Understanding and improving HPC performance using Machine Learning and Statistical analysis

Salah Zrigui

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Salah Zrigui

Thèse dirigée par **Denis TRYSTRAM**
et codirigée par **Arnaud LEGRAND**

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

Understanding and improving HPC performance using Machine Learning and Statistical analysis

Thèse soutenue publiquement le **25-03-2021** ,
devant le jury composé de :

Denis TRYSTRAM
Professeur des universités, LIG, Grenoble INP, France, Directeur de thèse
Arnaud LEGRAND
Directeur de recherche CNRS, LIG, France, Directeur de thèse
Jean-Marc Pierson
Professeur, Université Paul Sabatier, Toulouse, France, Rapporteur
Rizos Sakellariou
Professeur, University of Manchester,England, Rapporteur
Massih-Reza Amini
Professeur,Université Grenoble Alpes,France, Examinateur
Maria S. Perez Hernandez
Professeur, Universidad Politecnica de Madrid,Spain, Examinatrice
Frédéric Desprez
Directeur de recherche, INRIA, France, Président
Patricia Stolf
Maitre de conference, Université de Toulouse Jean-Jaurès, Examinatrice

Abstract

The infrastructure of High Performance Computing (HPC) systems is rapidly increasing in complexity and scale. New components and innovations are added at a fast pace. This instigates the need for more efforts towards understanding such systems and designing new, more adapted optimization schemes. This thesis is a series of data-driven analytical and experimental campaigns with two goals in mind. (i) To improve the performance of HPC systems with a focus on scheduler performance. (ii) To better understand the inner workings of HPC systems, which includes scheduling evaluation methods and energy behavior of submitted jobs.

We start by performing a comparative study. We focus on the evaluation methods of schedulers. We study two well-established metrics (waiting time and slowdown) and one less popular metric (per-processor-slowdown). We also evaluate other effects, such as the relationship between job size and the slowdown, the distribution of slowdown values, and the number of backfilled jobs. We focus on the popular First-Come-First-Served (FCFS) and compare it to other simple scheduling policies. We show that relinquishing FCFS is not as risky as it is perceived to be. We argue that using other ordering policies in combination with a simple thresholding mechanism can offer similar guarantees with significantly better performance.

Then, we proceed to show the limits of simple scheduling policies and we design and test two machine learning-based paradigms to improve performance beyond what these basic policies can offer. First, we propose a method to dynamically generate new scheduling policies that adapt to the changing nature of data in any given platform. Also, we study the possibility of applying online learning on scheduling data, and we detail the difficulties that one might encounter in such an endeavor.
For the second approach, we improve the performance of already established scheduling policies by reducing the inherent uncertainty in the scheduling data. More precisely, the inaccuracy of user runtimes estimates. We propose a simple classification of jobs into small and large. We show that this classification is sufficient to harness most of the improvement that can be gained from accurate runtimes estimates. We use machine learning to predict the classes and improve performance across all studied platforms.

Finally, we analyze the energy consumption of HPC platforms. We study the energy profiles of individual jobs. We Observe the similarities and differences between energy profiles and we propose a series of statistical tests through which we classify the jobs into periodic, constant, and non-stationary. We believe that this classification can be used to predict the energy consumption of future jobs and build energy-aware schedulers.

Abstract (French)

Les infrastructures de systèmes de calcul haute performance (HPC) augmentent rapidement en complexité et en échelle. De nouveaux composants et innovations sont ajoutés à un rythme rapide. Cela incite à redoubler d'efforts pour comprendre ces systèmes et concevoir de nouveaux schémas d'optimisation plus adaptés. Cette thèse regroupe une collection de travaux analytiques et expérimentaux avec deux objectifs. (i) Améliorer les performances des systèmes HPC en mettant l'accent sur les performances des gestionnaires de ressources. (ii) Mieux comprendre le fonctionnement interne des systèmes HPC, qui comprend les méthodes d'évaluation des ordonnanceurs et le comportement énergétique des jobs.

Nous commençons par présenter une étude comparative. Nous nous concentrons sur les méthodes d'évaluation des ordonnanceurs en étudiant deux métriques classiques (temps d'attente et ralentissement) et une métrique moins étudiée (ralentissement par processeur). Nous évaluons également d'autres effets, tels que la relation entre la taille des jobs et le ralentissement, la distribution des valeurs de ralentissement et le nombre de jobs remplacés. Nous nous concentrons sur la politique populaire First-Come-First-Served (FCFS) et la comparons à d'autres heuristiques de planification simples. Nous montrons que l'abandon du FCFS n'est pas aussi risqué qu'on le perçoit. Nous soutenons que l'utilisation d'autres politiques combinées avec un simple mécanisme de seuillage peut offrir des garanties similaires avec des performances nettement meilleures. Ensuite, nous montrons les limites des heuristiques d'ordonnancement simples et nous concevons et testons deux paradigmes basés sur l'apprentissage automatique pour améliorer les performances au-delà de ce que ces heuristiques de base peuvent offrir. Tout d'abord, nous proposons une méthode pour générer de manière dynamique de nouvelles politiques d'ordonnancement qui s'adaptent à la nature changeante des données sur une plateforme donnée. Nous étudions également la possibilité d'appliquer l'apprentissage en ligne sur ce type des

données, et nous détaillons les difficultés que l'on pourrait rencontrer dans une telle entreprise.

Pour la deuxième approche, nous améliorons les performances des politiques d'ordonnancement déjà établies en réduisant l'incertitude dans les jobs. Plus précisément, l'inexactitude des estimations des temps d'exécution des utilisateurs. Nous proposons une classification simple des jobs en petits et grands selon leur taille. Nous montrons que cette classification est suffisante pour exploiter la plupart des améliorations qui peuvent être obtenues à partir d'estimations précises des délais d'exécution. Nous utilisons l'apprentissage automatique pour prédire les classes et améliorer les performances sur toutes les plateformes étudiées.

Enfin, nous analysons la consommation énergétique des plateformes HPC. Nous étudions les profils énergétiques des jobs individuels et nous observons les similitudes et les différences entre les profils énergétiques. Puis, nous proposons une série de tests statistiques à travers lesquels nous classons les jobs en périodiques, constants et non stationnaires. Nous pensons que cette classification peut être utilisée pour prédire la consommation d'énergie des futurs jobs et pour construire des ordonnanceurs sensibles à l'énergie.

Acknowledgement

This work was not the result of a single person's effort, it is the fruit of multiple years of labor by the author and many others before him. I dedicated this section to show my appreciation to all those who contributed to the realization of this thesis.

I would like to start by thanking the members of the jury, especially the two reviewers Rizos Sakellariou and Jean-Marc Pierson for accepting to evaluate my work and providing me with insightful comments on my dissertation.

I would like to thank my first Ph.D. advisor Denis Trystram for all his support and contribution during the thesis and the occasional philosophical talk I had the pleasure to exchange with him. I would also like to thank my second Ph.D. advisor Arnaud Legrand to whom I owe the better part of my skills as a researcher. His advice, guidance, and relentless pursuit of better research quality had such a great impact on my work.

I thank the members of the teams DataMove and Polaris for all the help they provided. I am grateful to my dear friends and colleague Danilo Carastan-Santos with whom I had the pleasure to collaborate. I would also like to thank Valentin Reis, the person that I started my research journey with during my master's. I thank all the friends and colleagues that I had the pleasure to meet at LIG and Grenoble. You guys helped in ways that cannot easily be translated into words.

Last but not least, I would like to thank my parents, especially my father for providing me with the emotional and moral support that enabled me to carry through the hardships I encountered throughout this journey.

Contents

1 Introduction

1.1 High-performance-computing

High-performance-computing (HPC) has become an essential tool for science and industry to make breakthroughs and generate new discoveries. A few decades ago, only the largest institutions (e.g, military) and the biggest multinational corporations had access to supercomputers. But the continuous progress in technology and production techniques caused the cost of computing power to drop significantly over the last few years, opening new, more diverse, markets and making supercomputing accessible to new fields in the academic, commercial, and institutional worlds. Nowadays, almost every field relies on high-performance computing in one form or another. Supercomputers are no longer a luxury that only a few entities can afford, it's a necessary tool one must have to maintain a competitive edge in both research and industry. We refer to this phenomenon as the democratization of HPC.

Supercomputing, as the name entails, is the use of powerful, massively parallel computers to solve complex computational problems that would take normal computers years if not decades to solve. The presence of supercomputing can be sensed in almost every aspect of our daily lives. From weather prediction and automotive designs to biomedical applications.
Weather applications include forecasts (possibly the most known application of supercomputers), aircraft flight simulations, and natural disaster prediction. In the biomedical field, HPC is almost ubiquitous, genomic sequencing, drug designs via simulations of chemical reactions, and molecular modeling are but a few of the applications that rely on supercomputers. In Automotive designs, HPC plays an important role in accelerating the production and test cycle while simultaneously reducing its cost, enabling high fidelity crash simulations, new materials behavior simulation, and topological optimization (vehicle shape).

Performing such feats requires massive computational power. To give an idea of the scale of such machines, we refer the reader to the Top500 [1] site that maintains a ranking of the most powerful supercomputers. At the time of writing this thesis, 4 supercomputers managed to achieve a performance that exceeds the 100 PeraFlop/s mark that is 100×10^{12} flops (floating-point operations). The current top of the

list is the *Fugaku* supercomputer, boosting over 7,630,500 cores and capable of performing around 537 PFlop/s. Such great computational power always comes at an equally great cost that takes two forms. The cost of designing and building said supercomputers and the cost of maintaining and operating them.

1.2 Energy Consumption

Top Supercomputers consume between 10 and 20 megawatts. With such an exorbitant cost, it is becoming more and more clear that efficient management of supercomputers is a must and the “performance at any cost” approach can no longer be maintained. Especially as we are pushing toward the exascale age.
In the past, the cost of operating supercomputers was relatively manageable and optimizations (beyond what basic scheduling policies can afford) were desirable but not mandatory. Possible gains (in performance and cost of operating) were always sacrificed for the use of straightforward algorithms that offer simplicity and "perceived" fairness. Now with current Petascale machines and the coming of the exascale, the cost of operating such machines is increasing to impossible levels and better management and usage of supercomputers is no longer just desirable but mandatory. At the same time, clarity in decision making and fairness between users is still a top priority. Thus, future management techniques and algorithms should aim for optimal performance and efficient energy consumption, and interpretability at equal levels.

1.3 HPC management

Supercomputers are usually shared machines. Multiple users submit their applications, also known as jobs, programs that usually require high computational power. These jobs are then sorted into one or several queues and allocated to the proper resources available in the machine. To perform the aforementioned tasks, Supercomputers rely on Resources and Job Management Systems (RJMS) such as OAR [20], SLURM [60] for motoring and control. RJMSs are responsible for the orchestration between the jobs waiting to be executed and the available resources. They manage user’s access to the machines, by setting priority and ordering their job requests in waiting queues. Also, RJMS are usually responsible for resource allocation which includes designating and reserving which parts of the machine each job will be executed on. Such a task is not trivial, as it involves balancing multiple layers and objectives; Making sure that users are satisfied, allocating the proper resources, and maintaining several cost metrics under control (e.g waiting time,

energy consumption,...).

1.4 Problem overview

1.4.1 Solutions of HPC performance optimization

Numerous solutions and paradigms have been proposed throughout the literature, each is unique in its methods and objectives. A solution is heavily tied with the assumptions, objectives, and context. These elements are influenced by the researcher's view of the problem and how it must be tackled. Assumption can be implicit and are not always made clear. An assumption generally includes a certain understanding of the problem and an objective (or multiple) the author is trying to achieve. To better understand these assumptions and their entailing solutions, we propose the following 'classification' of the HPC scheduler optimization's body of work.

A solution is based on one of two assumptions. (i) All the platforms are unique and different from one another and the proposed method must be equally unique. (ii) All the platforms share common traits and the proposed method must focus on these common traits.

- (i) The first assumption is that supercomputers and their workloads are different from one another and a specific solution must be tailored for each situation. They study and detect patterns and anomalies within the same platform (certain bottlenecks, differences between periods, ...). After clarifying the problem, those works usually proceed to propose a solution that is specifically tailored to the problem.
  They could be divided into two subtypes:

  - The first subtype includes the detection and solving of a specific problem in HPC scheduling like GPU bottlenecks (for a certain platform), network congestion... The study is usually done using very specific data (that is sometimes characteristic of the platform) and the verification and testing are done on data that comes from the same platform. This does not mean that they are useless outside the scope of the platform they were conducted on. They can are meant to be transferable provided that proper conditions are met (similar data and/or similar problem).

  - The second subtype argues that HPC platform input is not static and evolves through time. It claims that these evolutions must be studied and

quantified and that scheduling algorithms should also evolve through the life of the supercomputer and adapt to the workload [42, 68].

- (ii) The second approach relies on the similarity aspect between platforms. This similarity is generally proven by performing a statistical study on various different platforms. Then a solution is proposed. The latter relies either on common sense and simplicity or a scheduling technique that is derived from the statistical study [41, 21, 65].
  The main selling point of this approach is its simplicity of implementation and that a certain level of improvement is "guaranteed" regardless of the platform. Also for most cases, it does not require any platform-specific knowledge.

  This assumption does not negate the previous. Although Supercomputers are different, certain traits are common across all the platforms. Some of these traits include job size distribution, the response of the platform to scheduling policies. This view offers many advantages when developing a scheduling scheme. It is transferable from one platform to another, and it tends to be simpler and less convoluted than the first type.
  Similarity-based solutions come in two major forms.

  - Generic scheduling policies. This subtype is the most popular in production systems because of the simplicity and the clarity it offers. EASY-FCFS [79] is particularly popular because it also adds the perception of fairness to the scheduler and prevents starvation (EASY-FCFS is discussed in detail in Section 3.1.3). But the inner workings of generic scheduling policies are not properly understood.

  - Machine learning-based solutions [21, 78, 68, 69, 46]. They include works that rely on machine learning to find an approach that can fit all the possible scenarios. The final outcome of the learning is usually a static scheduling scheme that is not very different from the generic scheduling policies.

Finally, it's worth noting that solutions don't always subscribe to one end of the spectrum or the other and there are also studies that are in between.

Our work falls under the first category. We detected and analyze common behavior between HPCs and propose solutions that are applicable across multiple platforms

1.4.2 Machine Learning to improve HPC systems

Understanding the goals we want to achieve and the context we are in is one of two pillars to build a proper solution. The second pillar is identifying the tools that will best help achieve these goals.
With the ever-increasing complexity of HPCs and the growth of the number of applications and users. Devising scheduling schemes that go beyond basic dispatching and allocation rules is no longer a trivial task. As HPC platforms evolve so should the methods and tools used to manage and improve them. Devising scheduling schemes that go beyond basic dispatching and allocation rules is no longer a trivial task. As HPC platforms evolve so should the methods and tools used to manage and improve them. This instigates the need to use tools that can analyze and extract knowledge from large complex structures.

Machine learning has been receiving great interest as a vehicle to detect and predict complex patterns. This interest is due to its remarkable success in various fields such as computer vision and natural language processing.
This interest is also shared within the scheduling community but the progress has been a little shy due to various reasons. Chiefs among them are the lack of sufficient and/or proper scheduling data in the past (generally due to the sensitive nature of data generated by HPC applications), a limited understanding of the inner-working of real-life schedulers, and the discrete nature of the scheduling problem which fundamentally differs from the continues optimization scheme that characterizes most ML algorithms. But in recent years the HPC community started to take more interest in ML-based solutions and logging data and techniques have been improving steadily.

