variant correspond to the EASY++ algorithm introduced by Tsafrir *et al.* [**tsafrir**].

As it is reasonable to expect that scheduling performances due to a loss function (and therefore, a learned model) are dependent on both the backfilling variant and correction mechanism, we evaluate all of them together. This induces a higher complexity and a high number of simulations. For each workload log, the experimental campaign runs 128 simulations.

[5] http://www.fi.muni.cz/∼xklusac/
[6] pyss - the Python Scheduler Simulator, available at http://code.google.com/p/pyss/

The experiment campaign contains significantly more heuristic triples than workload logs, and this raises a multiple hypothesis testing problem. Therefore, one should approach the analysis of the results using sound statistical practices.

Subsection **??** outlines the best heuristic triple. Afterwards, Subsection **??** contains an analysis of the predictions made.

4.4.3 Which heuristic triple is prevalent?

Raw results

As shown in Table **??** which displays the AVEbsld scores for the different approaches, the results seem promising as they tend to favor approaches based on learning (the CLAIRVOYANT results are reported for comparison purposes only and correspond to an upper bound of what one can expect). However, one should not conclude too hastily, as even though the best approach is always obtained using a predictive-corrective approach (corresponding to the columns EASY with Learning Techniques in Table **??**), it is not clear which heuristic triple is prevalent *a priori*.

In particular we observe that the SJBF variant introduced by [**tsafrir**] is rather efficient at leveraging accurate values of the running times, as the CLAIRVOYANT EASY-SJBF algorithm almost always outperforms its competitors.

Correlation between logs

A key part of the analysis of predictive techniques is to see how performances correlate between different systems.

Figure **??** shows the AVEbsld between the MetaCentrum and SDSC-BLUE logs, meant to illustrate the irregularity of performances between logs. We can observe that CLAIRVOYANT EASY-SJBF is the best in both logs, but there is no clear correlation between all the heuristic triples.

The correlation coefficient is measured here with the Pearson's Correlation coefficient, which is computed for each couple of logs. With a mean of 0.26 (min: 0.01, max: 0.80), this coefficient is low. This means that it is not possible to know from the result on one log if a heuristic triple will perform well on another logs. However, one can still try and learn an appropriate heuristic triple from existing systems, as described below.

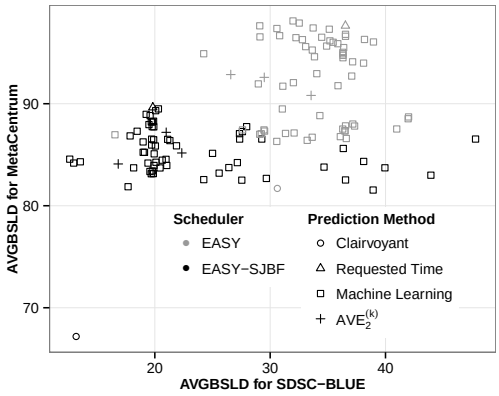


Fig. 4.3: Scatter plot of heuristic's relative performance between the MetaCentrum and SDSC-BLUE logs.

Triple selection

We consider here a *leave-one-out cross validation* process in which 5 logs are (alternatively) used to select the best triple, the performance of which is evaluated on the 6th log. The idea is to assess whether one can select a good heuristic triple from existing logs. The experiment is repeated six times (for the six logs) and the results are averaged over the six repetitions. The best heuristic triple is the one that optimizes the sum of the AVEbsld on the 5 logs. Table **??** displays the obtained results. As one can note, the results obtained with this selection process, denoted C-V (for cross-validation) heuristic triple, significantly outperforms the EASY and EASY++ approaches on all workloads except SDSC-BLUE.

Even more interestingly, it turns out that the best heuristic triple on all logs using the selection method above is the same[7] and corresponds to the following setting:

Prediction Technique : Regression function described in Section **??** with the loss function:

$$\mathfrak{L}(\mathbf{x}_j, f(\mathbf{x}_j), p_j) = \begin{cases} \log(r_j.p_j).(f(\mathbf{x}_j) - p_j)^2 & \text{if } f(\mathbf{x}_j) \geq p_j \\ \log(r_j.p_j).(p_j - f(\mathbf{x}_j)) & \text{if } f(\mathbf{x}_j) < p_j \end{cases} \tag{4.3}$$

Correction mechanism : INCREMENTAL

backfilling variant : EASY-SJBF

[7] with one exception: the C-V Heuristic selected for SDSC-SP2 uses the REQUESTED TIME correction mechanism. This could account for it's degraded performance.

Summary

We have shown here that one can learn an appropriate heuristic triple from existing logs. This heuristic triple yields better scheduling performances than EASY and EASY++. Furthermore, on the workloads considered, a heuristic triple singles out as it is the one always selected. This heuristic triple obtains an average AVEbsld reduction of 28% compared to EASY and 11% compared to EASY++, and can reduce the AVEbsld by 86% compared to EASY (on the Curie workload for example). This triple uses the INCREMENTAL correction technique and the SJBF queue ordering from [**tsafrir**], as well as a machine learning-based approach with custom loss functions (**??**). We call this loss function E-Loss (for EASY-Loss) and briefly discuss its behavior in the next Subsection.

4.4.4 Prediction analysis

The first question one usually asks after having used a predictive technique is, *what is the prediction accuracy*? Experimental results outline that while prediction performance is important, choosing the right loss for the prediction is even more critical. This is observed in Table **??** which shows the prediction errors of both the $AVE_2^{(k)}(p)$ prediction technique and our E-Loss based approach.

One can see from these values that while the $AVE_2^{(k)}(p)$ performs well with respect to the Mean Average Error (MAE), its performance on the E-Loss is quite poor.

Equation (**??**) shows that the E-Loss is an asymmetrical loss function, with a linear branch for under-prediction and a squared branch for over-prediction. Therefore this loss function discourages over-prediction. Additionally, the E-Loss uses a weighting factor that increases with the size of jobs in terms of p and q. A helpful visualization for understanding how the E-loss behaves in practice is the empirical cumulative distribution function (ECDF) of the prediction errors produced by the resulting machine learning model. Figure **??** shows the ECDFs of such prediction errors for main prediction techniques. From this figure, one can see the behavior of the E-loss with respect to that of the standard squared loss. The E-loss ECDF is shifted to the left, which means that more under-prediction errors are indeed made than with standard regression. This is coherent with intuition gleaned from the analysis form of the loss function.

Finally, Figure **??** shows the ECDF of the values that were predicted. On this graph, we see that in order to generalize well with respect to the E-Loss, the learning model ends up being strongly biased towards small predictions. This displacement suggests

that there might be a beneficial effect to backfilling jobs very aggressively when using EASY-SJBF.

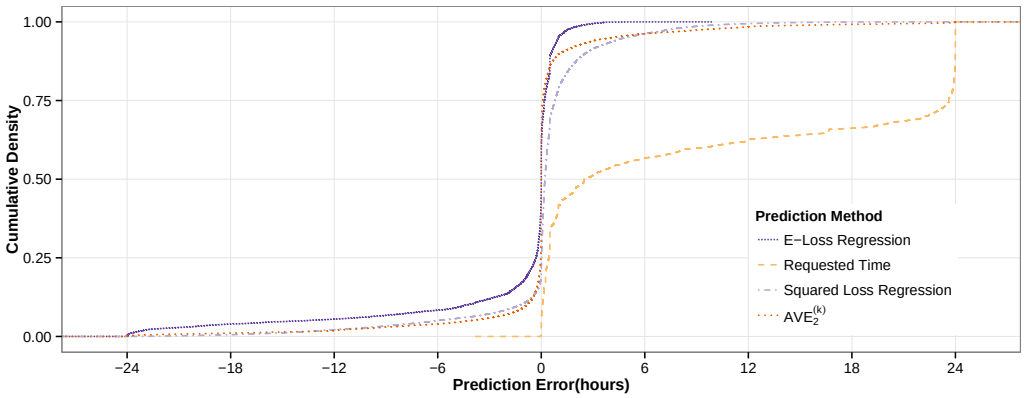


Fig. 4.4: Experimental cumulative distribution functions of prediction errors obtained using the Curie log.

4.4.5 Discussion

As mentioned above, the proposed approach outperforms EASY++ with a **11%** of reduction in average AVEbsld, and has a reduction of **28%** in average AVEbsld when compared to EASY. This result is obtained by changing the prediction technique of EASY++ to one that uses a custom loss function that we refer to as E-loss. We have furthermore observed that on each log, roughly 0.1% of jobs have extremely high values of bounded slowdowns. Such a behavior is obtained with every heuristic triple based on $AVE_2^{(k)}(p)$ or MACHINE LEARNING prediction techniques. Extreme values seem to be a shortcoming of incorporating predictions without a mechanism for dealing with extreme prediction failures. Moreover, because such failures are often due to jobs that do not run properly, we are confronted here with an evaluation problem, as the cost of such events could be incurred on the user rather than

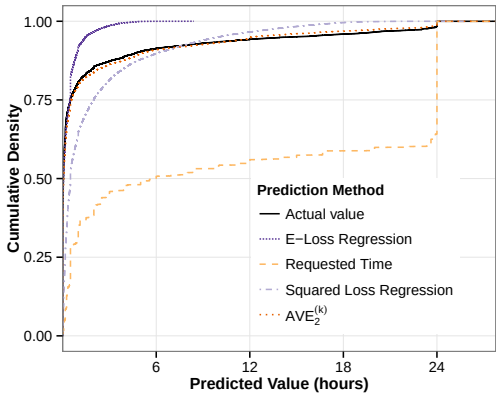


Fig. 4.5: Experimental cumulative distribution functions of predicted values obtained using the Curie log.

the system. New performance evaluation measures are needed to deal with such problems, especially for schedulers without no-starvation guarantees (as other authors already suggested [**FeitelsonPitfalls**]).