In this work, we use a data-driven machine learning approach to extract actionable information from logs of real-world systems. We explore the benefit of a careful study of HPC generated data.

1.5 Content and Contributions

This thesis is a series of data-driven analytical and experimental campaigns with two goals in mind. To improve the performance of HPC systems with a focus on improving scheduler performance and better understand the inner workings of HPC systems.
The remainder of this thesis is organized as follows. Chapter 2 presents the state of the art of scheduling and Machine learning for HPC. It gives an overview of Online scheduling and Machine learning and the interaction between the two. Chapter 3

details the experimental framework we adopted throughout this thesis. It introduced the algorithms, the objective functions, and simulation tools used.
Chapters 4, 5, 6, and 7 detail our contributions:

- In Chapter 4, we perform a rudimentary study of classical scheduling schemes and practices. We take a deep look at the EASY-backfilling heuristics (see Section 2), and the FCFS index policy. We conduct a comparative experimental campaign during which we use several orthodox evaluation metrics like the waiting time and the slowdown. But we don't limit our study to basic metrics, we also explore more detailed methods and statistics. Throughout this study, we Compare EASY-FCFS with other scheduling algorithms that offer the same guarantees.

- In Chapter 5, we propose a new method to dynamically generate scheduling policies that adapt to the nature of the workload. We combined several characteristics extracted from the jobs in a linear expression. We use historical data to tune the resulting policies. We conclude that using historical data to predict good scheduling policies for future jobs is not a straightforward task as we observe the drastically changing nature of the workload itself from one time period to the next. We also show that generic scheduling policies are far from optimal and that there is considerable room for performance improvement.

- In Chapter 6, we investigate one of the most known inaccuracies in the scheduling data, the user-provided runtimes estimates. We propose a simple scheme to reduce this inaccuracy. We classify the jobs into jobs with short runtimes and jobs with long runtimes and we slightly modify the scheduling heuristic to take our classification into account. This method shows impressive improvements in performance.

- In Chapter 7, we analyze the energy consumption of HPC platforms. We study the energy profiles of individual jobs. Although every job has a unique energy profile, some similarities between jobs can be identified. From this observation, we propose a series of statistical tests through which we classify the jobs into periodic, constant, and non-stationary. We observe that the percentage of jobs with a predictable energy profile (periodic and constant) is significant enough to be exploited by scheduling methods.

Finally, in chapter 8, we conclude our dissertation and give some perspectives for future investigations.

2 Background and Related work

2.1 Scheduling In HPCs

In the literature, the parallel job scheduling problem, in its majority has been studied under the view of a more general and closely related problem called multiple-strip packing problem [40]. Strip packing is a family of optimization problems that deal with finding a good arrangement of multiple items in a containing region like the resources of a machine. Scheduling and strip packing are not limited to the HPC world however, in fact, the field has one of the largest possible applications in areas of business and industry. Material properties manipulation, cutting process and bending process are but few examples of applications [55].
The multi-strip packing problem, as well as its multi-resource scheduling analog, is NP-complete [14]. A problem is NP-complete if all the currently known algorithm for finding an optimal solution requires a number of computational steps that grows exponentially with the problem size. Thus alternative approaches have been proposed instead, such as linear programming [38, 26], genetic algorithms [75, 56], and neural networks [6]. Xhafa and Abraham [96] present an overall review of scheduling algorithms, mainly focused on HPC platforms. These alternative approaches however do not guarantee to find the optimal solution, thus sacrificing solution quality to computational efficiency.

Scheduling in HPC environments is inherently Online which adds several layers of complexity. In an Online setting, scheduling decisions are taken without complete information about the tasks to schedule. This lack of knowledge comes in two major forms: the arrival time and the running (processing time) of the jobs. (i) Arrival times: jobs arrive at any time and scheduling decisions must be made "on the fly" without knowledge of any future job. (ii) Processing times: The exact runtime of any given job is not known in advance and its exact value can only be known at the end of the execution. Exact runtimes are often replaced by user provide estimates that are not accurate.

2.2 Machine learning for HPC scheduling

The evolving architecture of HPC platforms and the ever-changing nature of its users over time coupled with inaccurate runtime estimates makes attempting to determine a good scheduling scheme an elusive goal. To understand such uncertainties or at least try to circumvent them, many researchers have started evaluating the use of machine learning techniques. Throughout the literature, a wide range of learning-based solutions have been proposed. We distinguish two main approaches: (i) reducing the uncertainty in the scheduling data by adjusting job runtime estimates, and (ii) directly designing a scheduling scheme that improves specific objectives.

2.2.1 Reducing the uncertainty in the scheduling data

One of the main sources of the complexity of the online scheduling problem is the lack of information. Other than some essential dimensions like the number and possibly types of machines and the upper bound for the runtimes, nothing is known about the job. Important but absent pieces of information include the runtime (possibly the most important piece of information), memory requirements, I/O bottlenecks. Obtaining such knowledge a priori is very hard and even impossible. It requires users to have a very "intimate" knowledge about their jobs and the system.

Numerous approaches have been proposed to reduce inaccuracy and predict the missing information HPC jobs. Such works rely on feature engineering and machine learning to build a prediction framework. Generally, the features extracted in these works–including the contribution of this thesis– share certain similarities The history of the user and basic job characteristics: arrival time, requested processing time, and requested resources. (Section 3)

The exact runtime of jobs has received the greatest attention. It is well known that user-provided runtime estimations are far from accurate [9, 34]. Feitelson *et al.* introduced EASY++, a variation of the classical EASY strategy, which replaces user-provided runtime estimates by the average runtime of the two previous jobs submitted by the same user [90]. Despite its simplicity, it allowed for an improvement of around 25% over the classical EASY algorithm. In [80], the authors leveraged historical data by going through the platform logs to find the most similar jobs. They use features such as userID, date of submission, requested time, and resources. These fundamental features are still used in the following works. Gaussier *et al.* improved upon [90] by using used historical data from different traces and linear regression to predict runtimes with improved accuracy [42]. They also showed that predictions could be used more effectively if coupled with a more aggressive backfilling heuristic

(namely SPF). Yet, they only focus on manipulating the backfilling policy (replacing FCFS with SPF) and do not explore the effects of changing the main index policy (FCFS). Later works [65] showed that the main ordering policy has a more significant impact on general scheduling performance.
A problem all the aforementioned prediction-based approaches frequently suffer from is the underestimation of running times. Guo *et al.* proposed a specific framework that can be used to detect runtime underestimates [49], allowing to adjust job runtimes accordingly. They compared their approach with classical prediction schemes such as SVMs and Random Forests and showed that it also enhanced system utilization.

An interesting phenomenon is that, increasing the inaccuracy (e.g., doubling the user-provided estimates) sometimes improves performance [99]. Such surprising behavior is related to Graham's scheduling anomalies and stems from the fact that index policies generally produce suboptimal scheduling. The policy used for scheduling has a major impact on the effectiveness of accurate predictions, with policies that favor shorter jobs benefiting more. Gaussier *et al.* [42] show that, in some cases, predictions (which always have some inaccuracy) outperform their clairvoyant counterparts despite the latter's perfect knowledge of runtimes. During our experiments, we also often encountered similar situations but this remained an overall statistically insignificant effect.
In a recent study [64], the authors explored the effectiveness and limitations of using machine learning to improve the performance of computing clusters. They show that the workload is highly variable among periods, with large user churn and changes in machine utilization levels, and that a few users generate most of the workload. Consequently, model performance can vary strongly on a day-to-day basis. Moreover, more accurate runtimes do not systematically lead to better scheduling performance, and with the few datasets available today, it is difficult to assess the model's performance. Finally, they argue that training can take many months (or years) before it reaches a stable level when using a few features, which would prevent practical deployments. We also observed strong day-to-day performance variability and the potential inefficiency of static policies learned from long past periods. These observations motivate the need for a reactive online learning policy that can quickly adapt to rapid load variations.

Runtimes prediction is sometimes coupled with predicting the value of other "secondary" elements that are not *necessary* for all systems. These efforts include prediction I/O such as [95] where the authors built a neural network-based tool to predict both runtimes of jobs and their I/O requirement. Memory consumption has also received some attention In [85], focus on improving The Slurm RJMS [60] performance by increasing the accuracy of runtimes and memory consump-

tions.They compared multiple supervised machine learning algorithms including several variations of the linear regression algorithm and decision Trees.

Machine learning has been used for anomaly detection in HPC centers. This generally includes mass-collecting of data from multiple sources in the platform. Then, performing some feature selection followed by learning or a classification algorithm to pinpoint the root cause of the anomaly [91, 92, 24]. In [91] and in [92] the authors built automated frameworks that leverage data collected from fine-grained monitoring tools and time series clustering techniques to produce an accurate diagnosis.In [13] Bodik *et al.* proposed using collected data to generate signatures(or fingerprints) at different time epochs and logistic regression [52] to eliminate features that are irrelevant to the anomaly at hand. [24] presents an interference-aware scheduler for efficient co-execution of applications on GPU-based clusters and cloud servers. They use learning-based analytical models to characterize incoming applications then detect interference between them. And they use the results to guide the scheduler to minimize the interference and improve system throughput.
Another use of machine learning is application fault modeling [79]. It is done by identifying a fault signature (a set of attributes comprising of system and application state) Then machine learning is used to determine its nature and distinct characteristics.

2.2.2 Machine learning to build schedulers

In the previous approach, machine learning is used to reduce uncertainly or provide additional information that the existing scheduling scheme could use to better performance. The second approach gives machine learning more control as it aims to design scheduling schemes that directly act on the ordering and allocation of jobs. This approach is theorized to grant greater improvement of performance as it enables explore patterns and combinations that are very hard to detect via classical methods.

Carastan-Santos and Camargo [21] used synthetic workloads and simulations to create index policy functions that improve the slowdown metric using non-linear regression. Interestingly, the generated functions resemble the Smallest Area First (SAF) policy. Chung *et al.* [23] expanded this approach to include heterogeneous systems. Sant'Ana *et al.* [78] addressed the evolving nature of the workload by using machine learning techniques to select, in real-time, the best scheduling policy to apply for the next day on a given cluster, based on the current cluster and queue states. These attempts generated promising results but are rarely adopted by system

administrators as they require deploying significant changes to existing scheduling policies. Also, some strategies rely on black-box scheduling algorithms.

Reinforcement learning (RL) [61] have been receiving attention in the last few years as a holistic solution for the complete batch scheduling problem replacing both expert knowledge and established scheduling heuristics. This intuition stems from the compatibility of the RL framework to the batch scheduling problem and its success in other fields [15].
DeepRM [68] is one of the first attempts. The authors perform scheduling using the RL policy gradient algorithm and a loose adaptation of CCN [66] architecture. DeepRM was tested using synthetic data and its authors presented it not as a complete solution for the scheduling problem but as a *'good'* start for performing resources management with reinforcement learning.
In [69] the authors propose Decima, an RL based framework to schedule jobs that take the form of a DAG (Directed Acyclic Graph) in addition to basic single task jobs. The framework also handles unbounded stochastic job arrival sequences. Decima was tested on a simulation of a real spark cluster using a production workload from Alibaba. The authors showed that it outperforms classical scheduling algorithms such as Shortest processing time First (SPF), Tetris [47].
RRL [76] is a reinforcement learning-based framework specifically tailored to dynamically adjust the parallelism configuration of machine learning serving systems.
In more recent work, [97], the authors propose RLScheduler, a framework that derives adaptive policies for batch scheduling. The framework uses a kernel-based deep neural network [46] and a trajectory filtering mechanism. RLScheduler is a completely automated batch job scheduler and according to the authors, it negates the need for human expertise. It uses kernel-based neural networks and trajectory filtering. It was tested on real-world traces from the parallel workload archive [74]. It offers performance guarantees that are on par with state of the art scheduling heuristics.
RL approaches, although promising, are still in the early stages. Most of the proposed works in the literature rely on synthetic data and over-simplified assumptions for implementation and testing. To the extent of our knowledge, [46] is the closest so far to an applicable RL-based scheduler.

2.3 Energy

In recent years, power efficiency in HPC centers has attracted a great deal of attention from both the industrial and research communities. This is evident by the large number of studies that have been and are being conducted and the numerous techniques aiming to quantify and reduce the energy consumption of HPC systems.

Energy optimization has always been viewed as one of the more complex tasks in the HPC optimization Eco-system. This perception can be attributed to several reasons. (i) New hardware components appearing [88] and older components being upgraded every few years [89]. Such components although promise great improvement on both energy and performance do not have a properly defined energy profiles. Their interactions and impact on the general workload cannot be fully understood from the beginning. Thus, a period of study and analysis is always required. (ii) Energy optimization metrics are relatively new compared to more established metrics such as user sanctification and waiting time. This is shown by the lack of consensus about the metrics to use and the core components to monitor [25].

The growing interest in the energy question combined with general high activity in the HPC field resulted in a large number of works whose goal is the study and improvement of energy consumptions. Keeping track of all these works and their current standing is not a trivial task, Thus many researchers performed surveys to organize and structure energy works by proposing new taxonomies and classifications or to extend and include more recent advancements in the field [25, 8, 48, 84].
In [8], the authors propose a survey about energy monitoring in large-scale systems. They focus on solutions that are used for online power measurement. They point out and discuss several of the flows and weaknesses of current monitoring methodologies. They argue the need for high quality -high precision measurement of large systems. They also invoke several older extrapolation studies and conclude that such studies tend to be inaccurate. This point of view is not unique to them as many observers advise taking caution when using extrapolation studies. Prediction about future energy behavior have a notorious reputation of being widely different from reality [@57]. A recent survey [25] discusses state of the art APIs to control energy and power management. Their main conclusion (among other conclusions) is that there is a need for unification in the energy monitoring and evaluation processes. They advocate focusing efforts on determining what they refer to as a cardinal common subset of universal parameters related to power and energy. They also suggest building performance-energy models for a wider range of CPUs and GPUs architectures for various classes of applications.

2.3.1 Estimating energy consumption via models

A sizable portion of the studies focuses on building energy estimation models rather than direct energy measurement. This is mainly because the latter requires tools such as watt meters or power sensors which requires a significant software and hardware investment to put in place [45].
Performance monitoring counters (PMC) [73] are one of the cornerstones of building energy models. Such model estimates power consumptions of architectural units

based on their activity factors such as integer operations, floating-point operations, memory requests due to cache misses, etc [10]. [59] and [94] propose system-wide power prediction models for HPC servers based on PMCs. applications into groups and create specialized power models for them.
For more detailed information about energy models, we refer the interested reader to the following surveys: (i) [73] studies several models and extract common features between them, and proposes a classification based on the dominant components. (ii) [39] A survey that focuses on modeling energy consumption of machine learning algorithms.

2.3.2 Estimating energy consumption via Measurements

Measurement-based studies are generally shown to be more accurate and reliable. However, they require a certain level of instrumentation and involvement (hardware and software) to use.
Power meters are a popular tool to measure power consumption. However, they only provide the total power consumption of the whole system. Most devices are not instrumented for component-level power measurement and adding the instrument and the suitable interfaces to provide such reading usually involves intrusive modifications to the system that are not trivial to put in place and they often cause a power distribution network.
The PowerPack framework [43] proposes a combination of instrumentation and models. It couples power with PMC models [73]. PowerPack is a power profiling tool that uses power meters for general machine consumption and DC power data devices to gather component-level power consumptions. The main drawback of such too is that it requires a large number of power measurement instruments, making it difficult to collect data and conduct large, multi-cluster scale comparative studies.