4.5 Conclusion

The purpose of this chapter was to investigate whether the use of learning techniques on the job running times is worth for improving the scheduling algorithms. We proposed a new cost function for prediction and run simulations based on actual workload logs for the most popular variants of backfilling. The results clearly show that this approach is useful, as they reduce the average bounded slowdown by an average factor of 28% compared to EASY.

There are however, several cons to applying this technique in practice. First of all, many simulations have to be performed in order to select hyper-parameters. Because the simulation model used here is simplistic, the hyper-parameter values should not be directly generalized to real systems. This means that simulations with more precise simulators are necessary. This is costly both in terms of time and computation. The two other drawbacks we can see are the poor performance on some traces (CTC-SP2), compared to what could have been achieved using a different approach, as Table **??** shows. Finally, extreme values of the objective metric are not controlled at all by the approach.

In the next two chapters, we solve these problems by changing both the search space of our method and the learning approach. We'll use thresholds to control large values, add more diversity by searching directly in policy space, and reduce the number of hyper-parameters to one by using a multi-armed bandit algorithm.

Tab. 4.2: Features extracted from the *SWF* data, for job j, belonging to user k.

Feature	Meaning
$\widetilde{p_j}$	the time the user requested for her job.
$p_{j-1}^{(k)}$	the running time of the last job of the same user, or 0 if such a job does not exist.
$p_{j-2}^{(k)}$	the running time of the second-to-last job of the same user, or 0 if N/A.
$p_{j-3}^{(k)}$	the running time of the third-to-last job of the same user, or 0 if N/A.
$AVE_2^{(k)}(p)$	the average running time of the two last historically recorded jobs of the same user.
$AVE_3^{(k)}(p)$	the average running time of the three last historically recorded jobs of the same user.
$AVE_{all}^{(k)}(p)$	the average running time of all historically recorded jobs of the same user.
q_j	amount of (CPU) resource requested by job j.
$AVE_{hist,r_j}^{(k)}(q)$	average historical resource request of user k, taken at release date of job j.
$\frac{q_j}{AVE_{hist,r_j}^{(k)}(q)}$	amount of resource requested normalized by average resource request.
$AVE_{curr,r_j}^{(k)}(q)$	average resource request of the user's currently running jobs, at release date
JOBS CURRENTLY RUNNING	number of jobs of the user running, at release date
LONGEST CURRENT RUNNING TIME	longest running time (so-far) of the user's currently running jobs, at release date
SUM CURRENT RUNNING TIMES	sum of the running times (so-far) of the user's currently running jobs, at release date
OCCUPIED RESOURCES	total CPUs currently being allocated to the same user.
BREAK TIME	time elapsed since last job completion from the same user.
$\begin{cases} cos(\frac{2*\pi}{t_{day}} * (r_j \pmod t)_{day})) \\ sin(\frac{2*\pi}{t_{day}} * (r_j \pmod t)_{day})) \end{cases}$	time of the day the job was released. The periodic feature is decomposed into its cosinus and sinus, using the day period t_{day} (length of a day in seconds)
$\begin{cases} cos(\frac{2*\pi}{t_{week}} * (r_j \pmod t)_{week})) \\ sin(\frac{2*\pi}{t_{week}} * (r_j \pmod t)_{week})) \end{cases}$	time of the week the job was released. The periodic feature is decomposed into its cosinus and sinus, using the week period t_{week} (length of a day in seconds)

Tab. 4.3: Weighting factors considered for training the model. The constants are chosen to ensure positivity of the weights with typical running times and resource requests in the HPC domain. Logarithms are used to alleviate the high range produced by ratios.

γ_j	Interpretation
1	Constant weight.
$5 + \log(\frac{q_j}{p_j})$	Short jobs with large resource request should be well-predicted.
$5 + \log(\frac{p_j}{q_j})$	Long jobs with small resource request should be well-predicted.
$11 + \log(\frac{1}{q_j.p_j})$	Jobs of small "area" should be well-predicted.
$\log(q_j.p_j)$	Jobs of large "area" should be well-predicted.

Tab. 4.4: 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
Curie	2012	80,640	312k	3 Months
Metacentrum	2013	3,356	495k	6 Months

Tab. 4.5: Considered parameter values of the loss function. There are three effective parameters, for a total of 20 combinations.

Parameter	Possible Values
L_u	$z \mapsto z^2$, $z \mapsto z$
L_o	$z \mapsto z^2$, $z \mapsto z$
γ_j	See Table **??** (5 values)

Tab. 4.6: Overview of the AVEbsld for each simulations. For predictive techniques, only the best and the worst AVEbsld are given. The best non-clairvoyant heuristic triples are outlined in bold.

	Clairvoyant EASY				EASY with Learning Techniques	
Trace	FCFS	SJBF	EASY	EASY++	FCFS	SJBF
KTH-SP2	71.7	49.8	92.6	63.5	62.6 - 93.2	**51.4** - 74.5
CTC-SP2	37.2	17.6	49.6	85.8	25.5 - 163.5	**16.3** - 134.7
SDSC-SP2	70.5	56.8	87.9	79.4	70.9 - 102.3	**69.7** - 194.8
SDSC-BLUE	30.6	13.2	36.5	21.0	16.5 - 48.0	**12.6** - 47.8
Curie	69.9	12.1	202.1	193.5	26.3 - 9348.8	**24.3** - 4010
Metacentrum	81.7	67.2	97.6	87.2	86.3 - 98.1	**81.5** - 89.8

Tab. 4.7: AVEbsld performance of the heuristic triples resulting from cross validation. Values in parenthesis show the AVEbsld reduction obtained respective to EASY.

Log	C-V Heuristic triple	EASY	EASY++
KTH-SP2	**51.4** (44%)	92.6	63.5 (31%)
CTC-SP2	**20.5** (59%)	49.6	85.8 (-72%)
SDSC-SP2	**75.0** (15%)	87.9	79.4 (10%)
SDSC-BLUE	34.7 (05%)	36.5	**21.0** (42%)
Curie	**27.9** (86%)	202.1	193.5 (04%)
Metacentrum	**84.2** (14%)	97.6	87.2 (11%)

Tab. 4.8: MAE and E-Loss for different prediction techniques. All values are in seconds.

Prediction Technique	MAE	Mean E-Loss
$AVE_2^{(k)}(p)$	5217	10.2×10^8
E-Loss Learning	6762	2.35×10^5

5 A Queue Tuning Framework.

In this chapter, we take a step back from the previous approach. We consider in this chapter 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.

5.1 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 [**DOEASCAC**]. 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 [**easy**, **lifka**]. 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 chapter 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 chapter is organized as follows: Section **??** describes an experimental setup that is essential to the discussion. Section **??** introduces our approach, illustrating the discussion with results from the KTH-SP2 trace. Section **??** describes the thresholding mechanism used. Section **??** validates this approach using a comprehensive experimental campaign on 5 logs from the Parallel Workload Archive [**Feitelson20142967**].

5.2 Problem Setting

5.2.1 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 [**easy**]. 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 **??** describes the EASY-P_R-P_B heuristic.

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

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 [**easy**].
- LCFS: Last-Come First-Serve.
- LPF: Longest estimated Processing time $\widetilde{p_j}$ First.
- SPF: Smallest estimated Processing time $\widetilde{p_j}$ First [**bfchar**].
- LQF: Largest resource requirement q_j First.
- SQF: Smallest resource requirement q_j First.

- EXP: Largest Expansion Factor First [**bfchar**], where the expansion factor is defined as follows:

$$\frac{wait_j + \widetilde{p_j}}{\widetilde{p_j}} \tag{5.1}$$

  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.

5.2.2 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.

$$\textbf{Wait}_j = start_j - r_j \tag{5.2}$$

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.

5.2.3 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.

Tab. 5.1: AvgWait performance of EASY-EXP-EXP and EASY-SQF-SQF on the orig- inal 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 [**variability**, **bfchar**]. Table **??** 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.

Tab. 5.2: 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 [**Tsafrir_easypp_2005**, **learningruntimes**] that optimize the average cost by removing the FCFS constraint. Table **??** 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 chapter, 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.

5.3 Experimental Protocol

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

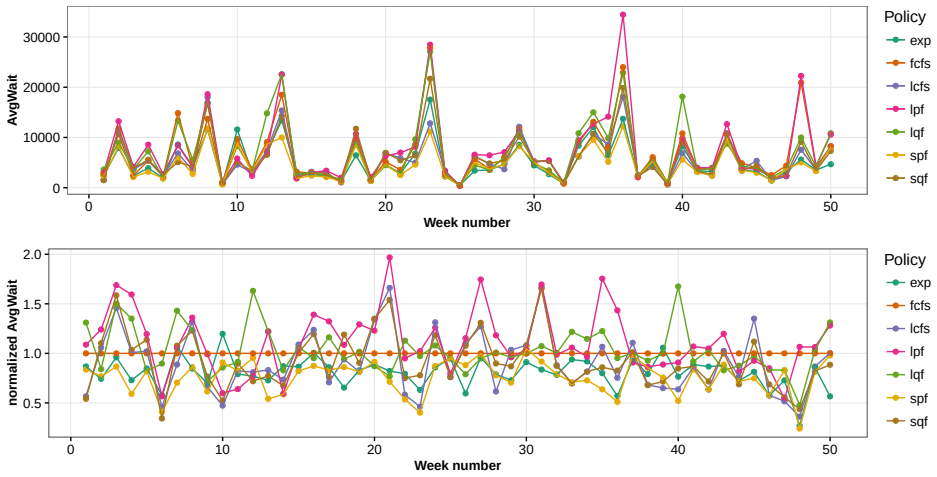


Fig. 5.1: AvgWait obtained for the 7 main queue policies with **FCFS** backfilling for 150 generated weeks on the KTH-SP2 trace. First, in absolute value, and then normalized with respect to EASY-FCFS-FCFS.

5.3.1 Statistical approach

The experimental approach used in this chapter is statistical by nature. Figure **??** 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 which has the first 14 jobs removed. The variability [**variability**, **frachtenberg2005pitfalls**] of cost metrics and their sensitivity to small changes in the workload logs [**flurries**] have been thoroughly studied in the literature. Our approach to measuring performance without reporting noise from workload flurries [**flurries**] 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 [**feitresampling**]: 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 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.

[1] See the Parallel Workloads Archive [**Feitelson20142967**] for details.

5.3.2 Simulation method and testbed

While high quality simulators like SimGrid [**casanova:hal-01017319**] are available in practice, this chapter 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 chapter 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 [**stodden2014implementing**] and provide a snapshot of the work that includes a build system that runs the experiments using the *zymake* [**breck2008zymake**] minimalistic workflow system. The archive includes our simulator and a nix [**dolstra2004imposing**] file that describes the dependencies. It can be obtained from:

$$https://zenodo.org/record/290428$$

5.4 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.

5.4.1 Maximum and Average Cost

Figure **??** and **??** 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 **??** and **??** vary in terms of y axis. In Figure **??**, the y axis is the maximum MaxWait over simulated week, i.e. the highest waiting time of any job on all the simulated weeks.

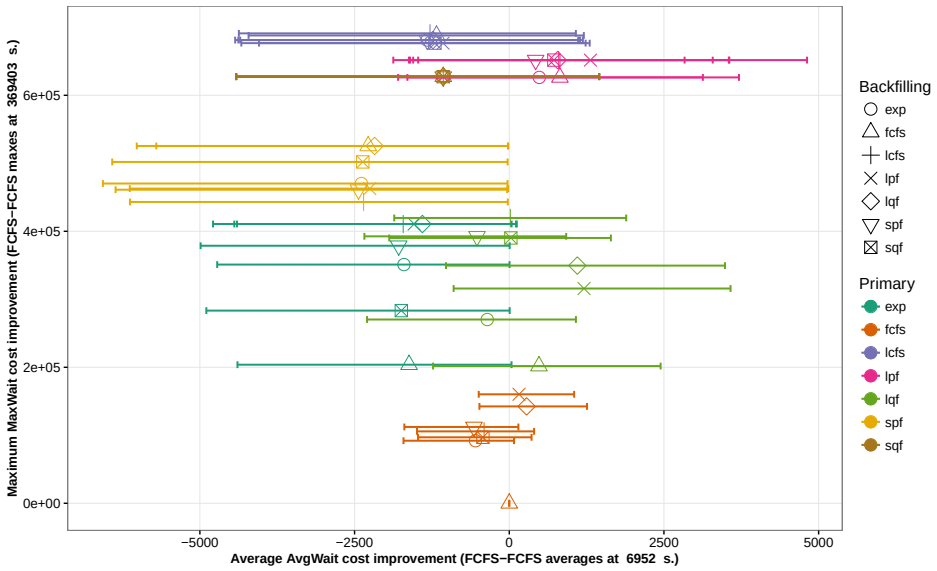


Fig. 5.2: Maximum and average waiting time cost of the 49 heuristics generated by using the 7 possible policies as primary and backfilling ordering averaged over 250 resampled weeks. All values are relative to the value obtained by the **EASY** primary queue policy with **EASY** backfilling. The maximum 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.

In Figure **??**, the y axis is the average MaxWait over the 250 weeks. The average value reported is the mean average cost over individual weeks, which allows for displaying deciles in both directions. Note that Figure **??** 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.

5.4.2 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 **??** shows, the

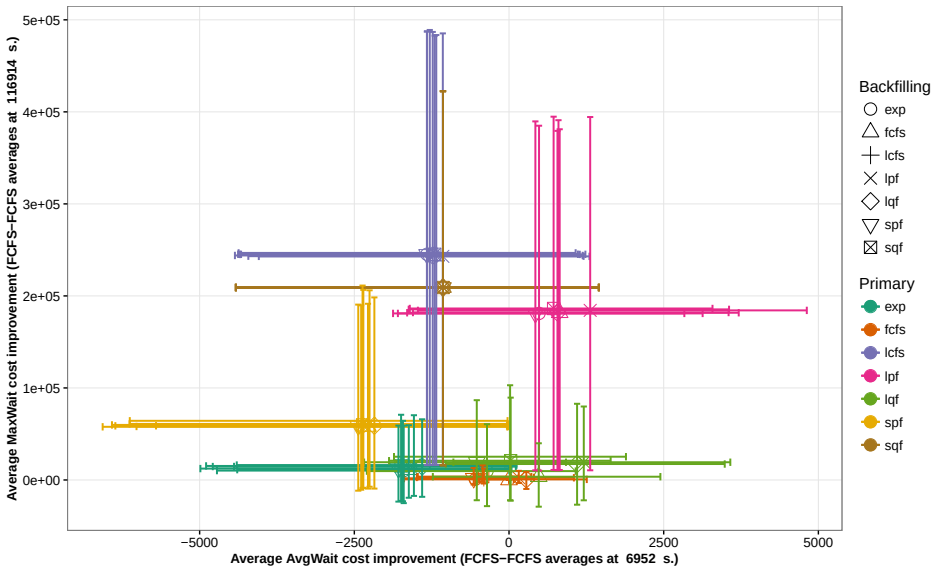


Fig. 5.3: Maximum and average waiting time cost of the 49 heuristics generated by using the 7 possible policies as primary and backfilling ordering averaged over 250 resampled weeks. All values are relative to the value obtained by the **EASY** primary queue policy with **EASY** backfilling. The average MaxWait value reported is the average of the maximum waiting time over 250 weeks. The average AvgWait value is as in Figure **??** 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.

AvgWait performance of all backfilling policies relative to EASY-FCFS-FCFS presents roughly the same relative performance for each primary 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.

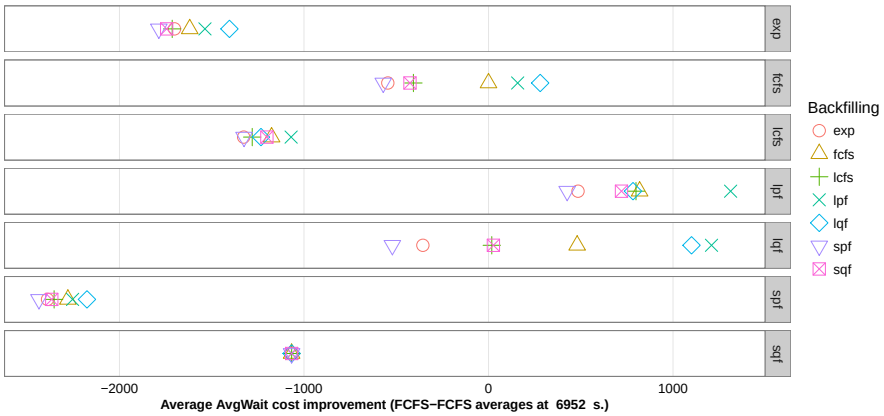


Fig. 5.4: Performance improvement over EASY-FCFS-FCFS of the 7 Backfilling policies conditioned on Primary policy.

5.5 Queue threshold

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

5.5.1 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 $t - r_j$ 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 **??**). If more than one job is in this situation, these jobs are ordered by submission time r_j at the head of the queue.

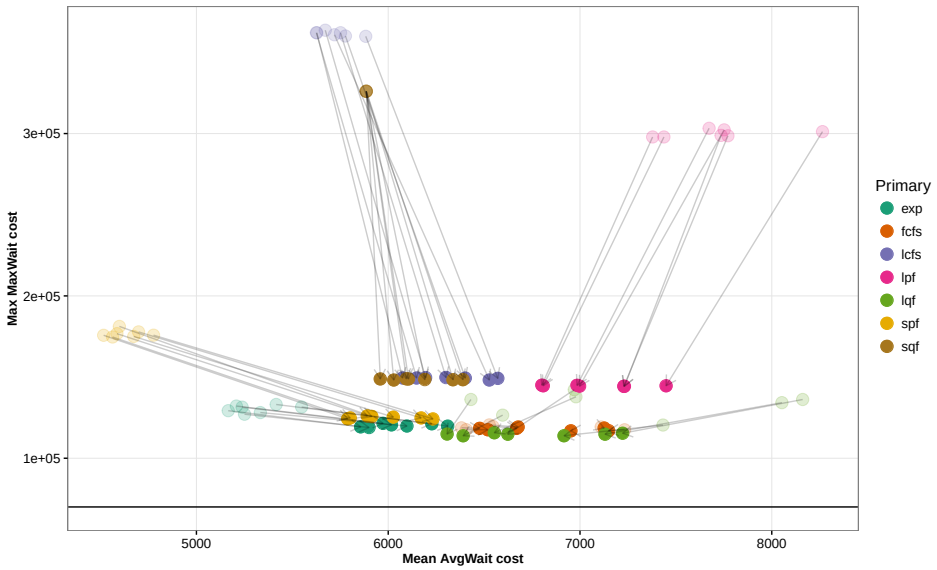


Fig. 5.5: Maximum and average waiting time cost of the 7 heuristics generated by using the 7 possible thresholded primary policies with SPF backfilling averaged over 250 resampled weeks. The threshold T is chosen at a value of 20 hours. All values are relative to the value obtained by the **EASY** primary 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 **??** the mean of the weekly waiting time averages. Semi-transparent points represent the performance of the un-thresholded policies.

Figure **??** 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 **??**) for pure visual reasons and exhaustive treatment

[2] Note that this is also valid for the more refined Average Bounded Slowdown [**feitelson1998metrics**] metric.

is delayed to Section **??**. 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.

5.6 Experimental Validation

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

5.6.1 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 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.

5.6.2 Workload logs

Table **??** outlines the five workload logs from the Parallel Workloads Archive [**Feitelson20142967**] 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.