Another option that has been gaining popularity in the last few years is Intel's Running Average Power Limit (RAPL) interface [58]. RAPL was introduced in Intel's Sandy Bridge architecture and has been included in later versions, evolving and improving with every iteration. RAPL is a model-based interface that doubles as an energy control and monitoring tool. It provides power limiting features and accurate energy readings for CPUs and DRAM which are easily accessible [98].
The growing popularity of RAPL can be attributed to several reasons. RAPL causes minimal interference to the regular operations of data centers, it is easily executable because it does not require any external sensors or energy meters to be mounted with the system. Also, RAPL seamlessly expose the energy reading and power states to the OS. And the measurement it provides can be gathered by a wide variety of tools, including the Linux *perf_event* interface [28]. This allows for unprecedented

easy access to energy information when designing and optimizing energy-aware code.

Many studies have been conducted to test and validate the accuracy of the reading provided by RAPL. Most of the research conducted on RAPL's quality of measurement reported that its readings are sufficiently accurate (highly correlated with plug power). Dongarra et al. [30] studied energy consumptions of high-performance dense linear algebra libraries LAPACK and PLASMA using power pack and Intel Running Average Power Limit (RAPL) API. They concluded that RAPL API is a good alternative to the more intrusive and expensive power meters based on near-identical power measurements observed between PowerPack and RAPL. In [50], the authors provide an overview of different power measuring techniques including RAPL and APM (AMD's energy measurement interface). They concluded that although RAPL's energy measurements are fairly accurate, its potential is severely hundred by the fact that RAPL provides energy (and not power) consumption data without timestamps associated with each counter update. For the aforementioned reasons, we decided to rely on RAPL for energy measurement in this thesis.

3 Problem Setting

In this chapter, we introduce the various elements we used throughout this work. In Section 3.1., We explain the backfilling algorithm and one of its most popular variants (EASY), the ordering policies, and the phenomenon of starvation. The objective functions are defined in Section 3.2. Finally, We present the experimental setup in Section 3.3; we detail the data used, the simulations used, and the general workflow.
Chapters 5, 4, and 6 all share the elements that will be introduced in this chapter. Any difference or specificity will be detailed in its respective chapter.

3.1 Preliminary Definitions

3.1.1 Jobs

A job in the scheduling world, depending on the goal and the available information, can have many definitions.
This work is based on real HPC logs. Mainly extracted from the parallel workload archives [74]. Our goal is to propose general methods to improve the performance of any given HPC platform. Thus we take the most general definition of a job. We limit the used information to include only the elements that are common between the jobs in all HPC platforms.

We consider an online scheduling model, where the jobs arrive at different times unknown in advance. Figure 3.1 depicts the shape of a job. The information available upon arrival are:

- p: The runtimes of the job, also called the processing time. It represents the exact execution time of the job. It is only known after the job finishes its execution. Thus when making a scheduling decision it is replaced by $\tilde{p}$
- $\tilde{p}$: The requested processing time, also called the estimated processing time. An estimation/upper limit of the processing time given by the user.
- q: The requested resources, also called number of requested processors , the requested, set by the user.

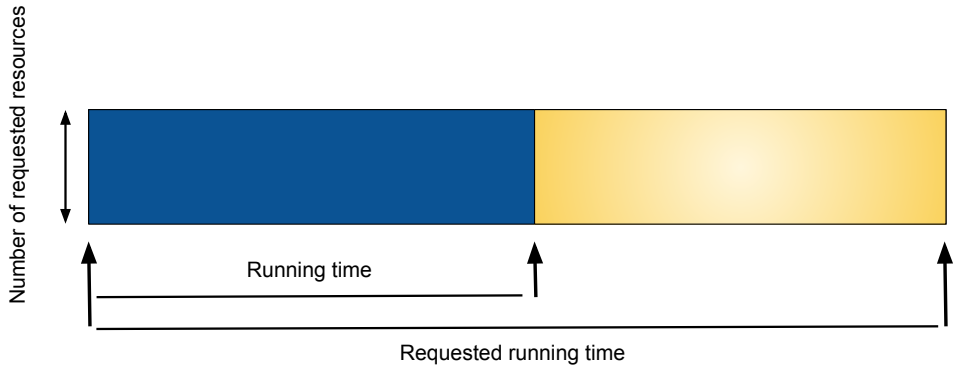


Fig. 3.1: A job is defined by three elements. the requested number of resources, and the requested running time, and the actual running time of the job.

- $submission\_time$: the arrival time of the job.

The scheduler chooses one or more of the waiting jobs to execute at each time-step. Jobs cannot be preempted and Internal Information such as intra-job communication, level of parallelism are not taken into consideration. Jobs are considered as independent from each other and no precedence constraint are taken into account.

3.1.2 Backfilling

At its core, the task of scheduling is simply the selection of the order in which jobs will be executed. One of the most popular techniques used to perform this task is the Backfilling algorithm. Backfilling works by finding ”holes” in the scheduling table and moving smaller jobs forward to fill these holes. Backfilling has the benefit of reducing fragmentation and improving throughput.

Two particularly known backfilling heuristics for HPC platforms are EASY [79] and conservative [70]. In conservative backfilling, each job is given a reservation upon its arrival. Jobs can move ahead of the queue via the backfilling mechanism as long as they don’t delay any of the reservations. When a new job arrives, all the reservations are revised and modified if necessary. With EASY backfilling, the reservation is only made for the job at the head of the queue. While conservative offers many advantages, it introduces significant overhead and limits opportunities for backfilling [81]. That could explain why EASY is still the most popular of the two as many machines in TOP500 [1] rely on some variation of it. In [82], the authors offer a detailed comparison between EASY and Conservative backfilling.
Since most of the works we will present in the subsequent chapters are based on the

EASY algorithm, we provide a detailed description of how it works in the following Section.

3.1.3 EASY-Backfilling

EASY-backfilling or EASY [79] is a scheduling algorithm that uses a queue to select and backfill jobs. The jobs are usually ordered and backfilled using an index policy which is an an ordering rule based on a certain job characteristic. e.g FCFS as it name entail used the arrival time to sort the jobs. Index polices are explained in detail in Section 3.1.4.

Algorithm 1 recalls how EASY works. At any time a scheduling decision is required (*i.e.* job submission or termination), the scheduler goes through the job queue in a *primary order* and starts them until it encounters a job that cannot be started immediately. At this point, the scheduler makes a reservation for this particular job which ensures that it will not be delayed from its initial position. Then, it goes through the rest of the job queue in a *backfilling order* and execute any jobs as long as it does not delay the unique reservation mentioned earlier. This is known as backfilling. One of the most popular variations of the EASY algorithm is EASY-FCFS-FCFS where the jobs are ordered and backfilled by their arrival time.

Algorithm 1: EASY Algorithm

```
   Data: Queue Q of waiting jobs sorted by increasing submission times.
   Order primary queue according to an index policy
 1 for job j in Q do
 2     if j can be started then
 3         Start j
 4         Remove j from Q
 5     else
 6         Reserve j at the earliest possbile time according to the estimated
             running times of of the currently running jobs
 7         break
   Backfill according to the selected policy
 8 L → Q -[reserved jobs]
 9 Order L according to selected policy
10 while L not empty do
11     Start all the jobs that can be backfilled without delying the reservation
         from Q
```

3.1.4 Scheduling policies

In Section 3.1.1 we stated that a job has 4 characteristics. The arrival time, the number of requested resources, the running time and the requested running time.

The running time can only be known after the job finishes the execution. Thus it cannot be used to help schedule jobs in the waiting queue. Thus, we rely on the three remaining pieces of information. A job characteristic can be one of the aforementioned three. We can also combine them in various ways to generate new characteristics that express different, more elaborate types of information. We use the following job characteristics during the experimental campaign:

- q_j: (requested resources) the number of processors the user requested.
- $\tilde{p}_j$: (requested/estimated processing time) the estimated processing time provided by the user, it also serves as an upper limit to the time the job is allowed to run. The actual processing time p_j can only be obtained after the execution of the job.
- $wait_j$: (waiting time) How long a job j spent in the waiting queue:
  $wait_j = current\_time - submission\_time_j$
- ρ_j: (estimated ratio) $\frac{\tilde{p}_j}{q_j}$.
- a_j: (estimated area j) $\tilde{p}_j q_j$. The total processing power a job is estimated to use during the execution
- exp_j: (estimated expansion Factor) $\frac{wait_j + \tilde{p}_j}{\tilde{p}_j}$: the ratio of the total time a job is expected to stay in the system (waiting time plus estimated processing time) normalized by its estimated processing time. This characteristic is rather special since it reflects the *estimated* value of the objective function. Note that the *BSLD* could have been used instead of the expansion Factor but it makes very little to no difference in term of ordering since only the smaller jobs (which usually have the least impact on performance) are marginally concerned.
  Scheduling using exp_j is expected to be a good or at least an important strategy. But it is unknown how it will perform at this point since it does not account for q_j.

With each of the six aforementioned job characteristics, we construct two scheduling policies: one that prioritizes the lowest score given by the characteristic and another the highest. So we have the following 12 ***pure*** policies:

- FCFS: First Come First Served
- LCFS: Last Come First Served
- SPF: Smallest estimated Processing time First
- LPF: Longest estimated Processing time First
- SQF: Smallest Resource Requirement First
- LQF: Largest Resource Requirement First

- SAF: Smallest estimated Area First
- LAF: Largest estimated Area First
- LEXP: Largest estimated Expansion Factor First
- SEXP: Smallest estimated Expansion Factor First
- LRF: Largest estimated Ratio First
- SRF: Smallest estimated Ratio First

3.1.5 Starvation

Among all the scheduling policies presented in Section 3.1.4, FCFS is the most popular. A major reason for that is EASY-FCFS's natural ability to prevent starvation. Starvation occurs when a job is denied the resources necessary for its execution for an unbounded period of time. In the FCFS case, the job will wait for a maximum equal to the time necessary to finish all the jobs that arrived before it.

The definition of starvation can be extended to include long periods of waiting. The exact time a job wait before it is considered a starving job is subjective as it depends on the size of jobs and the tolerance of the user (how much they are willing to wait) and the amount of the workload and many other factors.
Starvation is a concern for any policy that does not take the waiting time of the job into consideration. For example, under the SPF order, a job will wait until all the smaller jobs are executed, including the ones that arrived after it.

In this work, we mitigate the starvation problem by deploying a simple but effective thresholding mechanism. A threshold can be defined as the maximum time a job can wait before it is considered a starving job. The jobs that exceed the threshold value are moved to the head of the queue regardless of the ordering policy in play and are executed in FCFS order.

This mechanism can be seen as a re-introduction to the FCFS principle to other scheduling policies. By coupling non FCFS scheduling policies (e.g, SPF) with the thresholding mechanism, The scheduler guarantees that no job will wait beyond a certain amount of time. Moreover, by manipulating the value of the threshold the system administrator has the ability to flexibly create hybrid algorithms that are a compromise between any scheduling policy he chooses and FCFS. In [65], the authors perform a detailed study of the thresholding mechanism and the impact of different values. Thresholding offers the option of using more greedy or aggressive ordering policies while limiting the risk of starvation.

3.2 Objective functions

There is a large number of objective functions (also know as scheduling metrics or cost metrics) [36] – which focus on different performance aspects of the scheduling – that can be used by HPC platform administrators. In this regard, we focus on three platform-wise, job-oriented metrics. The first metric is the *waiting time* (Equation 3.1) which, measures the time that the job waited for execution, and it can be defined for a job j as:

$$wait_j = start_j - r_j \tag{3.1}$$

where $start_j$ is the time that j started its execution. The second metric is the *slowdown*(sld) or *stretch* which, measures the ratio between the time that a job j spent on the platform $wait_j$, and the actual processing time $p_j$of j. It was designed to be a metric that "*remedies the* waiting time*'s favoritism toward longer jobs*" [36].

Formally, the *slowdown* can be defined as follows for a job j:

$$sld_j = \max\left(\frac{wait_j + p_j}{p_j}, 1\right) \tag{3.2}$$

The reasoning behind slowdown is based on the expectation that the waiting time of a job should be proportional to its processing time, thus giving a balanced waiting time distribution among jobs with different characteristics, notably the processing time p_j. One major shortcoming of the slowdown is that it heavily punishes very short jobs with a reasonable waiting time. For that reason, the slowdown is almost never used in its basic form, instead, most systems use the *bounded slowdown* (Equation 3.3):

$$bsld_j = \max\left(\frac{wait_j + p_j}{\max(p_j, \tau)}, 1\right) \tag{3.3}$$

where τ is a constant to prevent smaller jobs from reaching very high bsld values, and it is often set to 10 seconds. In the remainder of this thesis, we use the term *slowdown* and *bounded slowdown* the refer to the *bounded slowdown*

Finally, the fourth metric is the *per-processor bounded slowdown* [100] (pp-bsld or pp-slowdown, Equation 3.4), which is defined for a job j as:

$$\textit{pp-bsld}_j = \max\left(\frac{wait_j + p_j}{q_j \cdot \max(p_j, \tau)}, 1\right) \tag{3.4}$$

where w_j and τ are the same as for bsld. The reasoning behind the per-processor bounded slowdown is to normalize the slowdown results for jobs that perform the same amount of work, though with different degrees of parallelism (number of processors). The pp-slowdown can be seen as a more appropriate objective for the parallel batch scheduling problem, as it tries to balance the waiting time of the jobs in function of the number of processors q_j, which is not taken into account by the waiting time and the slowdown.

Fairness and user satisfaction: There are arguments for and against all the aforementioned metrics (and every other metric for that matter). In fact, there is no single metric or number that can objectively determine the superiority of a scheduling policy compared to others. Thus we should identify the optimization goal we want to achieve and choose the objective function accordingly. For slowdown and pp-slowdown, the expectation of a good scheduling performance is that the waiting time of the jobs should be proportional to its running time, that is, a job that must perform a larger amount of work (and thus requires many resources and/or for a longer period) could "afford" a longer waiting time. Indeed, it is arguable that the slowdown metric can be a good performance metric for a job-centric fairness, in comparison to other metrics such as waiting time. One could envision, however, that a better performance metric could be a user-centric metric, that captures the overall satisfaction among users. Although this could be indeed the case, one can not simply simulate user behavior by reproducing a workload trace due to the fact that the workload would change (in an on-line manner) in function of the scheduler's performance (e.g. a more efficient scheduler would stimulate users to submit more jobs and vice versa). Although some effort has been performed to propose solutions in this regard [35], at the time of writing of this paper, there is no consensus in the community about accurate and/or meaningful ways to simulate user behavior, which leads us to choose a job-centric approach rather than a user-centric one.

3.3 Experimental framework

3.3.1 Simulations

The most reliable method to study the behavior of a scheduling policy is to apply it directly to a real-world system. However, this method is rarely used. (i) It takes a long time to observe the full effects and the tested algorithms or techniques. (ii) HPC management is usually a subject to many constraints like maintaining user satisfaction, maximizing machine utilization, and controlling energy consumption and any change other than the established practices are considered very risky. Thus most people opt for simulations.

Performing a realistic simulation is not a trivial task. One must take into account the numerous components in play like the type and the state of the CPU, memory, and network, the current load of the machine, and its impact on the performance to name a few. Any of these factors can have a decisive impact on the performance of the platform which in turn directly impacts the time it takes to execute the jobs. For that reason, a lot of effort has been directed toward simulating an HPC environment or at least a specific aspect of it [63, 72, 32, 54, 7].