Tab. 5.3: 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

5.6.3 Empirical generalization results

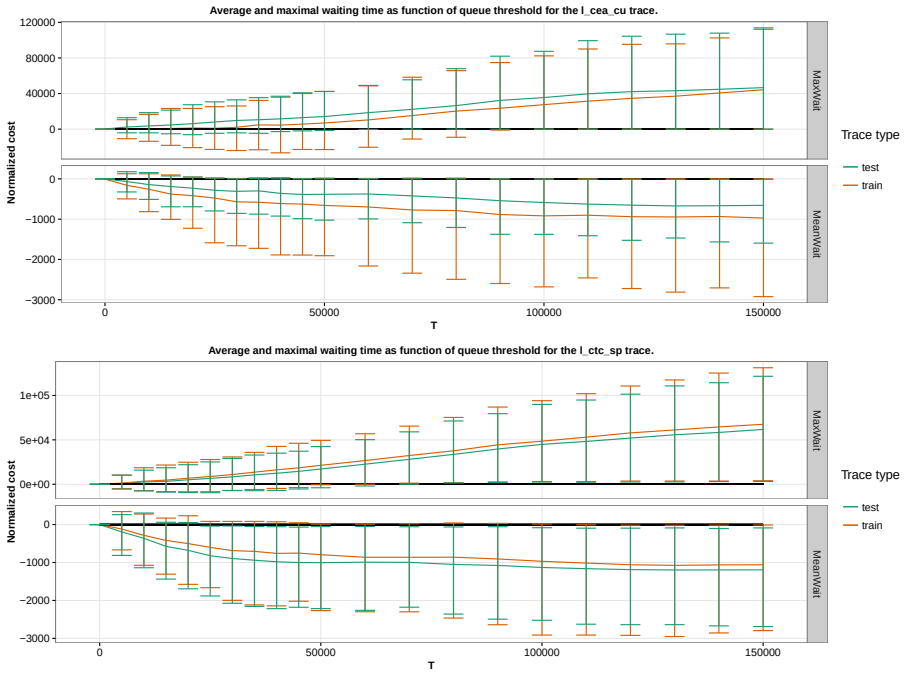


Fig. 5.6: AvgWait and MaxWait generalization of thresholded policies as affected by the queue threshold. The Value reported as "train" is that of the least costly heuristic among the 49 possible policy parametrizations averaged on the *training* logs. The Value reported as "test" is the averaged cost of the same heuristic on the *testing* logs. This figure is continued as Figure **??**.

Figure **??** 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.

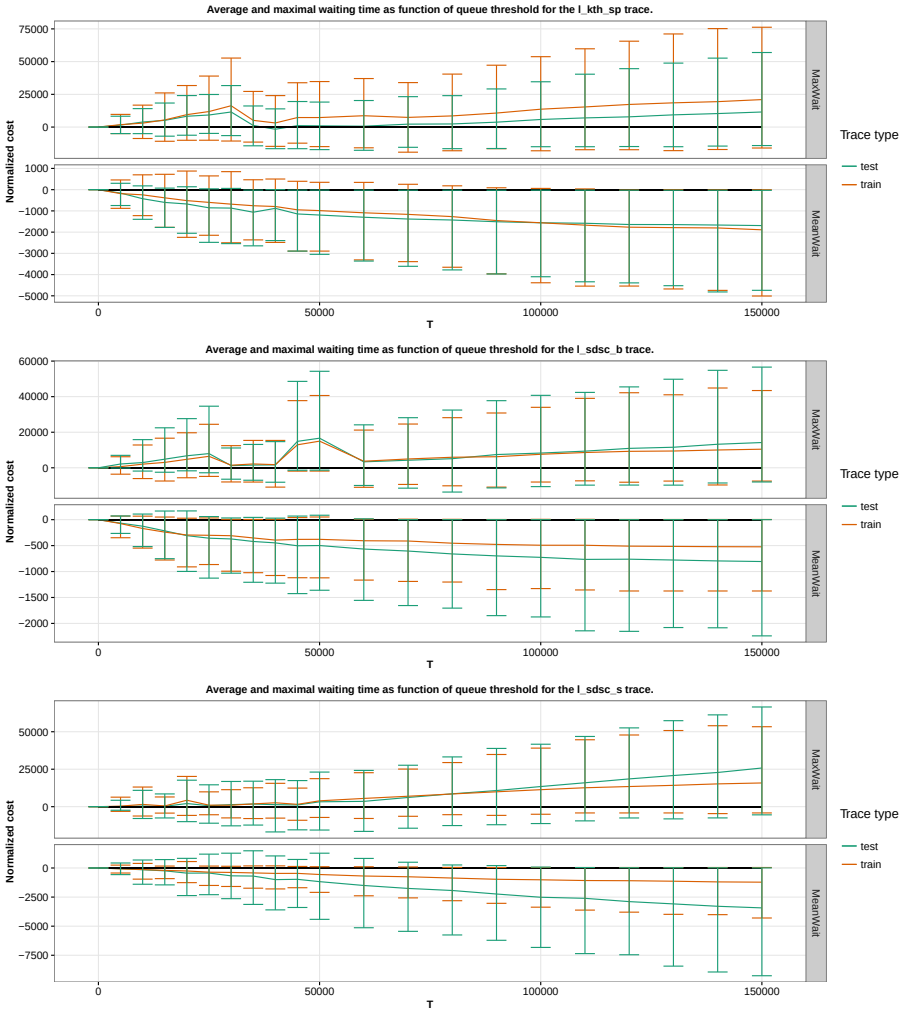


Fig. 5.7: Follow-up from Figure **??**.

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 **??** 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 permitted 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 chapter are also valid for the more refined Average Bounded Slowdown [**feitelson1998metrics**].

5.6.4 Generalization with T=20h

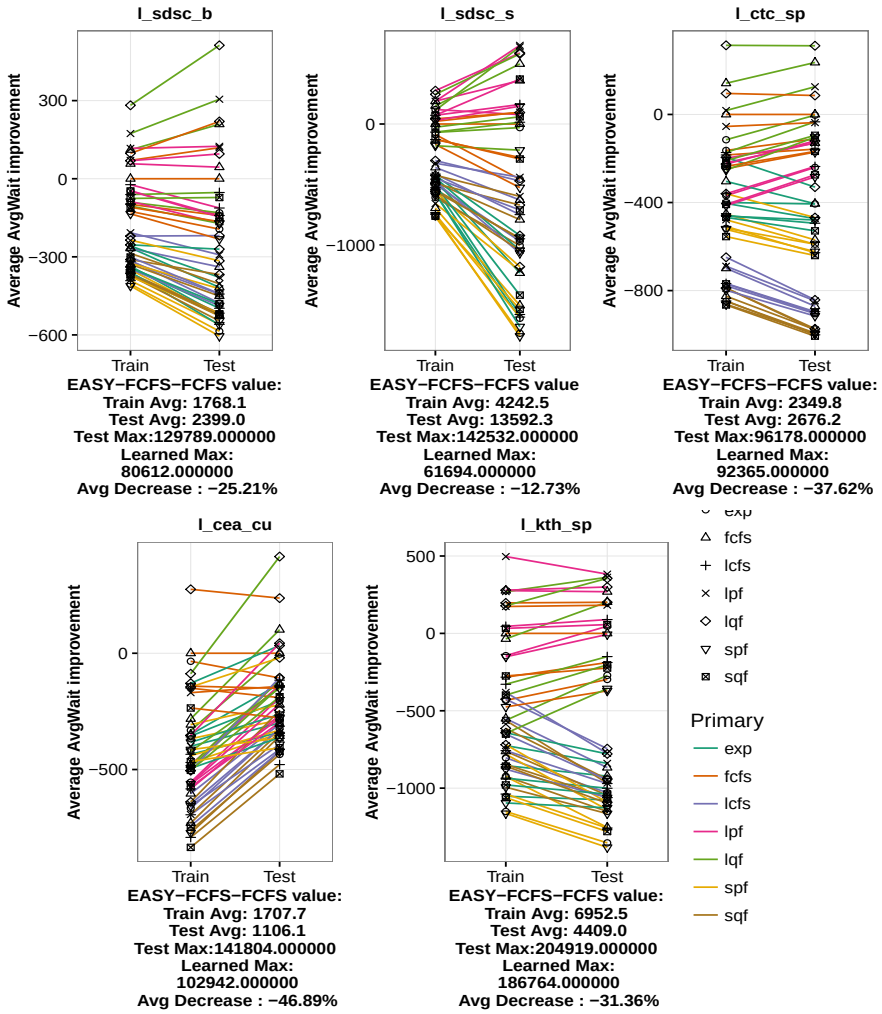


Fig. 5.8: AvgWait generalization of thresholded policies obtained by using a threshold value of 20 hours. Note that each plot has a different vertical y axis. The reported AvgWait and MaxWait values are averaged over 250 resampled weeks from the training or testing original logs, and we report the difference 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).

The final step is to study how the performance thresholded queue policies generalizes. Figure **??** 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.

5.7 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 chapter.

First, the stability of other EASY heuristic classes remains unknown. We showed empirically that the "simple" class of composed of 7 (thresholded) primary policies and 7 backfilling policies (cardinality 49) is an useful search space. Indeed, a discrete a-posteriori risk minimization choice in this search space does generalize. 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 criteria. A more ambitious endeavor is to ask whether it is possible to learn a contextual job ranking model [**joachims2002optimizing**] that performs well.

6 Online Multi-Armed Bandit Tuning.

In this chapter, we leverage the queue tuning framework that was explored in the previous chapter. We consider an online approach to the automatic tuning of the EASY heuristic for HPC platforms. More precisely, we consider the problem of selecting a reordering policy for the job queue under several feedback modes. We show via a comprehensive experimental validation on actual logs that periodic simulation of historical data can be used to recover existing *in-hindsight* results that allow to divide the average waiting time by almost 2. This results holds even when the simulator results are noisy. Moreover, we show that good performances can still be obtained without a simulator, under what is called bandit feedback - when we can only observe the performance of the algorithm that was picked on the live system. Indeed, a simple multi-armed bandit algorithm can reduce the average waiting time by 40%.

6.1 Introduction

In this chapter, we provide algorithms that use feedback from the system in order to alter the scheduling behavior online. Our focus is to learn how the platform behaves within the usual framework of EASY-Backfilling. More precisely, we provide strategies for choosing the most appropriate queue reordering policy to be used for job selection. This approach is assessed by an extensive resampling-based experimental validation that uses 7 actual traces.

6.1.1 Contributions

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

The first method, **full feedback**, continuously uses simulation on data gathered from the past and current system workload. In order to reflect the uncertainties in

the data, we also evaluate a more realistic "noisy" variant in which simulations are imprecise.

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

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

Our main results are as follows: The approach based on the simulator provides very good results, close to the best policy in the search set. In essence, we are able to recover in an online fashion existing results that reduce average waiting times by 11% to 60% compared to EASY-FCFS. Indeed, we show that the required sample sizes for policy selection in batch schedulers are small enough to avoid workload resampling in practice. The variant based on noisy data, which is closer to the actual conditions of uncertainties on both jobs and platform, behaves similarly. The alternate approach based on a multi-armed bandit also provides reasonable results at a much lower cost, reducing the average waiting times by 8% to 46% compared to EASY-FCFS.

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

6.2 Tuning EASY by Reordering and Thresholding the Job Queue

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

6.2.1 Reordering

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

1. FCFS: First-Come First-Serve, which is the most commonly used policy [**easy**].

2. LCFS: Last-Come First-Serve.

3. SPF: Smallest estimated Processing time $\widetilde{p_j}$ First [**bfchar**].

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 [**bfchar**], where the expansion factor is defined as follows:
$$\frac{wait_j + \widetilde{p_j}}{\widetilde{p_j}} \tag{6.1}$$
where $wait_j$ is the time job j has been waiting until the time at which the decision is taken. This corresponds the the value of the stretch(or slowdown) where the user runtime estimate is used.

8. SEXP: Smallest Expansion Factor First

9. LRF: Largest Ratio $\frac{\widetilde{p_j}}{q_j}$ First

10. SRF: Smallest Ratio First

11. LAF: Largest Area $\widetilde{p_j} \times q_j$ First (often called "Total Resources")

12. SAF: Smallest Area First

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

6.2.2 Thresholding

Reordering the job queue means losing the no-starvation guarantee; some individual jobs therefore can wait an undue amount of time. It is possible to introduce a thresholding mechanism to prevent this behavior: When a queued job's waiting time exceeds a fixed threshold Θ, it is at the head of the queue. We denote by EASY(P, Θ)

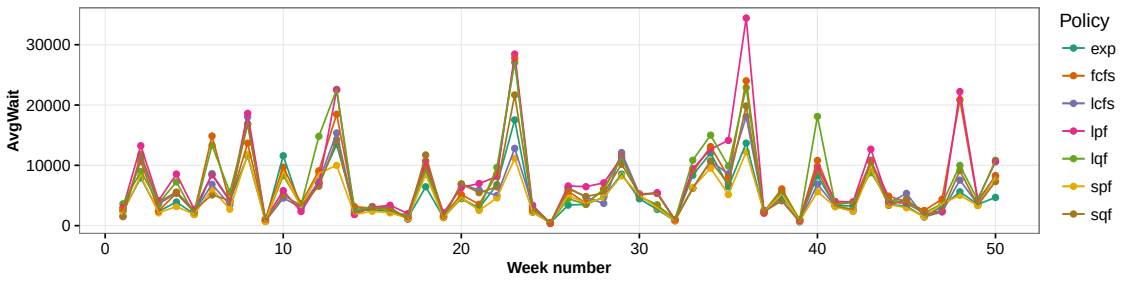


Fig. 6.1: Variability in the weekly average waiting time in the KTH-SP2 trace (see Subsection **??**) for 7 different policies with $\Theta = 0$. The policy set is reduced to 7 policies as not to obstruct the figure with too many colors.

the scheduling policy that starts and backfill jobs according to the (thresholded) reordering policy P. For the sake of completeness, Algorithm **??** describes the EASY(P, Θ) heuristic.

Algorithm 2 EASY(P, Θ) policy

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

Sort Q according to P
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**
  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

6.3 Online Tuning

Here, we present our policy selection strategies. We will refer to the period during which a selected policy is applied as the *policy period* and will denote the length of this period as Δ. The time interval is thus divided into periods of equal lengths

(Δ): $\Delta_0, \cdots, \Delta_T$, where T is the index of the current policy period. A new policy is selected at the beginning of each period and applied during the whole period.

We further assume that there is a certain regularity among periods, i.e. that the distribution of the job submission process does not radically differ between consecutive weekly periods. This assumption is validated in the study presented in [**jsspp17**]. It further entails that the behavior of a policy during the previous periods reflects its behavior on the current one, so that the selection of a policy can be based on its past behavior. However, there may be some variability between different periods for certain cost metrics [**feitelson2001metrics**]. This is illustrated in Figure **??** which displays the average waiting time of various policies for the KTH-SP2 trace (see **??** for a description of the workloads) using weekly periods encompassing one year. As one can note, the average waiting time varies a lot from one week to the other, for all 7 policies considered. This indicates that when the cost metric is averaged over different periods, there is a trade-off to find in between longer periods that would somehow limit the variability, and shorter ones that yield more values for the estimation. We will come back to this issue below.

The selection of a policy is reminiscent of reinforcement learning, where solving a search and inference problem in a (perhaps restrictive) policy space is easier than in the space of the original problem. It is important to note, however, that a pure reinforcement learning approach is difficult to develop in our context. Indeed, while we have outlined a reduced action space, the state space to consider is infinite. This is not prohibitive *per se*, as modern methods [**aprl**] can bypass dimensionality issues via function approximation. However, these methods rely on large amounts of batch data, and we are facing an online problem with no pre-existing data[1]. 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.

6.3.1 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 **??**) 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.

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

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

$$w(t; P_i) = \sum_{j=1}^{l(\Delta_t)} \text{Wait}_j^{P_i} \tag{6.2}$$

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

$$w(\rightarrow T; P_i) = \sum_{t=0}^{T-1} \lambda^{T-1-t} w(t; P_i) \tag{6.3}$$

where $\lambda \in [0, 1]$ is a decaying discount factor that can be used to privilege recent history (*i.e.* recent periods). The above formulation is general and covers several possible situations : From stationary job distributions where λ can be set to 1 to non-stationary distributions where the more recent history has to be privileged. As explained in Section **??**, in our experiments λ is set to 1 as the distributions considered are stationary but the reader has to bear in mind that our framework can be use on different distributions by resorting to different values of λ that can be set by cross-validation.

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

$$P = \text{argmin}_{P_i, 1 \leq i \leq 12} w(\rightarrow T; P_i) \tag{6.4}$$

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

Furthermore, in the context of this study and as detailed in Section **??**, 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

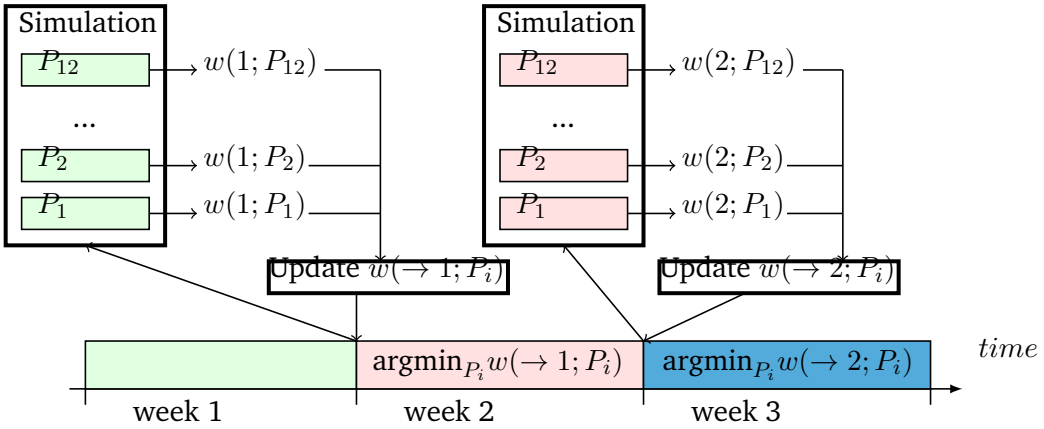


Fig. 6.2: Illustration of the online simulation-based strategy. Every week, the workload from the past week is used to run simulations using every reordering policy. The "model" w is updated according to (**??**) (or (**??**) in the Noisy case from Section **??**) and the policy for the next week is chosen according to (**??**)

.

estimating the cost as defined in Eqs (**??**) and (**??**) 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.

6.3.2 Policy Selection with Noisy Simulation

We take another step to evaluate how the simulation strategy for selecting policies would work in a batch scheduler. Simulation of real systems can be imprecise. Existing work reports values as low as 2% imprecision [**batsim**], we take a very conservative approach and consider an imprecision up to of 20% – more than ten times the reported value. This is chosen large as to make our proof of work more applicable to realistic situations. We randomly introduce multiplicative noise in the estimate of the policy cost defined by Eq. (**??**) by rescaling it, either down or up:

$$w_{\text{noisy}}(t;P_i) = \rho_t^i \sum_{j=1}^{l(\Delta_t)} \text{Wait}_j^{P_i} \tag{6.5}$$

where ρ_t^i is uniformly sampled in the interval $[0.80, 1.20]$, thus adding a +/- 20% multiplicative noise on the waiting time estimated by the simulator. This value is taken as to be sufficiently larger than values previously reported, see e.g. [**batsim**] which reports 2 % variations of the simulated metric compared to the metric on the

real systems. The overall cost is then defined in the same way as before, leading to:

$$w_{\text{noisy}}(\rightarrow T; P_i) = \sum_{t=0}^{T-1} \lambda^{T-1-t} w_{\text{noisy}}(t; P_i) \tag{6.6}$$

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

$$P = \text{argmin}_{P_i, 1 \leq i \leq 12} w_{\text{noisy}}(\rightarrow T; P_i) \tag{6.7}$$

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 **??**, 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.