In the context of our work choosing a simulator is a question of a trade-off between accuracy and speed. The experiments presented in chapter 5 focus on generating and testing various scheduling combinations with the EASY-backfilling algorithm. It is a series of machine learning tasks that require a substantial number of simulations. For this reason, we decided to fully favor performance over precision. We discard all topological information related to the platforms that generated the traces. We consider all processors to be indistinguishable from each other and the cost of communication non-existent. We use a lightweight simulator that can replay the EASY-backfilling scheduling process at very high speeds. Thus, we replace the RJMS with a EASY-backfilling lightweight simulator[1]. The experiments in chapters 4 and 6 are not as computationally demanding as in Chapter 5. Instead, they require real-time modification and manipulation of EASY. So we opted for using Batsim [31], an HPC simulator based on SimGrid [22]. Bastim allows us to accurately simulate the platform and the scheduling process of many workloads within a reasonable amount of time.

[1] `https://gitlab.inria.fr/szrigui/mixed-policies`

3.3.2 Data

3.3.2.1 User scheduling data

One of the main objectives of this thesis is to propose practical, grounded approaches to understand and improve the quality of RJMS schedulers. We build our solutions from observations of actual scheduling events. For this reason, we choose real-world traces (from the parallel workload archives [74]) instead of artificially generated data. Table 3.1 outlines the workload used throughout the experimental campaign. These particular traces were chosen for the following reasons. (i) They come from different environments. (ii) They have a high resources utilization, which is required to properly test scheduling algorithms in as many settings as possible. (iii) They have been used in previous works [90, 21, 42], which provides a valuable reference point and allows us to compare our approach with similar approaches from the literature.

Trace	#CPU (#nodes*node_size)	#Duration	#Jobs	Average job duration
KTH-SP2	100 (100*1)	11 Months	27670	8579 (s)
CTC-SP2	338 (338*1)	11 Months	68687	9807 (s)
SDSC-SP2	128 (128*1)	24 Months	49809	6318 (s)
SDSC-BLUE	1,152 (144*8)	32 Months	208716	3184 (s)
HPC2N	240 (240*1)	42 months	2020,871	??(s)
Metacentrum-zegox	576(576*2)	24 months	79,546	??(s)

Tab. 3.1: Workloads

4

An in-depth study of simple scheduling policies: Performance and evaluation metrics [1]

4.1 Introduction

Throughout the history of HPC optimization, a vast number of queue ordering policies have been conceived, from hand-engineered [87] to tuned or machine-learned [42, 21, 65, 41] policies.
Despite this impressive number of works, it is well known [37] that most RJMSs still deploy the First-Come-First-Served (FCFS) policy with some backfilling mechanism [70] (EASY-FCFS), and optionally with an arbitrary job prioritization, represented by multi-queue priorities [77].

Many reasons can be devised to justify the choice of EASY-FCFS: it is established that EASY-FCFS increases the overall utilization of the platform, while keeping a relative simplicity and job starvation guarantees. Furthermore, although it is also established that there is room for improvement in the scheduling, replacing EASY Backfilling with another algorithm might be seen as a risky change: one can see this change as a "jump into the dark", with the changes in performance only noticeable after a long period of time, and potentially after many strong-worded emails from many (important) users. This work goes towards bringing light to this jump. We selected a class of scheduling algorithms that keep the same simplicity and starvation guarantees of EASY Backfilling and we used a fast and reliable HPC simulation software to provide sound evidence on what could be gained – considering many relevant performance metrics – if one replaces EASY Backfilling. More specifically, in this chapter:

- We present an experimental study that addresses the expectations and potential gains that come from replacing the EASY-FCFS scheduling policy in typical high-performance computing platforms;

[1] The text of this chapter is adapted from the following published paper:Danilo Carastan-Santos, Raphael de Camargo, Denis Trystram, **Salah Zrigui**. One can only gain by replacing EASY Backfilling: A simple scheduling policies case study. CCGrid 2019 - International Symposium in Cluster, Cloud, and Grid Computing, May 2019,

- We highlight the Shortest Area First (SAF) scheduling policy, which, we argue, has the best-observed overall performance among the tested policies. In fact, we propose SAF as a new benchmark for future batch scheduling studies;
- We highlight an aspect that is often overlooked when evaluating the performance of a scheduling policy, which is the link between the number of resources used by jobs and the fairness of a given scheduling policy;
- We address the influence of the aggressive backfilling mechanism on the transparency and predictability of scheduling algorithms.

The remainder of this chapter is organized as follows. Section 4.2 details the experimental protocol we adopted for this chapter. In Section 4.3 we present and discuss the obtained experimental results. Finally, we summarize the main conclusions of the paper and present future works in Section 4.4.

4.2 Experimental protocol

The experimental campaign we present in this chapter relies on the elements introduced in Chapter 3. We use 5 of the of traces presented in 3.1; KTH-SP2, SDSC-SP2,CTS-SP2, SDSC-BLUE, and HPC2N.
We focus our analysis on the policies representing one of the fundamental aspects of any job: The arrival time, the running time, the number of resources, and the total area. We also select the policies that prioritize shorter/smaller jobs since they tend to largely outperform their longer/larger counter parts [41]. We use FCFS, SPF, SQF, and SAF (Section 3.1.4).
We make use of BatSim [31] (Section 3.3.1), which allows us to rapidly and accurately simulate the scheduling of many workload traces with using only a single workstation and in only a matter of days, which would not be feasible without simulation.

In order to provide statistically meaningful results with the scheduling of the traces, we adopted a sampling technique based on [35]. Algorithm 2 presents the pseudo-code. The idea is to generate new data using existing user profiles. A profile can be defined as the activity of a single user throughout the trace, split into many weekly time periods. To generate a new trace we combine several random permutations of each user's profiles. One can observe that this sampling technique is not capable of reflecting the workload changes in function of the scheduler's performance (as discussed in Section 3.2). However, it allows to generate as many logs as needed while preserving the jobs' properties of each user.

Algorithm 2: Workload trace resampling algorithm.

Data: List of user $profiles$ P extracted from the original workload trace.
Number of weeks in the resampled trace n_w.
Result: Resampled trace W_{res}

```
W_res ← ∅
for i = 1 to n_w do
    w_res ← ∅
    foreach user profile p in P do
        p_{i_res} ← random weekly split from p
        add p_{i_res} to w_res
    append w_res in W_res
return W_res
```

For each trace, we generate 10 samples using the aforementioned procedure. The size of each sample is proportional to the size of the original trace. Each sample is then simulated following the EASY scheduling algorithm, taking into consideration each of the four chosen scheduling policies . The results for each scheduling policy and workload trace presented in the next Section are statistical summaries of the ten samples of each trace.

4.3 Experimental Results

In this section we present the main results obtained by the experimental procedure described in Section 4.2. We perform several analysis in order to provide a better understanding of the behavior of the scheduling policies and what gains could be expected if a certain scheduling policy is chosen.

4.3.1 Overall Scheduling Performance

Figure 4.1 shows the overall performance results for the average slowdown, waiting time, and pp-slowdown. Each subplot refers to a workload trace from Table 3.1. To avoid outlier interference in the results, for each trace and scheduling policy we discarded the best performing and the worst performing workload sample (see Section 4.2) from the 10 initial workload samples. In other words, we present only the scheduling results of the samples whose performance belongs to the 10-90% percentile range. Each subplot contains statistics of the scheduling simulation of these remaining samples. The solid lines in the subplots represent the cumulative mean of the objective metric (average slowdown, waiting time, or pp-slowdown) of the finished jobs at each week of simulation, from the beginning to the end of the workload, and the dashed lines represent the cumulative maximum and minimum average values of the respective metric at each week.

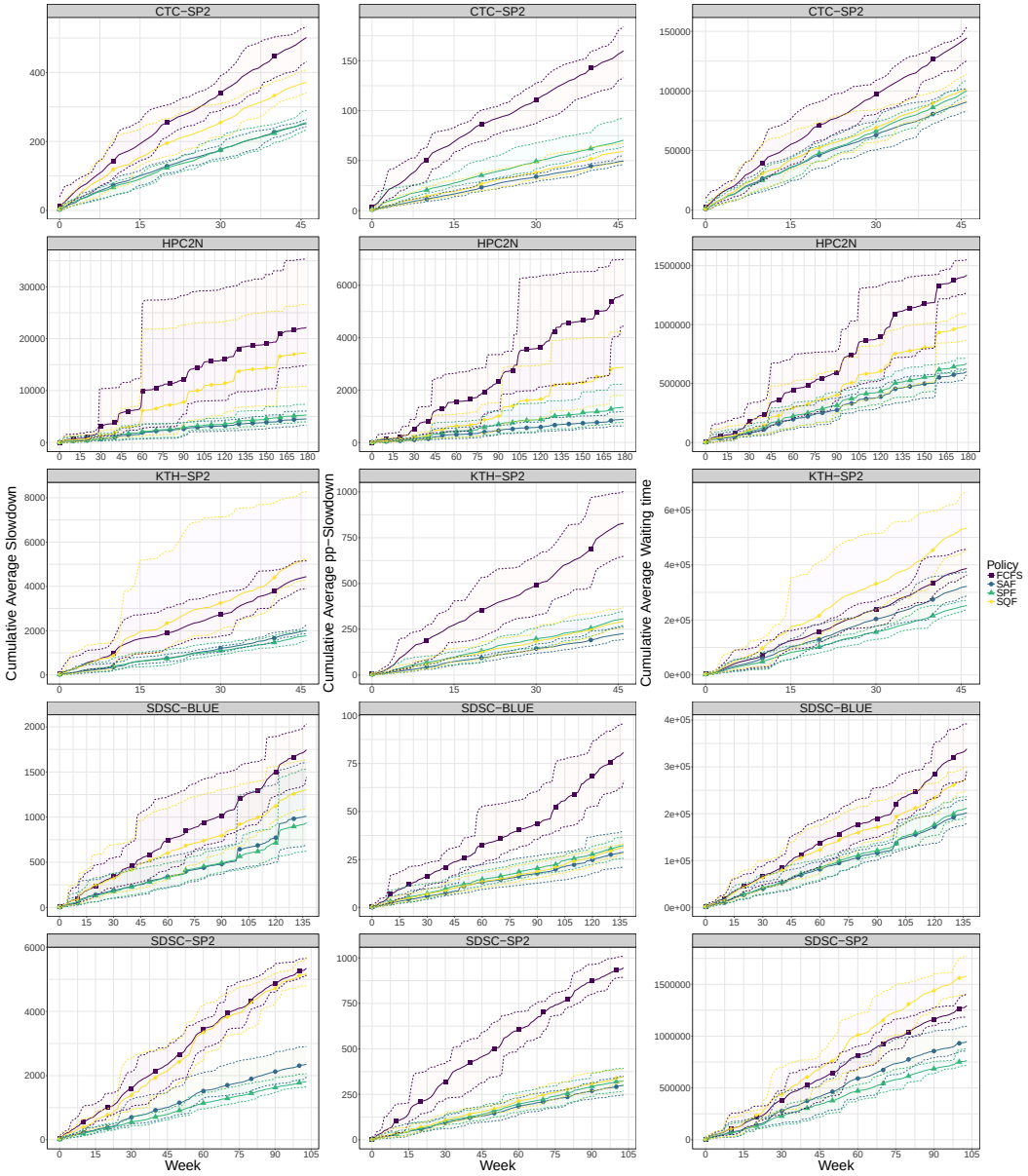


Fig. 4.1: Cumulative weekly average slowdown, pp-slowdown and waiting time: For each trace, the middle solid line represents the mean and the two dashed lines represent the lower and upper 10-90 percentiles.

Looking at the scheduling performance in Figure 4.1, we can cluster the tested policies in two classes: the ones that are oblivious of the processing time estimate $\tilde{p}$ (FCFS and SQF), and the ones that are not oblivious (SPF and SAF). From the aforementioned Figure, we can observe a strong correlation between the scheduling performance of these clusters, with the former cluster consistently presenting worst performances than the latter. This result is expected: for the slowdown and pp-slowdown, jobs with a lower $\tilde{p}$ – and thus lower p, since $\tilde{p}$ is an upper bound of p – have a higher risk of inflating the metrics if they wait too much (see Equations 3.3 and 3.4). By favoring jobs with a lower $\tilde{p}$ (SPF and SAF), we assure that these high risk jobs are executed quickly, and thus the average for both slowdown and

pp-slowdown are kept under control. The waiting time is also favored by prioritizing jobs with lower $\tilde{p}$, since for all traces these jobs are more frequent [74].

One point that is worth noticing is how much can be gained in quantitative values if a policy other than FCFS (notably SPF or SAF) is chosen and kept during a long period. In our experiments we achieve performance gains up to 83.4% (SPF), 61.4% (SAF), and 85.1% (SAF) for the slowdown, waiting time, and pp-slowdown respectively, in comparison with FCFS. It is important to note here that the scheduling simulation is performed with a starvation prevention mechanism. Therefore, these gains can be obtained while guaranteeing that no job will starve.

Another important observation is how SAF – which in contrast with SPF, is less known in the literature – performs consistently well in all objectives considered. We further address this phenomenon in the next Section.

4.3.2 Is SAF the ultimate simple policy?

As highlighted in the previous section, the scheduling policies that are not oblivious to the processing time estimate $\tilde{p}$ (notably SPF and SAF) are the ones who achieved the most consistent good performances in the experiments that we performed. In this Section we make a further analysis on which are the characteristics of the jobs that make them prioritized/delayed by these two policies, with an emphasis on the delayed jobs.

For the processing time estimate $\tilde{p}$ this analysis can be easily devised: SPF delays jobs with a larger $\tilde{p}$ and SAF is similar, with the distinction that it considers the number of processors q as well. This raises the importance of our thresholding mechanism, which specifically concerns jobs with a large $\tilde{p}$.

In its turn, for the number of processors q, Figure 4.2 shows the number of processors q of the top 100 jobs – of each sample of each trace – who got delayed the most (here defined as the jobs with the highest slowdown) for each scheduling policy. An interesting observation here is that SPF is oblivious to the number of processors q and thus no correlation should be expected for the delayed jobs in function of q. Therefore, SPF had a high risks of delaying jobs with smaller q which, in principle, should be easier to be scheduled in an HPC platform.

Indeed, we recall a known observation [36] that the slowdown and the waiting time metrics (arguably the most popular ones) do not take into consideration one important dimension of the scheduling problem: the number of requested processors q. Jobs that perform the same amount of work though with different shapes are

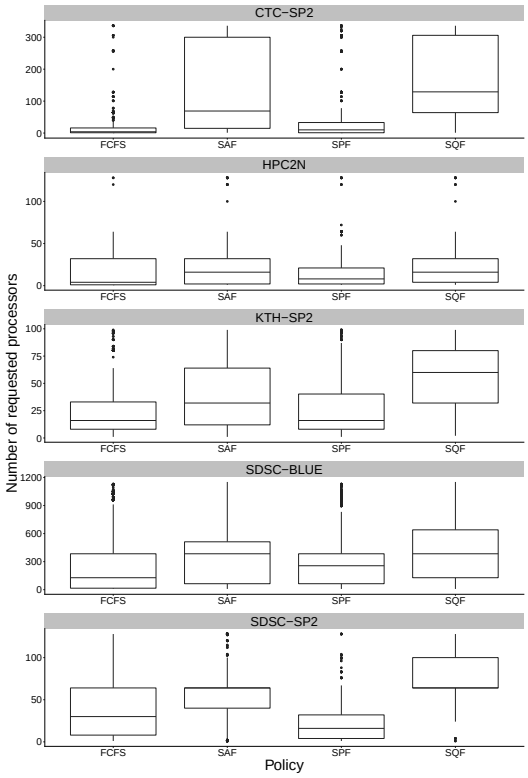


Fig. 4.2: Number of processors of the top 100 jobs with highest slowdown values.

treated indifferently by these metrics. The pp-slowdown generalizes the standard slowdown by including the number of processors q in the metric.

At this light SAF shows up as a solid policy among the simple ones we evaluated. It achieved close to best observed performances for the slowdown and waiting time objectives, and systematically outperformed all other simple policies for the pp-slowdown objective (Figure 4.1) . This complies with the results of our previous work [21], where the machine learned policies converged to functions that contain a SAF-like component. Although one can claim that SAF could be biased towards pp-slowdown, since with pp-slowdown we would seek to minimize an objective function that is related to the area of the jobs, we argue that the pp-slowdown is a more appropriate objective for the parallel batch scheduling problem, in comparison with waiting time or slowdown.

4.3.3 Accounting the Maximum: one should care with caution

One can notice that in this work we only seek to find good scheduling algorithms aiming at the average of the objective functions and not the maximum. Although one can argue that the maximum of the objective functions are important as well, in

this Section we present some observations found by our study that show that aiming only for the maximum can be potentially problematic.

The first point is that the maximum metric is centered at the performance of only one job, meaning that the value of the maximum can be unstable and subject to unpredictable factors, such as unavoidable bursts of jobs submissions and/or jobs that have some characteristic that can potentially mistakenly inflate the metric. To illustrate this potential, we clustered the jobs into two classes: the premature jobs, in which the difference between the processing time estimate $\tilde{p}$ and the actual processing time p is at least 100 times higher, and the standard jobs, which are the remaining jobs. Table 4.1 shows the percentage of premature jobs found for each workload trace. What is interesting to observe is that the number of premature jobs is not negligible, up to one third of all of the jobs of the trace. Furthermore, the difference between $\tilde{p}$ and p can be sometimes quite extreme: jobs that are marked as successful jobs (i.e. job that did not crash) and require the maximum processing time allowed $\tilde{p}$, though actually execute for around one minute happen in every trace. Since these jobs are marked as successful, we can not discard them from the analysis.

As a consequence, any scheduling policy that prioritizes jobs in function of the processing time estimate $\tilde{p}$ risks delaying these premature jobs and, when evaluating the objective function of these jobs, they will obtain poor results which will harm the maximum of the objective function. To illustrate this effect, Table 4.2 shows the ratio between the average slowdown of the premature jobs and the average slowdown of the standard ones, for all traces and scheduling policies. We can notice that the difference in scheduling performance of these two classes of jobs is large, up to 17 times larger for all policies in the HPC2N trace, and this difference in the maximum slowdown between these two classes (result not shown in Table 4.2) is even larger. We can also notice that this difference is often amplified by policies that takes $\tilde{p}$ into account (SPF and SAF).

Agreeing whether or not these performance gaps are due to the scheduler is always up to argument. However, Figure 4.3 shows a more holistic view of the scheduling performance: we grouped the jobs in many categories that are in function of the jobs' scheduling performance, from the jobs that were executed immediately (slowdown of 1), to the jobs that were poorly scheduled (slowdown of at least 100). We can observe that choosing another policy than FCFS shows performance improvements in all categories: the number of jobs who got executed immediately increases and the number of jobs in all other categories (the jobs who had to wait) decreases, with an exception of the SPF policy at the 1-10 slowdown range. These results are even more impressive for the category of jobs with poorer scheduling performances (100+ slowdown). For instance, by choosing SAF, the number of jobs who got badly

scheduled can be lowered by more than half, up to 2.8x less poorly scheduled jobs in comparison with FCFS.

All of these points elucidate the importance of analyzing the scheduling performance in a holistic view, and the caution that must be taken into account when evaluating the scheduling performance with maximum values. We would certainly overlook these good properties of the studied scheduling policies if we had considered only the maximum of the objective functions.

Tab. 4.1: Percentage of premature jobs for each workload trace

Trace	% of premature jobs
HPC2N	17.4
SDSC Blue	30.2
SDSC-SP2	16.1
CTC-SP2	9.5
KTH-SP2	12.4

4.3.4 Backfilling Influence

One important question that rises when the queue ordering policy is changed; is how the backfilling mechanism behaves in function of the queue ordering policy. Although it is well known that backfilling increases the platform's utilization and is unlikely to harm the original (without backfilling) schedule, its relevance to performance is not clear. This question is also worth of importance to bring a clearer notion about the predictability of the scheduling policies, that is, given one policy, how much it is likely that the jobs will actually follow such an order.

In order to clarify this point, for all samples of each trace and each scheduling policy we kept track on how many jobs got scheduled to execution by the backfilling mechanism. Figure 4.4 shows the distribution of the number of backfilled jobs over all samples, for each workload trace and scheduling policy. One interesting observation is the absence of backfilled jobs for the SQF policy for every trace and sample. This result is expected and we formalize it with the following proposition:

Proposition 1. *If the aggressive backfilling algorithm uses a queue of jobs sorted by SQF and there is no threshold mechanism added to the scheduling, no job is backfilled.*

Proof. Scheduling decisions are performed in two cases:

1. When a job arrives in the queue: in this case, let t_h be the job with the highest priority in the queue. Job t_h is in the queue, therefore there is not enough resources to process t_h. Since the queue is sorted by SQF order, there is no

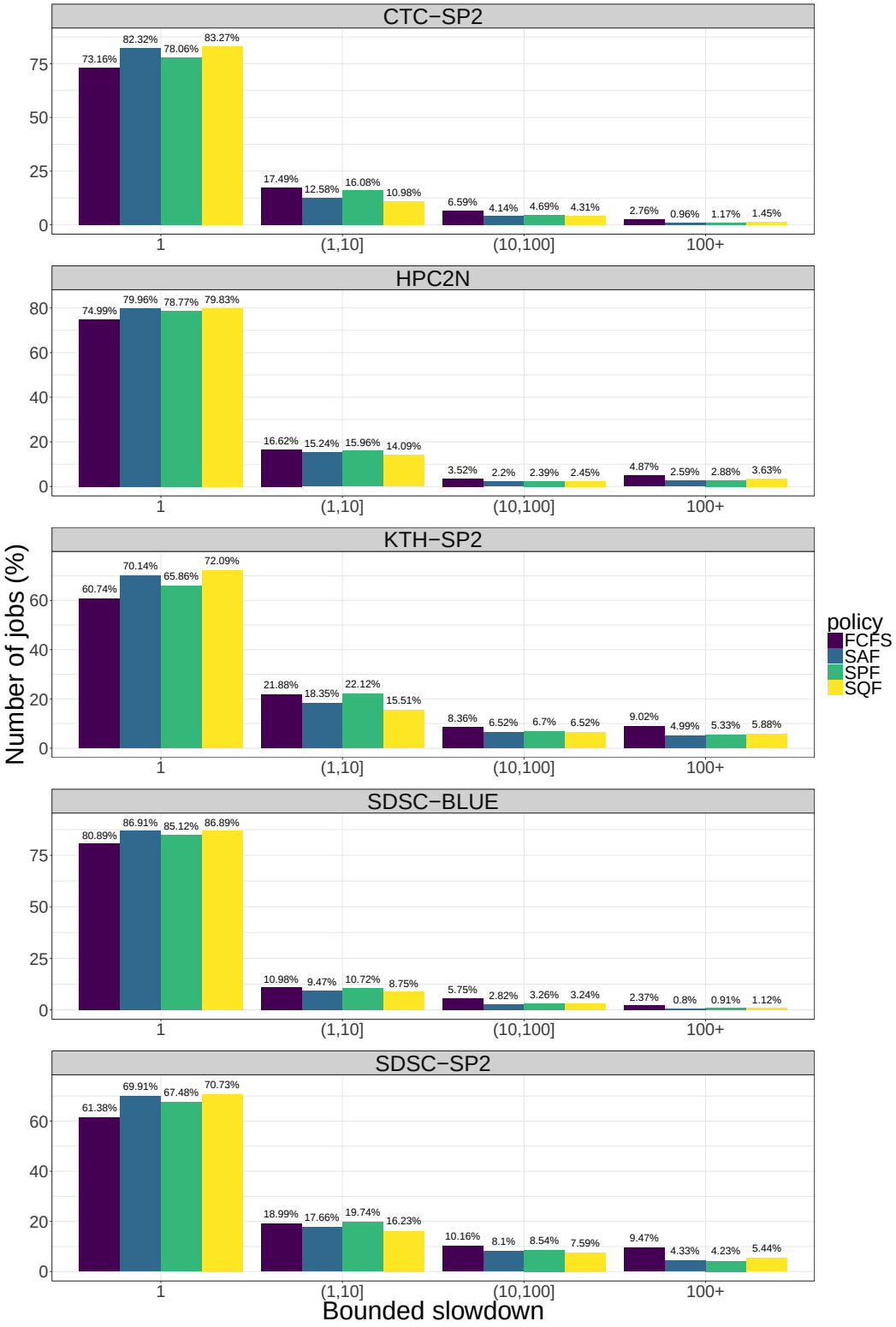


Fig. 4.3: Distribution of the bounded slowdown values for all jobs

job in the queue that requires less resources than t_h, so none of them can be backfilled. If a new job t arrives in the queue and its number of required processors is lower than the number of processors required by t_h, SQF will

Tab. 4.2: Ratio of the average slowdown between the premature the standard jobs

Policy	HPC2N	SDSC Blue	SDSC SP2	CTC SP2	KTH SP2
FCFS	17.84	3.59	3.94	5.58	8.69
SPF	17.29	7.17	4.36	5.04	12.09
SQF	14.02	2.96	2.04	1.67	9.31
SAF	17.88	7.41	3.79	2.61	11.41

assign t with the highest priority and thus backfilling will no longer be applied for t. Conversely, where t requires more processors than t_h, t cannot be backfilled as aforementioned.

2. When a job is finished and its allocated resources are released: in this case, the jobs will be scheduled for execution following SQF order until it is no longer possible to schedule jobs with the current available resources. At this point, there are not enough resources to schedule the job with the highest priority in the queue and, since the queue is sorted in SQF order, no other job in the queue can be backfilled as aforementioned.

Since in both of the above cases it is impossible to backfill jobs, no jobs are backfilled. □

Yet, some backfilling may happen when using SQF with jobs that exceeded the threshold in the waiting queue (since they break the SQF order). However, such jobs are expected to be very few. This explains some results found by Lelong *et al.* [65], in which they state that the SQF policy did not lead to many backfilling decisions in their experiments.

Interestingly, using SAF and SPF resulted in 78% and 56% less backfilled jobs on average, respectively, when compared to FCFS. Although it is unlikely that backfilling would harm the scheduling, as mentioned above, SPF and SAF are more consistent and predictable policies, since jobs are more likely to be scheduled for execution following the policy order, as oppose to being scheduled by "jumping ahead" in the waiting queue in unpredictable moments.

4.4 Conclusion

In this chapter, we move towards providing more knowledge and experience on what are the expectations if one decides to change the First-Come-First-Served

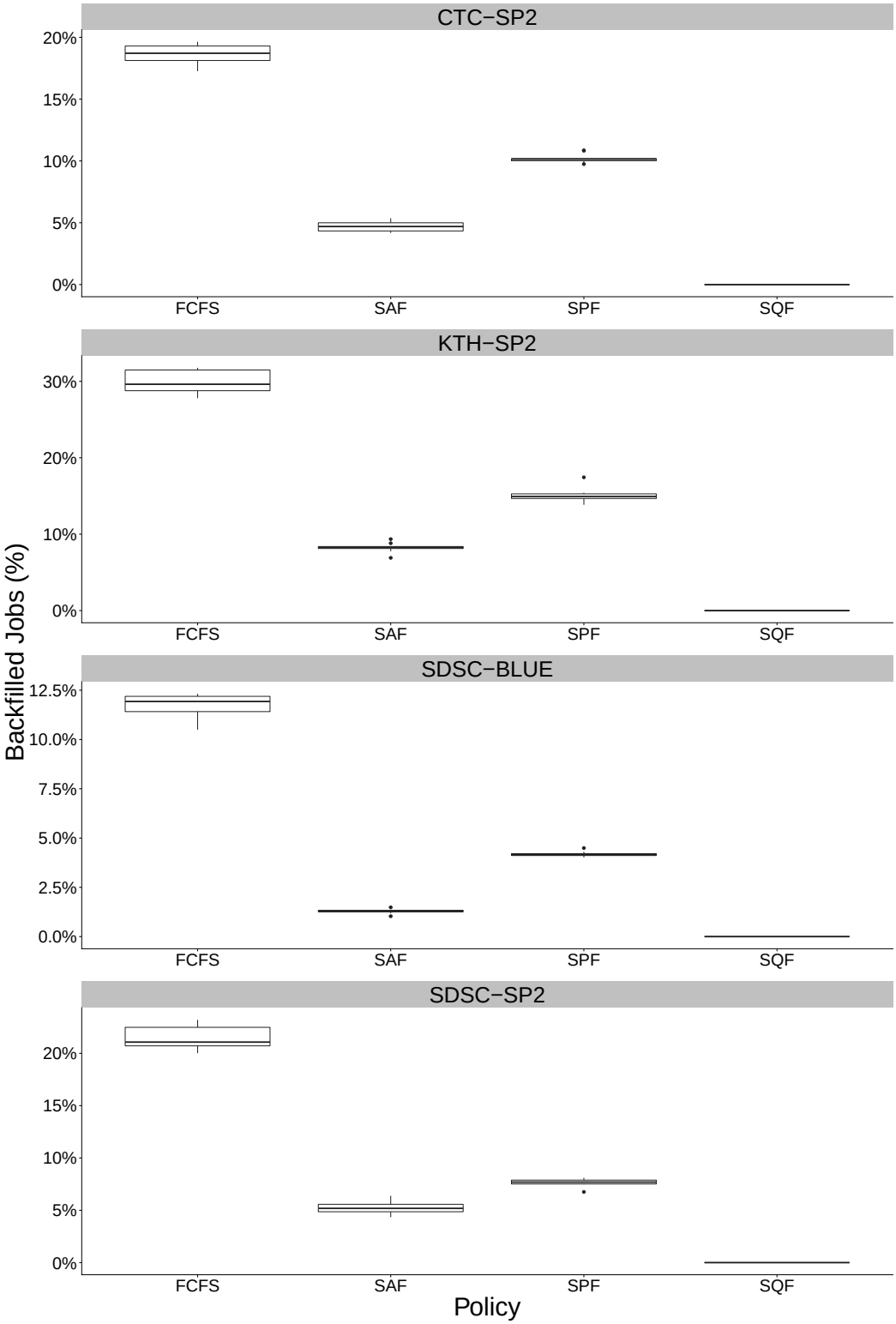


Fig. 4.4: Distribution of backfilled jobs between resamplings.

(FCFS) scheduling policy with aggressive backfilling – the popular EASY Backfilling – scheduling algorithm. We selected a class of simple scheduling algorithms that differs from EASY Backfilling by changing the scheduling policy (other than FCFS)

and adding a thresholding mechanism (to provide the same no starvation guarantees as FCFS). We used a flexible and reliable simulation software and exploited the rich information presented in HPC platform workload traces to find what could be observed and gained by using these other simple scheduling algorithms rather than EASY Backfilling.