6.3.3 Policy Selection with ϵ-greedy Exploration

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

Thus, we explore here a more efficient strategy that dispenses with estimating the cost of all policies at a given time and only updates the cost of the policy that is currently being used. This strategy makes use of the ϵ-greedy exploration method, standard in reinforcement learning [**Watkins:1989**] and bandit problems [**Auer2002**], to trade-off exploitation and exploration, and relies, as before, on past estimates of the cost to select the best policy in the exploitation mode.

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

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

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

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

$$w_{\text{epsilon}}(\rightarrow T; P_i) = \frac{1}{\sum_{t=0}^{T-1} \mathbb{1}(P_i, t) l(\Delta_t^{'})} \sum_{t=0}^{T-1} \mathbb{1}(P_i, t) \lambda^{T-1-t} w_\epsilon(t; P_i) \tag{6.8}$$

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

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

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

$$P = \text{argmin}_{P_i, 1 \leq i \leq 12} w_{\text{epsilon}}(\rightarrow T; P_i) \tag{6.9}$$

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

Figure **??** illustrates this process. Several studies study the decay of ϵ over time, the exploration being less important once the estimates for the different policies are reliable [**Tokic:2010**]. 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 **??**. The ϵ-greedy strategy is known to be outperformed in terms of regret by other classical algorithms such as the UCB or EXP family of algorithms (see [**bubnow**]). See [**bubnow**] for a precise definition and thorough treatment of this subject. These algorithms however all rely on pre-normalized rewards. Here, the scale of $w_\epsilon(t; P_i)$ is not known in advance. While the usual doubling tricks can be devised to go

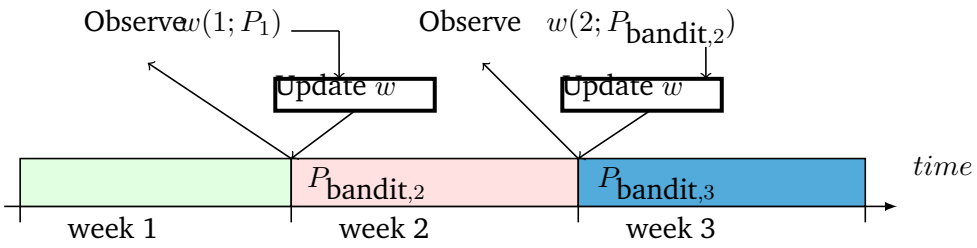


Fig. 6.3: Illustration of the bandit-based setting. Every week, the wokload from the past week is used to run simulations using every reordering policy. The "model" w is updated according to (**??**) and the policy for the next week is chosen according to (**??**). We denote by $P_{\text{bandit},t}$ the policy chosen by the bandit at week t.

.

around sequential issues of scale with good asymptotical performance, they ruin performance in practice. Accordingly, we only perform experiments using the simple ϵ-greedy method.

6.4 Experiments

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

6.4.1 Experimental Protocol

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

Traces

The experimental validation performed in this chapter uses a mixture of small and large systems from different periods. These platforms are all targeted by this work, since they all are homogenous systems that run a batch scheduler. Table **??** presents the 7 logs, which can all be obtained from the Parallel Workload Archive [**Feitelson20142967**]. We use the “cleaned” [**pwa-clean**] version of the logs as per the archive. We impose an additional filtering step to the workloads[2] in order to clean erroneous data:

[2]This filtering step is available in the reproducible workflow [**repro**] as shell script `misc/strong_filter`

Tab. 6.1: Workload logs used in the simulations.

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

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 (no data is available), we remove the job;

3. If the runtime or submission time is negative (no data is available), we remove the job.

Simulator

The choice of a simulator is a critical part of experimental validation that raises a trade-off between precision and runtime. High precision can be obtained by carefully modeling the platform and its network topology, or extracting information from the jobs from their sources or post-mortem logs. This is the approach used by high-fidelity batch scheduling simulators such as BatSim [**batsim**]. In the present work, the focus is on studying the EASY-Backfilling mechanism in itself, without addressing the allocation problem. Moreover, the experimental protocol used here requires many simulation runs. Therefore, we set the precision/runtime trade-off at the point which minimizes simulation runtimes. We discard all topological information relative to the platform and use the job processing times of the original workloads. In this setup, the processors are considered to be undistinguishable from each other, and jobs can be discontinuously mapped to any available processor on the system. During this work, we developed a lightweight simulator [**ocst**] and now release it to the community along with this thesis. See the reproducibility paragraph below for more information. With this simulator, we are able to replay EASY-FCFS on the CEA-CURIE trace in 33 seconds with a machine equipped with an Intel(R) Core(TM) i5-4310U CPU @ 2.00GHz. As a comparison, the Batsim simulator with

network communication modeling disabled and no resource contiguity takes more than 7 hours to replay the same trace on the same hardware. The complete set of simulations presented below is executed on a single Dell PowerEdge R730 in 22 hours, including input preparation and statistical analysis code.

Resampling

Comparing algorithms in batch scheduling is made especially difficult by the fluctuations in commonly used metrics [**feitelson1998metrics**]. The variability in performance is known to outrank the available sample sizes: *It is not enough to simply replay an algorithm using a workload trace*. Indeed, as presented in Figure **??**, the objective function we are using is subject to high variability. In our experience, randomly shuffling months in a trace at random is enough to generate contradictory conclusions about which algorithm is better. This is a known problem [**flurries**] that motivated decades of research in workload modeling [**feitbook**]. Workload models have drawbacks in that they usually lose the details of the workloads. This is problematic due to the non-linear aspect of the job scheduling process. This is discussed in [**feitresampling**], an approach that attempts to solve this problem by resampling data directly from the original traces. In this chapter, we use a simple implementation of these ideas. More precisely, we want to sample the job submission process, which is achieved in the following manner: (a) For every week in a resampled trace, we iterate over every existing user in the original trace; (b) For each of these users, we choose a week of the original trace uniformly at random; (c) The jobs from this user in this original week are chosen to be present in the week being resampled. This simple mechanism enables us to preserve the dependency within a week. Although the system state may not be independent from one week to an other, as illustrated in [**jsspp17**], it is reasonable to assume that the system is in a stationary state. Indeed, [**jsspp17**] performs train/test experiments using the same policy space as we use here. We do provide guidelines as how to incorporate tracking into our algorithms in Section **??** by introducing a decaying discount factor λ. In our experiments, we use stationary resampled data, and therefore no decay is necessary: λ here is set to 1.

Experimental Validation

For each of the 7 workloads considered, we use the original data to resample 60 traces, each with a length of two years. For each resampled trace, we run simulations using the EASY(P, Θ) scheduling heuristic. We fix the value of the job waiting time threshold Θ at 40 hours. The choice of this default stems from

the analysis developed in [**jsspp17**]. Essentially, this value is high enough for queue reorderings to be leveraged while being low enough to provide the functional requirement of preventing starvation. We simulate EASY($P, \Theta = 40h$) using the following strategies:

- All fixed P_i for $1 \leq i \leq 12$ from the search set defined in Subsection **??**. This is done for two reasons. First, it allows to see which policy works well for which platform. Second, it allows to compare how well strategies used in this chapter with the best *a-posteriori* known policy;

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

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

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

Tab. 6.2: Hyper-parameter leave-one-out selection for ϵ.

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

6.4.2 Replicability

We publish a declarative workflow [**repro**] based on our simulator [**ocst**] that runs experiments and generates all the figures from this chapter.

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

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

The archive containing the experimental workflow can be obtained at:

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

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

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

The experiments in this chapter can run in less than 24 hours on a moderately parallel host. More precisely, we had access to a Dell PowerEdge R730 equipped with a total of 28 cores @2.4GHz each (56 threads) and 757 Gigabites of RAM. For

this reason, the experimental workflow was designed to be executed on a single host.

All data from this chapter were obtained from the Swf Parallel Workloads Archive [**Feitelson20142967**

All experiments are tied together using zymake [**zymake**], a minimalistic workflow system designed for computational experiments[3]. This system is analogous to a traditional build system with added workflow capabilities. The entire workflow that generates this chapterfrom the input data is contained in a single `zymakefile` to be found at the root of the main archive. This workflow is composed of the following steps:

- Data extraction from archives, data filtering, and data resampling. This is principally using shell scripts for data manipulation (see file `misc/strong_filter` for the filtering steps used) and ocaml code that implements the resampling method described in **??**.

- Simulation. This step runs the lightweight ocaml backfilling simulator specially written for this work. This simulator is made available under the ISC [**isc**] license as a persistent zenodo archive at [**ocst**].

- Analysis of the results. This step runs R code that generates the figures presented in this chapter.

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

6.4.3 Results

Analysis of Basic Policies

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

[3]The zymake system is also packaged by our nix expressions.

Tab. 6.3: Average total waiting time diminution of fixed policies with respect to EASY(FCFS). The precise definition of the value reported is provided in Eq (**??**)

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

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

$$\bar{w}(T_k, P) = \sum_{j=1}^{l(T_k)} \text{Wait}_j^P \tag{6.10}$$

The values appearing in Table **??** are exactly:

$$\frac{\sum_{k=1}^{k=60} \bar{w}(T_k, \text{EASY}(P_i, 40h)) - \bar{w}(T_k, \text{EASY}(\text{FCFS}, 40h))}{\sum_{k=1}^{k=60} \bar{w}(T_k, \text{EASY}(\text{FCFS}, 40h))} \tag{6.11}$$

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

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

In the present work, we are studying adaptive policies that obtain a good performance on any given system by selecting one of these policies. Figure **??** reports the evolution of the average cumulative waiting time for the UniLu-Gaia trace. This figure shows how the average improvement in cumulative waiting time evolves as a function of time. In other words, this is the average curve obtained if one were to

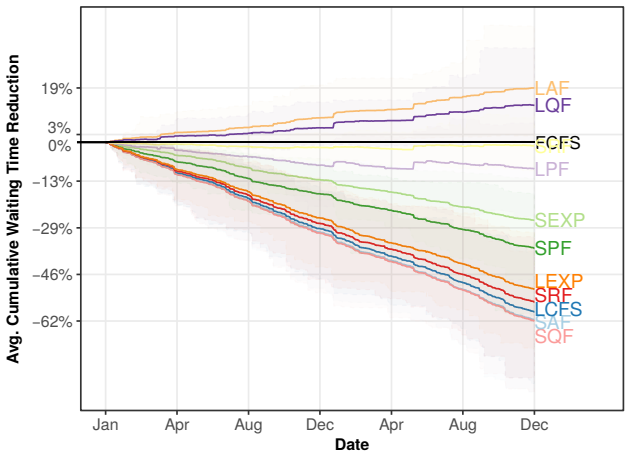


Fig. 6.4: Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the policies P_i for $i = 1 \dots 12$ on the UniLu-Gaia trace. The average is obtained by resampling the original trace 60 times. The dashed lines represent the 10th and 90th percentiles of the values across this resampling.

plot the cumulative sum of job waiting times for every resampled trace, incrementing the sum when a job finishes. Accordingly, the final values attained by every curve correspond exactly to the values reported in Table **??**. The dashed values represent the 10th and 90th percentile of the values.