Our results indicate that one can only gain by replacing EASY Backfilling with simple policies that consider the estimated processing time and the required resources, notably the Shortest Processing time First (SPF) and Shortest Area First (SAF). By adding a simple thresholding mechanism, it is possible to obtain significant performance improvements for the long run, using three relevant performance objectives, while also guaranteeing that every job in the waiting queue will eventually be executed. We show that these simple policies not only present better performance in average values, but they also significantly increase the number of jobs executed instantly (without waiting) and lower the number of jobs that wait for a long time. The performance gains over EASY Backfilling is distributed among all waiting jobs.

These simple policies also show that they can perform well with less interference from backfilling: the scheduler is more likely to follow the original order as set by the chosen scheduling policy, and not by the rules of backfilling, thus providing more predictability and transparency, two properties that are sought by HPC platform administrators.

We also highlight a less known scheduling policy in the literature, the Shortest Area First (SAF). In our experimental campaign, we found that this policy managed to consistently provide close to the best (if not the best) observed performance in all scenarios and performance objectives we evaluated. For instance, considering the slowdown objective, SAF not only provided an average overall performance increase up to 83.4%, but as well increased the number of jobs that run immediately by up to 9% and lowered the number of jobs who waited for a long time (very long slowdown) by up to 2.8 times, in comparison with FCFS. This result reinforces the relevance of the jobs' area property, which was seen in our previous work [21], and raises the question about possible analytical properties of SAF. Nevertheless, we reinforce that SAF must be considered as a baseline of comparison in future parallel batch scheduling research.

Last but not least, we present some cautions that must be considered if one wants to provide a scheduling algorithm that minimizes the maximum of an objective function. Taking the slowdown objective function as an example, we observed a class of jobs whose presence in the workload is not negligible and can mistakenly lead to inflated maximum slowdown values. If one only looks at the maximum of an

objective function to evaluate the scheduling performance, some good scheduling policies (as the aforementioned ones) can be overlooked.

One interesting questions can be derived from this study. *If we are prepared to relinquish the simplicity of basic index policies, can we achieve greater improvement? And under what conditions?*. We attempt to answer this question in the following chapter.

5

Adapting batch scheduling to workload characteristics: what can we expect from Online Learning ? [1]

5.1 Introduction

Taking the right scheduling decision is a complex problem that requires considering a large number of factors. Some of which are clear and visible but most are not. In the face of such growing complexity, many system administrators opt for the simple answer: use simple dispatching rules that are based on intuition and that offer certain guarantees, e.g. First Come First Served (FCFS) to prevent starvation, or Shortest processing time First (SPF) because it favors interactivity, or Smallest area First (SAF) as we argued in the previous Chapter. However, they are far from optimal and many studies [90, 65, 42] show that there is still room for software optimization. A common practice for RJMS is to keep execution logs that detail the history of the platform: the characteristics of the submitted jobs, their arrival times and other important information (Section 3.1.1). In this work, we explore the possibility of employing this historical data to adapt to future workload using more flexible scheduling policies. We base our experiments on EASY [70], which is one of the most popular backfilling schemes, and we propose a data-driven experimental campaign through which we exploit real execution traces in the form of logs extracted from the parallel workload archives [74]. First, we show the limits of simple, index policies. Then, we propose a new class of policies, which we call Mixed policies. Using this class we prove that simple policies are far from optimal and that under the correct conditions, we can obtain significant gains.

- We prove that it is possible to generate policies that significantly outperform any pure policy by mixing job features such as the estimate processing time,

[1] The text of this chapter is adapted from the following published paper: Arnaud Legrand, Denis Trystram, **Salah Zrigui**. Adapting Batch Scheduling to Workload Characteristics: What can we expect From Online Learning?. IPDPS 2019 - 33rd IEEE International Parallel & Distributed Processing Symposium, May 2019,

the required resources, and the waiting time in a simple weighted linear combination.

- We present a mapping of the space of possible policies through which we show that the evolution of the workload through time is very chaotic, which prevents online learning algorithms from being effective.

The remainder of this chapter is organized as follows. Section 5.2 details the specific experimental setting used in this work. In Section 5.3, we provide a quick comparison of the pure policies presented in Section 3.1.4 and provide the incentive for this work. In Section 5.5, we present and test the proposed method to obtain mixed scheduling policies.

5.2 Experimental setting

We use 4 of the real-world traces from the parallel workload archives presented in Table 3.1; CTC-SP2, KTH-SP2, SDSC-SP2, SDSC-BLUE.

For every trace, we ignore the first period since it generally corresponds to a benchmarking/testing phase and is not representative of the true workload of the system. Then, we split the trace on a weekly basis and remove the jobs that start in one week and finish in another. We consider 45 consecutive weeks from CTC-SP2 and KTH-SP2 and 100 consecutive weeks from SDSC-SP2 and SDSC-BLUE, and we simulate the execution of all the policies for each week and measure the weekly average *BSLD* given in Equation (3.3).

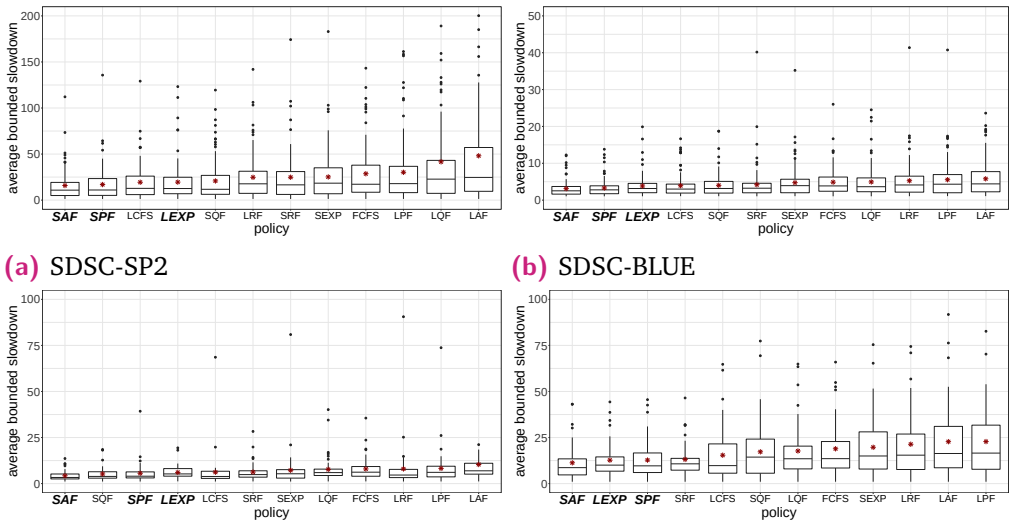
We tried to be as transparent as possible and to make our work reproducible [83]. We provide a snapshot of the workflow we used throughout this work as a link to a git repository[2], which includes a nix [29] file that describes all the dependencies and four R notebooks that allow regenerating all the figures.

In this work, we replace the RJMS with a EASY-backfilling lightweight simulator [2] written in OCaml. It supports tuning the Primary, Backfilling queues and it discard all topological information of the machine.

5.3 Performance evaluation of pure policies

We decided to expand the policy set from Chapter 4 to include other pure policies presented in Section 3.1.4.

[2] `https://gitlab.inria.fr/szrigui/mixed-policies`



(a) SDSC-SP2 **(b)** SDSC-BLUE

(c) CTC-SP2 **(d)** KTH-SP2

Fig. 5.1: Tukey box-plot of the weekly average bounded slowdown of pure policies for the 4 traces. The policies are sorted in an increasing order by the mean of the weekly average bounded slowdown for all the weeks. The three most efficient policies are highlighted.

Figure (5.1) illustrates the results. The order of the policies with regard to performance changes between the traces. In general, the policies that prioritize shorter jobs, namely SAF and SPF and LEXP, are better for the average *BSLD*. SAF comes on top for all the tested traces followed by SPF and LEXP.

As expected, FCFS is not a good policy for minimizing the average *BSLD*. Although its exact position changes between traces, it always ranks among the worst policies. Interestingly, LEXP, the policy that represents the estimate of the very metric we are trying to optimize, is not the top policy, which indicates the importance of considering the amount of required resources when taking a scheduling decision.

The good performance of SAF, SPF, and LEXP can be explained by the fact that the slowdown of a job is proportional to its length. Longer jobs can wait for a longer time without having their slowdown grow drastically. The slowdown of shorter jobs, however, increases very fast the longer they wait.

The one size fits all policy?

From the previous comparison and the results of chapter 4, we observe that SAF is overall better than all the other tested policies to optimize the average BLSD. It gives the lowest mean on an aggregation of weeks and its outliers are not as extreme as other policies.

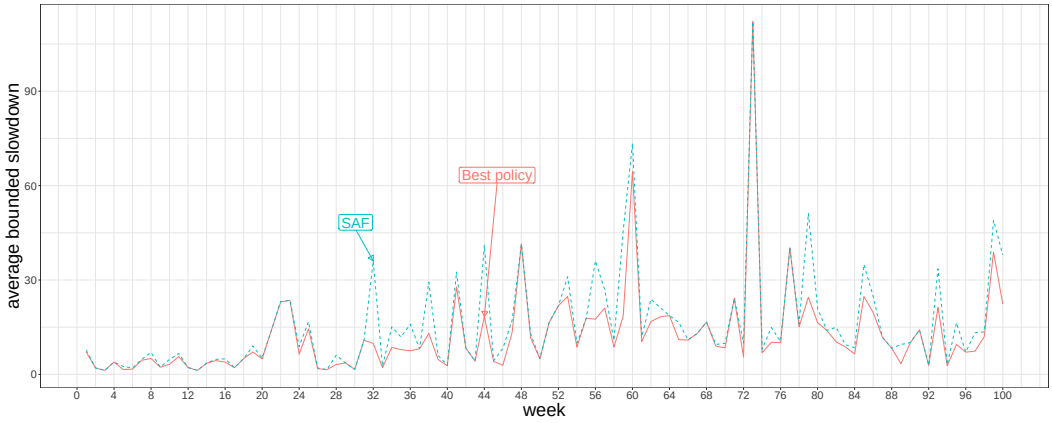


Fig. 5.2: Comparing SAF, the best pure policy on average, with the best pure policy for every week for the SDSC-SP2 trace.

Figure 5.2 illustrates a more detailed comparison between SAF and the other policies on a given workload. We compare the average *BSLD* of SAF with the average *BSLD* of the best pure policy for every week individually. As expected, SAF performs well for most weeks. It is either the best policy or very close to the best. However, we can spot many weeks where another pure policy performs better than SAF by a significant margin (e.g. 38, 44, 56, and 85). Regardless of which policy outperformed SAF, the observation is the same for the four studied traces; SAF is good overall but it remains far from the optimal in many cases.

Finally, it is worth noting that traces where SAF fails can be found. For example, for the ANL-Intrepid trace from the Parallel Workload Archive [74], SPF is the best pure policy with an average slowdown of 35.92 while SAF ranks at 7 over 12 with 39.78. Likewise, with the Sandia trace, LAF is the best with an average slowdown of 7.353 while SAF ranks again at 7 over 12 with 10.396. We pick the traces in Table 3.1 to study due to their popularity in the literature [42, 21]. Moreover, the focus of this work is not to show that a single pure policy is dominant but to study the possibility of improving the performance of schedulers using historical data.In this section, we give a quick comparison of pure policies as part of the reason we decide to expand the possible polices space. Chapter 4 include a full study of pure policies

To make the reading and the analysis easier and to avoid redundancy, all the experiments in the following sections are done using a single trace: SDSC-SP2. The same behavior is observed in the rest of the traces. We show their results at the end of this Chapter (Section 5.7).

5.4 Mixed policies

We consider a job j to be characterized by a feature vector $x_j = (q_j, \tilde{p}_j, wait_j, \rho_j, exp_j, a_j)$.
At each scheduling decision, we define the score of any job j using Equation (5.1).

$$score(\mathbf{w}, x_j) = \mathbf{w}^T x_j \quad \mathbf{w} \in \mathbb{R}^n \tag{5.1}$$

where **w** is the weight vector of the mixed policy: each feature x_i has a corresponding weight $\mathbf{w}_i$. These weights are what determine how the mixed policy behaves. The absolute value of a weight $|\mathbf{w}_i|$ indicates the importance of the corresponding characteristic x_i when ordering the jobs. While the sign determines the ordering itself, a positive value means that shorter/smaller jobs are prioritized, while a negative value means that longer/larger jobs are prioritized.

The scoring function is scale-invariant; the order given by $score(\lambda\mathbf{w}, x_j)$ is the same as the order given by $score(\mathbf{w}, x_j)$ for all $\lambda > 0$. Hence, we normalize **w** and impose that $||\mathbf{w}||_1 = 1$. This constraint reduces the size of the search space and stabilizes the learning process (which will be explained in detail in Section 5.6.1). Every pure policy corresponds to a vertex of the polytope $||\mathbf{w}||_1 = 1$. *E.g.* FCFS corresponds to (0,0,1,0,0,0) and LCFS corresponds to (0,0,-1,0,0,0).

Mixed policies are an alternative method to model the scheduling problem. We move from a discrete optimization to a continuous optimization problem. We construct a search space that is small in size and instead of finding the best ordering of n independent jobs we intend to find the best weight for i features where i is much smaller than n.

5.5 Scheduling using mixed policies

In the previous Section, we showed that among all the pure evaluated policies there is no single policy that is dominant across all weeks. SAF offers a reasonable compromise but it fails in many cases. This motivates the need for developing a scheduling approach that adapts to the state of the system and the workload.

In this section, for the sake clarity, we limit the mixed policies vector to only three elements: $x_j = (q_j, \tilde{p}_j, wait_j)$. Further results involving all the six features will be presented in Section 5.6.

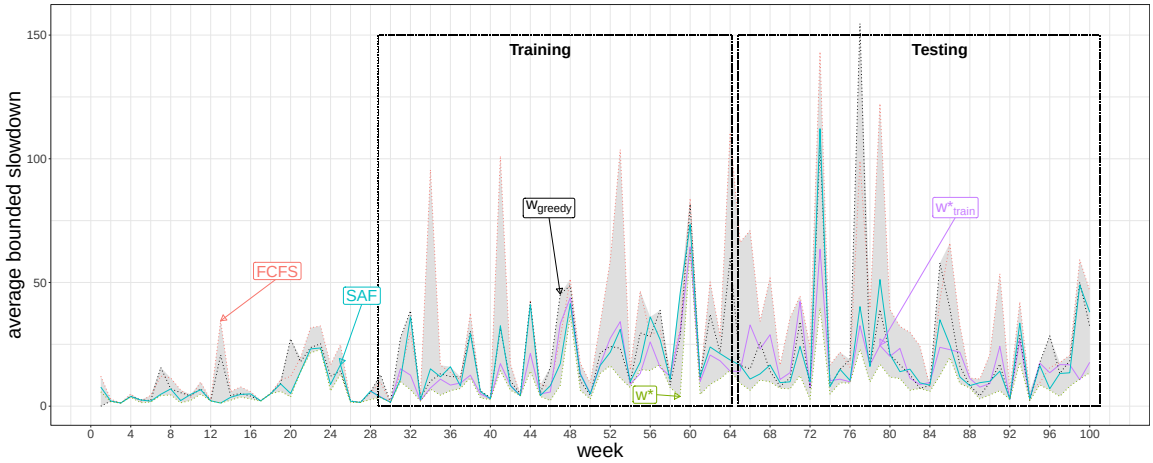


Fig. 5.3: Comparing the performance of various policies on the SDSC-SP2 trace. w* represents the best policy in hindsight for every week. w^*_{train} is the policy obtained from learning on the *Training* weeks, and w_{greedy} gives the results of testing the best policy of one week on the next.