Observe that while the 10th and 90th percentiles of the improvement with respect to EASY-FCFS are large, the values are still contained enough to make definite statements about how these policies compare to the FCFS baseline. This figure does not report the variability of the policies against one another, and does not allow to compare them beyond their average performance. The fact that the size of the percentile band is roughly half the best attainable improvement was our original motivation for this work, as one can expect to be able to distinguish between these policies in an online manner.

Analysis of Policy Selection Strategies

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

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

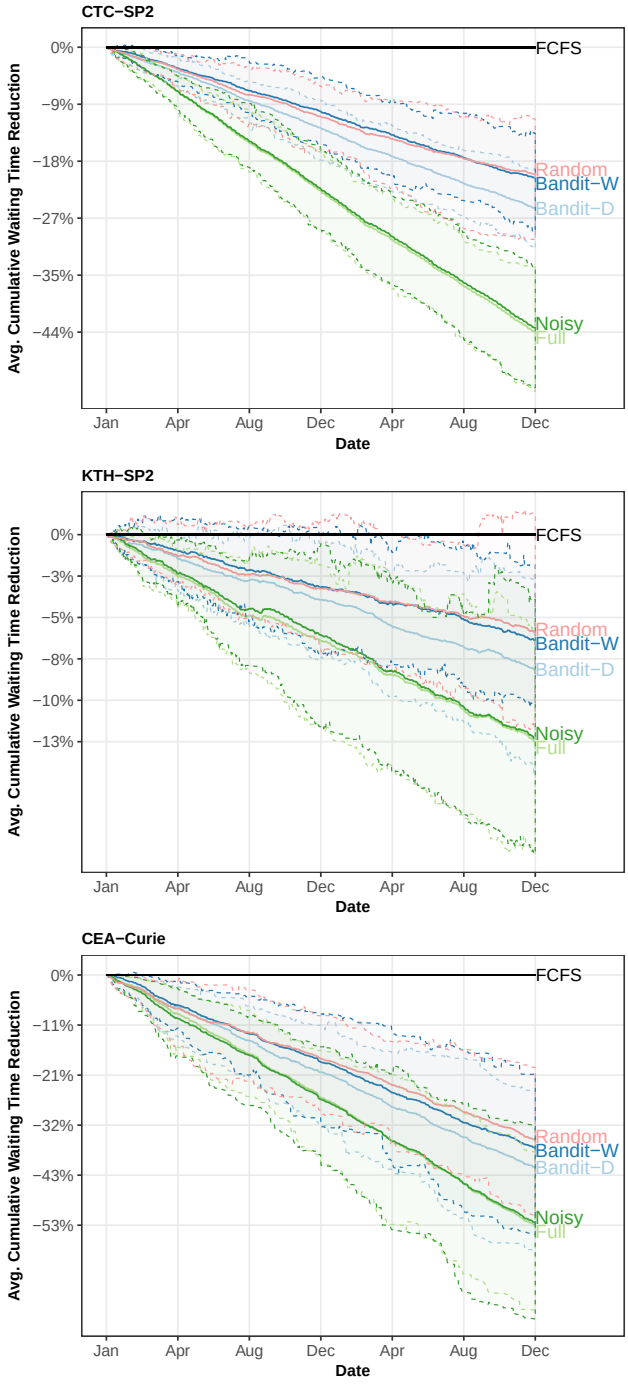


Fig. 6.5: Evolution of the average cumulative waiting time improvement compared to EASY-FCFS of the Simulated Feedback, Noisy Simulated Feedback and Bandit Feedback policies. The average is obtained by resampling the original trace 60 times. The dashed lines represent the 10th and 90th percentiles of the values across this resampling. Each figure is a different trace. This figure is followed-up in figure **??** for the remaining UniLu-Gaia trace.

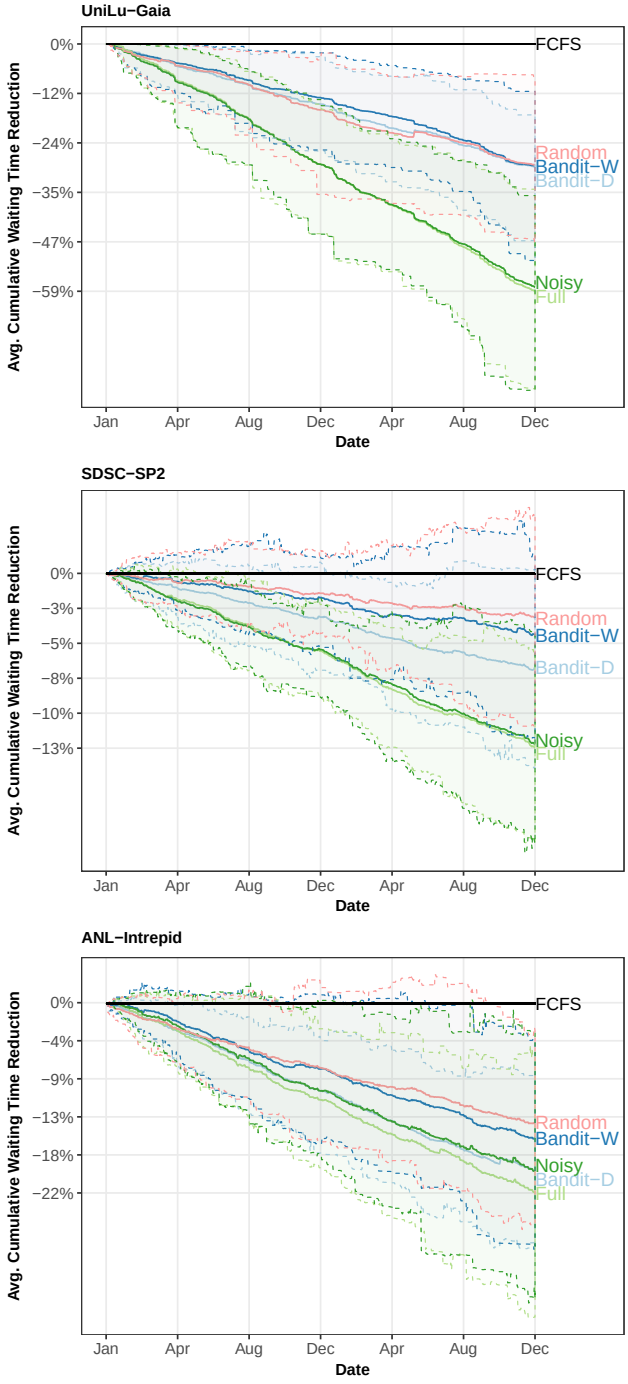


Fig. 6.6: Figure **??** continued. Plot for the UniLu-Gaia, SDSC-SP2, Anl-Intrepid logs.

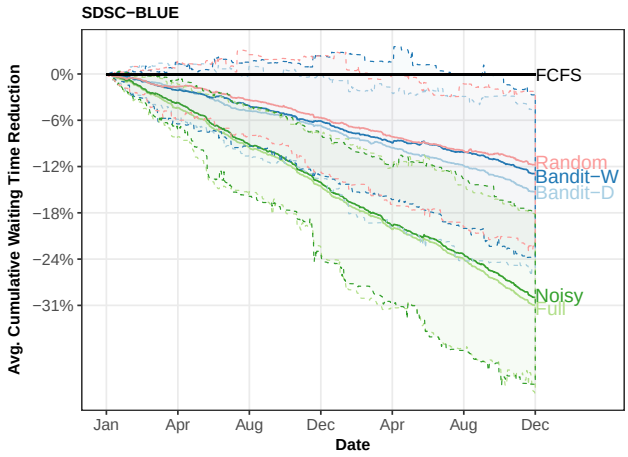


Fig. 6.7: Figure **??** continued. Plot for the SDSC-BLUE log.

Tab. 6.4: Average Cumulative waiting time improvement of the policies used with respect to EASY(FCFS).

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

- Bandit Feedback (Δ =1 day)
- Simulated Feedback (Δ =1 week)
- Noisy Simulated Feedback (Δ =1 week)

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

FCFS vs Random. Randomly taking a policy in the search set defined in **??** is better than using the FCFS order. This is due to the known fact that the FCFS order is not optimal for optimizing the average waiting time objective, and confirms that the search set is well chosen. The value of the period length Δ has an impact on the performance of the random policy. Indeed, shorter periods lead to better performance.