5.5.1 Comparing pure and mixed policies

We consider a set of 100 weeks from SDSC-SP2 and we separate them in the same way as in Section 5.3. Then, for each week we perform the following:

- Simulate using the two pure policies: (1)FCFS because of its popularity (although it is not very effective for the Average *BSLD*), and (2)SAF because, as observed in Section 5.3, it is the best policy overall.
- Generate a large number of weight by performing a uniform discretization of the search space. We take a sequence of 100 points from each dimension which can take a negative or a positive value. Thus, for three features we have $100^3.2^3 = 8.10^6$ points. Then we simulate each point and we pick the best vector *i.e.* the one that gives the lowest scores for this week, and which we denote **w***.

The results are shown in Figure 5.3. **w*** represents the average *BLSD* of the best weekly linear combination. The gain of **w*** compared to the pure policies varies significantly. We can classify the weeks into two types.

- Weeks where there is no or a very small difference in performance between both pure and mixed policies. The average *BSLD* of such weeks tends to be very close to 0. Weeks 43, 92 and 94 are good examples of this type. Their workload is so relaxed that no optimization is required. According to Figure 5.3, around half of the weeks of SDSC-SP2 belong to this type.

Policy	*Training*	*Testing*
w*	376.67	357.51
$w*_{train}$	682.11	778.44
SAF	691.10	721.54
SPF	706.24	787.92
w_{greedy}	818.71	902.55
LEXP	820.94	934.21
SQF	970.49	869.41
SEXP	1016.52	1204.73
LRF	1041.18	1134.92
SRF	1147.96	1114.46
FCFS	1180.24	1398.13
LPF	1239.79	1483.35
LQF	1702.14	2191.97
LAF	2109.84	2355.16

Tab. 5.1: Comparing the sum of the average *BSLD* for SDSC-SP2 for weeks: 65 to 100. The highlighted values are obtained in hindsight.

- Weeks where there is a difference in performance between the policies. For weeks such as 64, 73, 79, and 100, we observe significant variation in performance and a much higher *BLSD*. For this type, we also notice that **w*** is significantly better than all other policies. In week 73, for example, **w*** reduces the average *BLSD* by a substantial margin, approximately 2.5 times less than SAF, the best pure policy for that week, and 3 times less than FCFS.

Pure policies are thus far from the optimal and a carefully selected combination of features can give substantial improvement. However, the value for the best weight for each week can be quite different from the others ($\mathbf{w*}_i \neq \mathbf{w*}_j \quad \forall i \neq j \in 1..100$). This shows the changing nature of the workload through time and will be discussed in detail in Section 5.5.3.

5.5.2 Learning: scheduling using best combination learned from a previous part of the trace.

In this section, we evaluate the generalization capacity of our approach. We investigate how the best combination **w*** for a part of the trace performs on another part. We evaluate this ability by using two different strategies.

5.5.2.1 Learning over a long period of time

The idea is to divide the trace into two equal parts and see how the best policy on the first half performs on the second.

For this particular trace, we decided to ignore the first 28 weeks because the workload at the beginning of the trace is rather light, hence all the tested policies perform similarly. So we consider the first 28 weeks as non-representative of the actual workload. Then divide the 72 remaining weeks into two parts of equal sizes. We call the first part *Training* and the second part *Testing*.

$$\mathbf{w^*}_{train} = \underset{\mathbf{w}}{argmin} \sum_{week=28}^{64} average\_BSLD_{week}(\mathbf{w}) \tag{5.2}$$

Weeks 28 to 64 (*Training*): We aggregate using equation 5.2 and we find the weights w^*_{train} that minimizes the sum of the weekly average *BSLD* over all weeks.
Weeks 65 to 100 (*Testing*): we evaluate w^*_{train} on the new *Testing* weeks.

The aggregated results are illustrated in Table 5.1 and the details for each week are given in Figure 5.3.

Training: w^*_{train}, the learned policy, slightly outperforms SAF in general. But if we look at individual weeks we see that SAF still has a lower *BSLD* sometimes over the training period (e.g. 34 and 52).

Testing: Table 5.1 show that w^*_{train} performs quite well compared to other policies. But it is still surprisingly equivalent and even outperformed by SAF.
Figure 5.3 shows the performance of both individual weeks. SAF is better for some weeks (namely 68, 81, and 82) but w^*_{train} is better for others like (e.g 73,79, 85). Sometimes both policies give similar results.

Although *Training* and *Testing* do not particularly appear as different, The best weights for *Training* are not the best for *Testing*: there is no one size fits all strategy. By comparing $\mathbf{w^*}$ (see Section 5.5.1) and w^*_{train} in Figure 5.3, we observe that w^*_{train} is far from the best possible vector even for the weeks used for *Training*.

5.5.2.2 Learning over a short period of time

We investigate if the policy learned from one week can be effective on the next by evaluating the vector learned from week i ($\mathbf{w^*}_i$) on the next week $i+1$.

In Figure 5.3, the policy w_{greedy} represents the results of simulating the workload of one week using the top policy from the previous week. There are unfortunately no patterns to distinguish. The vectors learned from the previous week seem to evolve and perform in a chaotic manner. Sometimes they perform better than SAF (weeks 56, 83, and 89), sometimes worse (weeks 20 and 55), and sometimes on par with SAF.

Using the policy learned from the previous week does not lead to good performance at all. We hypothesize that the structure of the workload (the jobs submitted) changes substantially from one week to the next. Thus, online-learning the optimal weights may be very difficult.

5.5.3 Exploring the search space

In this section, we explain why there is no single vector of weights that is optimal for all cases. We visualize the search space and observe the position of the optimum for different weeks.

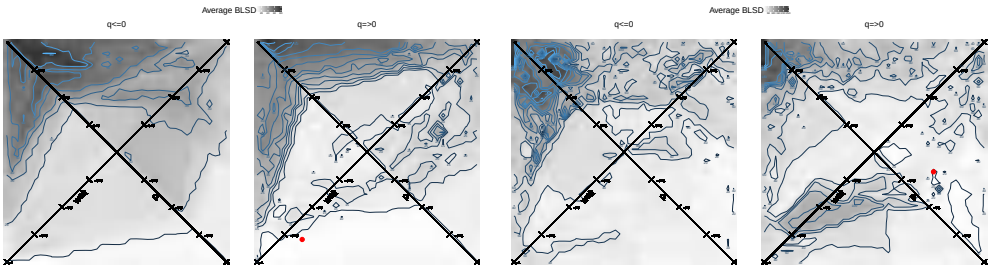
Figure 5.4 is a 2D representation of the search space for 4 consecutive weeks of the SDSC-SP2 trace. Each week is represented by two figures: the left figure displays the weekly average *BLSD* where $q \leq 0$ and the right figure, where $q \geq 0$. The $\nwarrow$ and $\nearrow$ axes respectively represent the weights of $\tilde{p}$ and $wait$. The optimal combination always lies in the lightest area and is represented by a red dot.

The coordinates of the optimal point change drastically from one week to another. Using the optimal point of week 72 to schedule week 73 give poor results because the optimal point in 72 lies in an area that has a very high slowdown in week 73. This explains why the short period learning failed.

Furthermore, with the exception of general similarities like the half where $q \geq 0$ have a lower *BSLD* than $q \leq 0$, we also observe that the position, shape, and even the size of the optimal area changes radically from one week to the next. This explains why online learning seems compromised without further information.

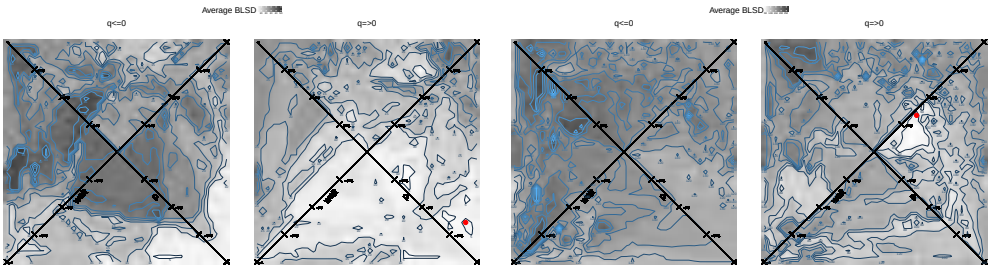
5.6 Increasing the size of the search space: using more jobs characteristics

In this section, we investigate the impact of using all six job characteristics on performance. Indeed, the experiments in all the previous sections were done with only the three basic job characteristics: p,q, and $wait$. In this Section we extend the search space to include the three other characteristics introduced in Section 3.1.4 which are a,r,exp.



(a) week 70:
w*($q = 0.30, \tilde{p} = 0.37, wait = -0.33$)

(b) week 71:
w*($q = 0.27, \tilde{p} = 0.35, wait = 0.38$)



(c) week 72:
w*($q = 0.32, \tilde{p} = 0.62, wait = 0.06$)

(d) week 73:
w*($q = 0.47, \tilde{p} = 0.05, wait = 0.48$)

Fig. 5.4: Visualization of the search space for 4 consecutive weeks 70, 71,72, and 73. The two diagonal axis represent $\tilde{p}$ and $wait$. The lighter the area is, the better the performance (lower average *BSLD*). The optimal area change from one week to the next. The red dot (in the lightest area) represents $w*$ and the blue triangle represents w$*_{train}$.

5.6.1 Black-box optimizers: a quick way to find the optimal

Algorithm

In the previous Section, finding the weekly best mixed policy was done using a uniformly exhaustive search. We made a fine discretization of the whole search space and we selected the weight vector **w*** that provides the lowest average $BLSD$(equation (5.3)). Performing an exhaustive space search becomes costly very fast because the size of the search space grows exponentially with the number of job characteristics we include in the linear combination. Thus another method to find the minimum is required.

$$average_{BSLD} = \frac{1}{n} \sum_{j=1}^{n} F(x_j, \mathbf{w}), \tag{5.3}$$

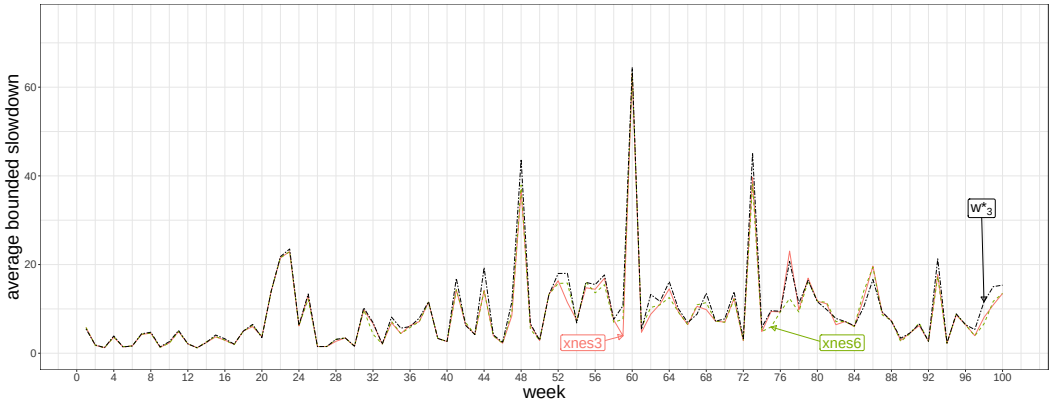


Fig. 5.5: Comparing average *BSLD* of the vectors of the 3 original features (xnes3) with the extended vector of 6 features (xnes6) and the minimum we obtain from space coverage (w*$_3$)

Our goal is to find a combination of weights **w*** that minimize average $BLSD$ while enforcing the constraint $||w||_1 = 1$. This can easily be done by optimizing the following objective function:

$$\sum_{j=1}^{n} F(score(\mathbf{w}, x_j)) + \lambda \left(||\mathbf{w}||_1 + \frac{1}{||\mathbf{w}||_1} \right) \tag{5.4}$$

Function F has *a priori* no particular properties. Furthermore, we have seen in Section 5.5.3 that the search space is not convex and it may exhibit several local minima. Therefore, gradient-based methods cannot be used and we have to rely on stochastic derivative-free methods. We initially tried the standard simulated annealing method [62] but it got frequently stuck in local optimums. A study of the existing literature [12] led us to the evolutionary algorithms family that considers an ensemble of candidates. We tested several algorithms, Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [51] and eXponential Natural Evolutionary Strategy (XNES) [44] provided the best results. Since XNES is faster than CMA-ES, we chose the former.

Performance

For each week we apply the XNES algorithm to obtain a solution of Equation (5.4) for a vector of dimension 3 (*xnes3*) and a vector of dimension 6 (*xnes6*) and we compare the results with the minimum obtained from the space coverage which we call w*$_3$ (corresponds to the **w*** used in Section 5.5.1). Figure 5.5 illustrates the results.

For most weeks *xnes3* and w*$_3$ give the same result. For few other weeks, *xnes3* managed to slightly outperform w*$_3$. This is due to the method used to cover the

search space: Each dimension of the vector gets 200 points distributed uniformly over [-1,1]. *XNES* does not have that constraint, hence it can produce policies that are more *refined*. The differences in performance are minor which indicate that *XNES* managed to find a vector that is the actual or at least very close to the optimum every time. Thus *XNES* can be considered as a viable option to find an optimal vector.

The *BSLD* of *xnes3* and *xnes6* are not very different from each other. For most of the weeks, both vectors perform equally. In some rare cases (weeks 86 for example), *xnes3* gives a slightly better performance than *xnes6* but the difference is marginal (*XNES* converged to a local optimum instead of the global optimum in the case of 6). On average *xnes6* is better than *xnes3* but not by a larger margin.

Increasing the size of the search space by adding job characteristics improves the results by a small very margin.

5.7 Using other traces:

We reproduce all the Figures in the previous sections of this chapter for the other three traces: SDSC-BLUE, CTC-SP2, KTH-SP2. We use the same experimental setting as the main article (same threshold, same granularity, same parameters of the optimization algorithm).
Here we give general observations that are shared by all the traces. For each trace, we dedicate a notebook that contains all the details.
observations:

- **Pure Policies:** The scale of the platform and the workload both change greatly between traces but we can observe that the general order of the pure policies is the same. As observed SAF is still the dominant followed by SPF and LEXP.

- **Mixed policies:** The optimal value **w*** that can be achieved by the Mixed policies is still far better than any of the pure policies.

- Learning the optimal combination for one part of the trace (aggregation of half of the used weeks in this case) gives a good policy that is comparable to SAF in terms of performance. We can see this in the various notebook by looking at the performance of **w_train**. For the testing half, **w_train** is either the best or the second best for all the traces.

5.7.1 SDSC-BLUE

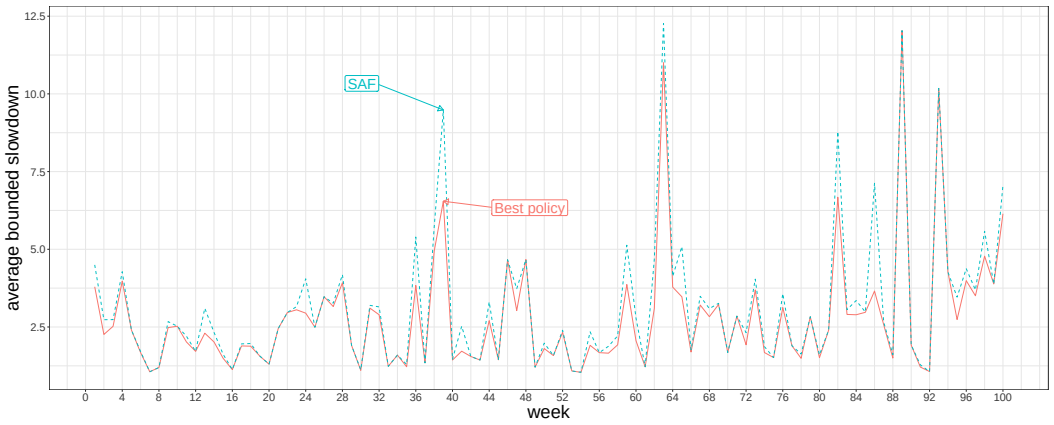


Fig. 5.6: **SDSC-BLUE:** Comparing SAF, the best pure policy on average, with the best pure policy for every week.