The Simulated Feedback strategy consistently performs almost as well as the best fixed policy. The values attained by the exact simulation based strategy almost

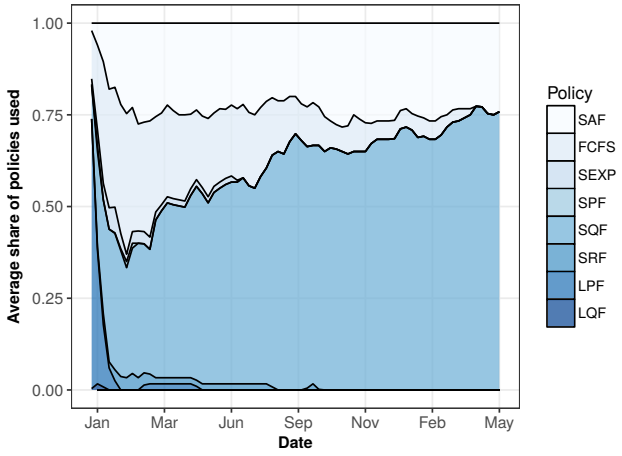


Fig. 6.8: Share of the policies chosen by the Noisy policy (Δ =1 day, $\lambda = 1$) as a function of time for the UniLu-Gaia trace. The average choice is obtained by resampling the original trace 60 times and aggregating by date. The legend excludes policies that are never used by the strategy.

match the highlighted values from Table **??**. This corresponds to the curve labeled "Full" in Figures **??** and **??**. Moreover, these average cumulative waiting time curves exhibit linearity (as opposed to being concave), which further shows that the choice of the best policy can be made rapidly. Note that the value of Δ has no impact over the performance of this strategy.

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

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

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

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

policies, LAF and LQF, are the best overall policies. This means that the estimation converges to the correct point.

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

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

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

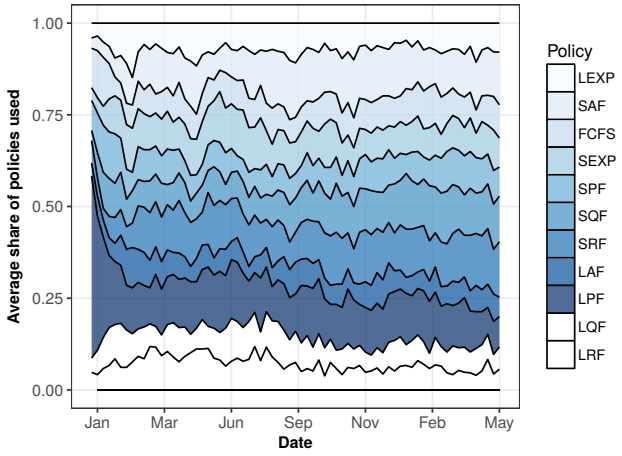


Fig. 6.9: Share of the policies chosen by the Bandit Feedback(Δ =1 day, $\epsilon = 0.5, \lambda = 1$) policy as a function of time for the UniLu-Gaia trace. The average choice is obtained by resampling the original trace 60 times and aggregating by date. The legend excludes policies that are never used by the strategy.

6.5 Conclusion

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

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

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

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

7 Conclusion

Improving user metrics is a crucial challenge for High Performance Computing. This challenge is made especially difficult by the online parallel job scheduling problem that is at the core of managing these machines. This problem has two challenging aspects. First, it is hard algorithmically, even in offline settings. Second, its parameters are subject to high uncertainty. Decisions for this hard problem happen under strict performance constraints and therefore list heuristics are usually applied. There are many studies considering alternative list heuristics, which are usually performed in the context of one or several platforms of the same class. In this thesis, we take a different viewpoint and argue that it should be useful for the field to have generic batch scheduling software that may adapt to any parallel system. We propose to leverage system metadata and performance feedback to achieve this adaptivity. Making data-based choices motivates the usage of machine learning techniques. Accordingly, this thesis focuses on two particular approaches adapted to our setting.

Our first approach to improving user metrics is to reduce the uncertainty in the parameters of the problem. This is the method outlined in Chapter **??**, where we use supervised learning to predict job running times. An online regression model is used to continuously learn and predict running time values from job metadata throughout the system life cycle. Instead of relying on the standard approach of feeding user-provided upper-bounds to a list heuristic, we propose using predictions from this regression model, to achieve better performance. The approach is succesful, but requires heavy simulation use in order to cross-validate the hyperparameters of the scheduling heuristic. Moreover, the above approach exhibits poor behavior with respect to the higher statistical moments of the considered user metrics.

Our second approach attempts to solve these remaining issues. We take a step back and consider learning the scheduler directly in policy space. We believe this to be the larger, more important question for future resource management software. In order to achieve adaptivity in policy space, we have to choose a search space and a way to parametrize policies inside. This is what we study in Chapter **??**, which explores a larger framework for the tuning of scheduling heuristics. We consider a discrete set of reordering policies for job waiting queues, along with a thresholding heuristic. This framework is expressive enough to allow for halvening the average waiting

times, while being amenable to train/test policy selection among a small discrete set in simulation. Additionally, this chapter studies how to appropriately set the values for the thresholding heuristic, which can be seen as a risk management trick for extreme values of waiting times. Finally, in Chapter **??**, we study how to learn policies online in practice using this framework. We show the tradeoffs between using simulations or a multi-armed bandit algorithm. While having access to a truthful simulator allows to learn the best policy in a discrete set very fast, a simple period-based epsilon-greedy algorithm works half as well, and does not require any simulation to be run.

Most large HPC centers delegate the setting of the scheduling policy to the system's administrators. The main message from this thesis to the institutions that manage these systems is that it is appropriate and gainful to move schedulers away from manually set or FCFS-based policies torward techniques that take system feedback into consideration. Indeed, our experiments show that simple reordering strategies allow to considerably minimize user-based objective metrics. We can leverage this fact using adaptive technique and change the human-in-the-loop way to set scheduling policies. Future systems should use systematic techniques that learn to optimize global system metrics.

We outline three directions for future research in this direction.

Simulation method and data resampling In this thesis, we used a bare-bones simulation model. This was necessary because of the exploratory nature of chapters **??** and **??**, where a large quantity of simulations had to be performed. Now that the multi-armed bandit approach was proven in this simulation context, it would be interesting to implement in batch schedulers and experiment using emulation techniques. Sophisticated models would most probably increase the disparity in behavior between different machines. A key point in HPC systems simulation is the way the data is collected and generated. Indeed, user metrics are volatile and thus, the more data the better. In this thesis, we used data resampling as a way to compute meaningful statistics through this volatility. This is also an area that should benefit from more careful attention.

Search Space The search space defined by Chapter **??** can be naturally extended to parametrized reordering policies. An important question is how to parametrize the reordering function in order to be able to optimize and generalize. Answering this question may lead to a more powerful method. A closely related question is that of optimization: While a larger search space may provide better policies and generalize too, it might be hard to optimize into. A recent surge in smooth automatic differentiation techniques may reveal to be a useful solution to this problem [**inala2018reas**, **chaudhuri2010smooth**]. Indeed,

the full-chain differentiation of a simulation method would enable to perform stochastic search techniques. This step forward would allow for optimizing globally in the parameter space with a tractable the number of (differentiated) simulation runs.

Learning from Feedback Online learning algorithms that use feedback are in the author's view a very promising tool for the control future extreme large-scale systems. Indeed, as systems become larger and more heterogenous, taking rule-based decisions informed by system characteristics becomes harder. This is true both for theoretical and experimental approaches, as both modeling and/or simulation of such large appliances is a long and costly process. Online learning methods such as Reinforcement Learning or Multi-Armed Bandits propose to optimize a system seamlessly during its operation. We believe this to be is a key tool for future systems. Many interesting machine learning problems arise from large-scale systems. For instance, figure **??**, shows that the rewards of different scheduling policies are somewhat correlated. A more efficient bandit technique should take this into consideration. Moreover, context information is available, which could lead to the succesful application of contextual bandit techniques. Another, more general question is whether Reinforcement Learning techniques could be applied to such systems.

List of Figures

List of Tables

Additionally, the work conducted in this thesis directly led to the following communications.

Peer-reviewed international journals

Peer-reviewed international conferences

Peer-reviewed international workshops

Abstract

Providing the computational infrastructure needed to solve complex problems arising in modern society is a strategic challenge. Organisations usually address this problem by building extreme-scale parallel and distributed platforms. High Performance Computing (HPC) vendors race for more computing power and storage capacity, leading to sophisticated specific Petascale platforms, soon to be Exascale platforms. These systems are centrally managed using dedicated software solutions called Resource and Job Management Systems (RJMS). A crucial problem addressed by this software layer is the job scheduling problem, where the RJMS chooses when and on which resources computational tasks will be executed. This manuscript provides ways to adress this scheduling problem. No two platforms are identical. Indeed, the infrastructure, user behavior and organization's goals all change from one system to the other. We therefore argue that scheduling policies should be adaptive to the system's behavior. In this manuscript, we provide multiple ways to achieve this adaptivity. Through an experimental approach, we study various trade-offs between the complexity of the approach, the potential gain, and the risks taken.

Résumé

Fournir les infrastructures de calcul nécessaires à la résolution des problèmes complexes de la société moderne constitue un défi stratégique. Les organisations y répondent classiquement en mettant en place de larges infrastructures de calcul parallèle et distribué. Les vendeurs de syst'emes de Calcul Hautes Performances sont incités par la compétition à produire toujours plus de puissance de calcul et de stockage, ce qui mène à des plateformes "Petascale" spécifiques et sophistiquées, et bientôt à des machines "Exascale". Ces systèmes sont gérés de manière centralisée à l'aide de solutions logicielles de gestion de jobs et de ressources dédiées. Un problème crucial auquel répondent ces logiciels est le problème d'ordonnancement, pour lequel le gestionnaire de ressources doit choisir quand, et sur quelles ressources exécuter quelle tache calculatoire. Cette thèse fournit des solutions à ce problème. Toutes les plateformes sont différentes. En effet, leur infrastructure, le comportement de leurs utilisateurs et les objectifs de l'organisation hôte varient. Nous soutenons donc que les politiques d'ordonnancement doivent s'adapter au comportement des systèmes. Dans ce manuscrit, nous présentons plusieurs manières d'obtenir cette capacité d'adaptation. A travers une approche expérimentale, nous étudions plusieurs compromis entre la complexité de l'approche, le gain potentiel, et les risques pris.