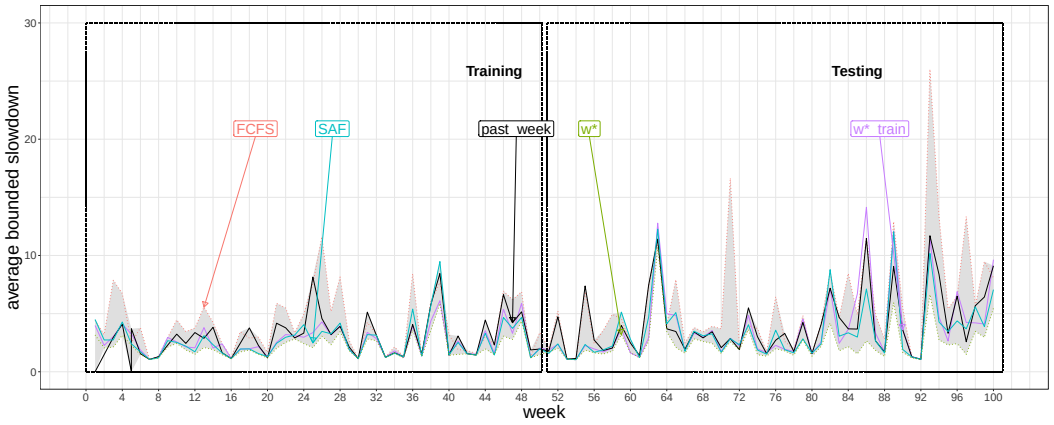


Fig. 5.7: **SDSC-BLUE:** Comparing the performance of various policies. w* present the optimal policy for every week. w_train* is the optimal policy obtained from learning on the **training** weeks, and **w_greedy** is the results of testing the optimal policies of one week on the next.

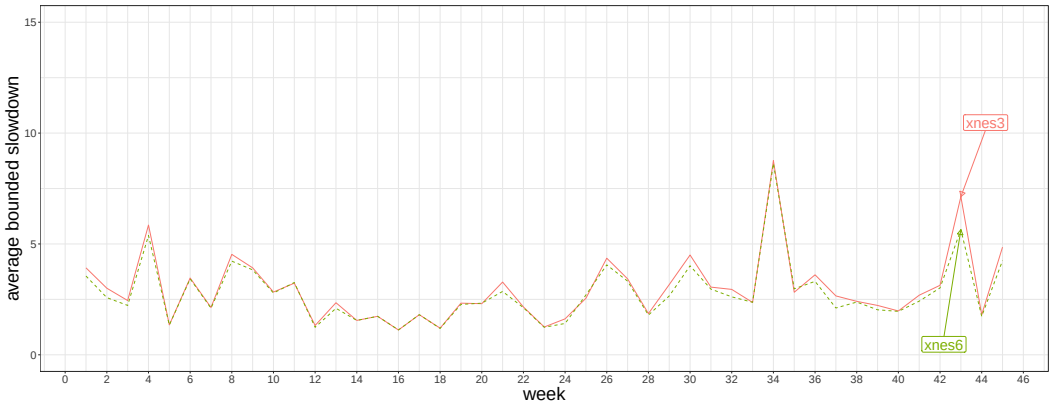


Fig. 5.8: **SDSC-BLUE:**Comparing average *BSLD* of the vectors of the 3 original features (xnes3) with the extended vector of 6 features (xnes6)

5.7.2 CTC-SP2

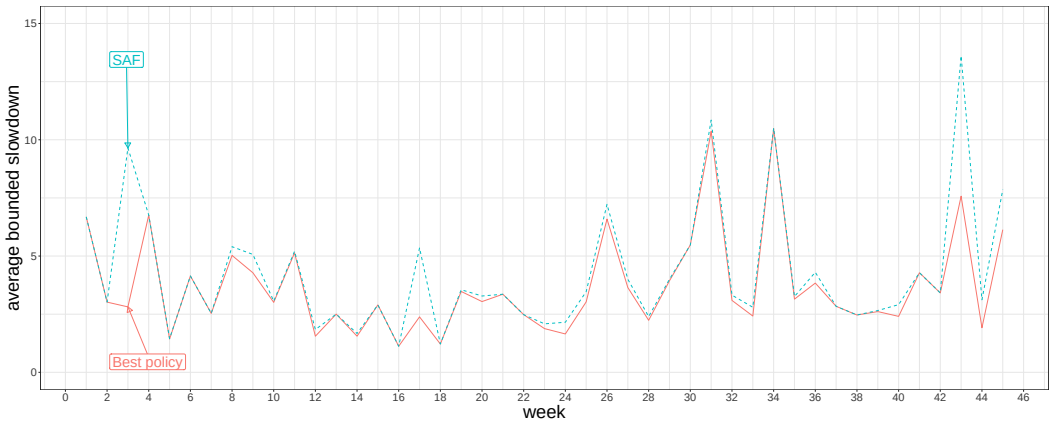


Fig. 5.9: **CTC-SP2:** comparing SAF, the best pure policy on average, with the best pure policy for every week.

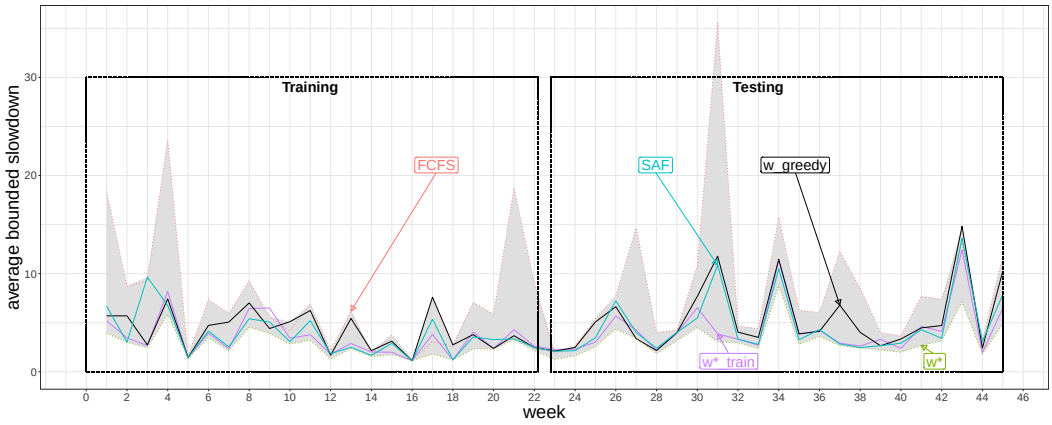


Fig. 5.10: **CTC-SP2:** comparing the performance of various policies on the CTC-SP2 trace. w* present the optimal policy for every week. w_train* is the optimal policy obtained from learning on the **training** weeks, and **w_greedy** is the results of testing the optimal policies of one week on the next.

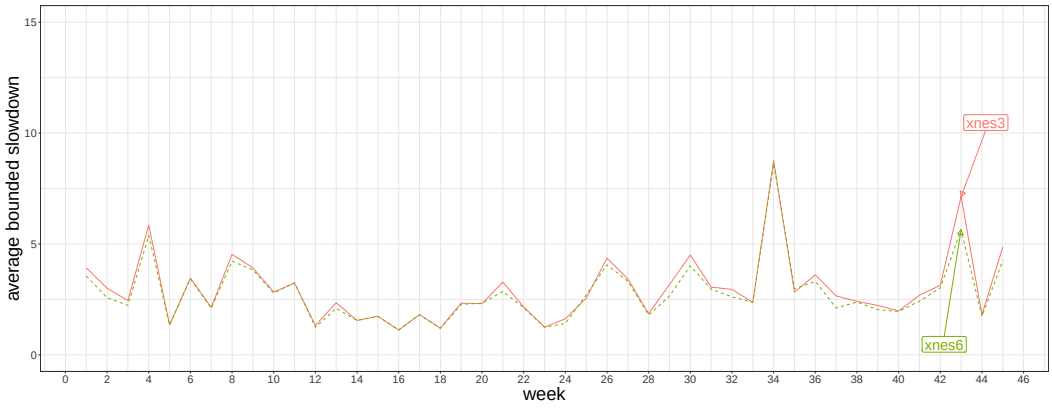


Fig. 5.11: **CTC-SP2:** comparing average *BSLD* of the vectors of the 3 original features (xnes3) with the extended vector of 6 features (xnes6)

5.7.3 KTH-SP2

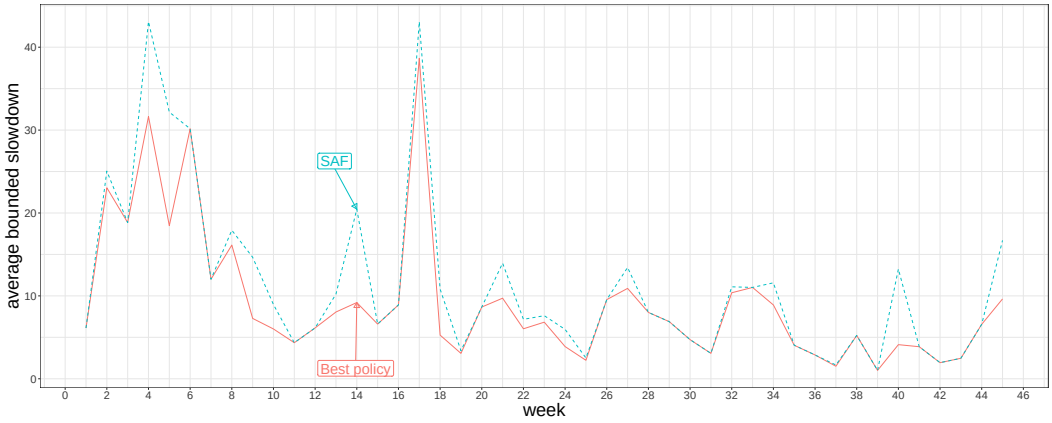


Fig. 5.12: **KTH-SP2:** comparing SAF, the best pure policy on average, with the best pure policy for every week.

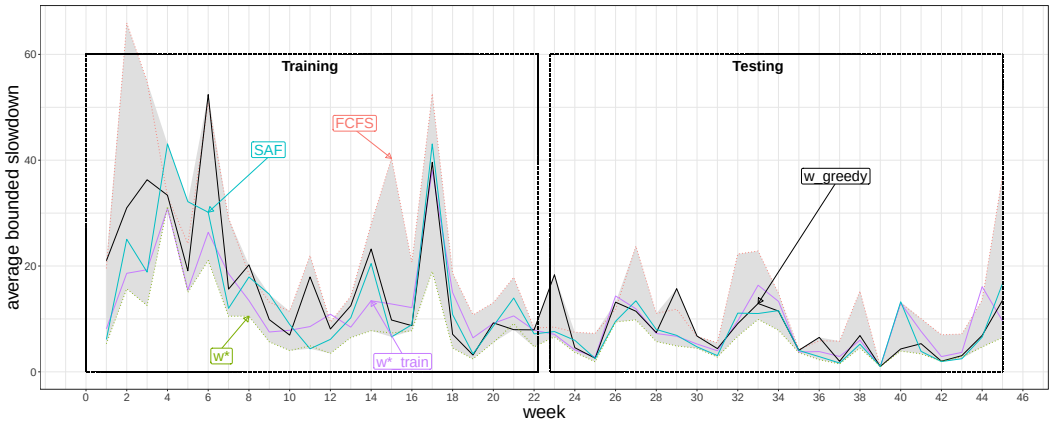


Fig. 5.13: **KTH-SP2:** comparing the performance of various policies. w* present the optimal policy for every week. w_train* is the optimal policy obtained from learning on the **training** weeks, and **w_greedy** is the results of testing the optimal policies of one week on the next.

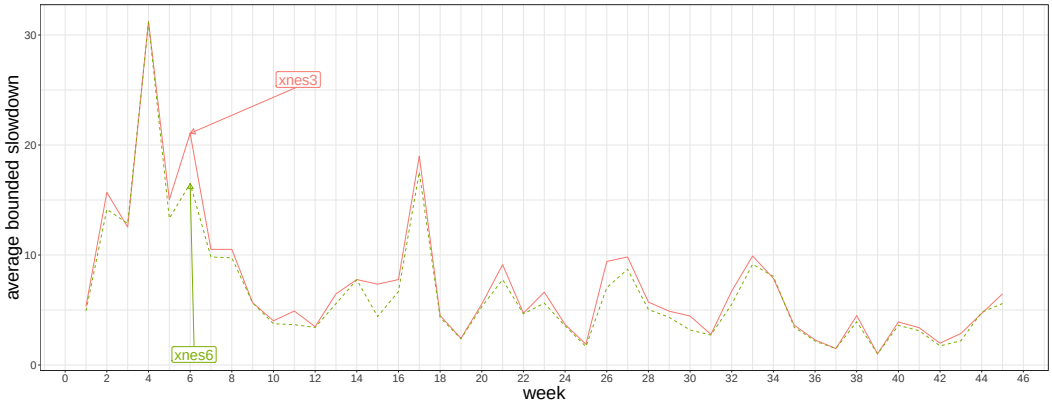


Fig. 5.14: **KTH-SP2:** comparing average *BSLD* of the vectors of the 3 original features (xnes3) with the extended vector of 6 features (xnes6)

5.8 Starvation/thresholding

We test three thresholding configurations. First, we consider the case where there is no threshold. The jobs wait as much as the selected index policy demands. Second, we consider a low thresholding value (20 hours) and finally, we test the value that is used for the rest of our research (2.31 days). Tables 5.2, 5.3, 5.4 and 5.5 show the aggregated results.

Removing the threshold

When we remove the threshold altogether the tested policies become more effective. The ordering set by the policies is not perturbed by the thresholding mechanism. For that reason, we notice a bigger difference in the performance of various policies.

Choosing a tighter threshold: 20 hours

When we select a low threshold, we notice all that the values of the average BSLD get closer to FCFS. This is because the waiting time of a high number of jobs surpassed the threshold. The thresholded jobs are scheduled using the FCFS rule.
Table 5.6. Show the longest jobs in the SDSC-SP2 trace. their runtime is clear much longer than 20h = 72000s. Where using a policy that prioritizes shorter/smaller jobs the waiting time of such jobs are almost guaranteed to surpass the threshold.

In this work, we treat thresholding as a contingency mechanism and it should only be used in extreme cases. So we set it to be 2,31 days (200000 seconds).
Choosing the right thresholding value is important. It is a compromise between giving the scheduling policy the freedom to order the jobs as it wills, and preventing the waiting time of low priority jobs from becoming too long.
We do not go into too much detail on how to fix a thresholding value but deeper and a more elaborate study on how to choose the proper thresholding value is available in [65].

5.9 changing the granularity: Using months

Instead of using weeks we use months. We find the best achievable value for each month $\mathbf{w}^*$, and we train on the first 10 months to get $\mathbf{w}^*_{train}$, then we test it on the

policy	No threshold	2.31 days	20 hours
SAF	1461.39	1585.53	2395.29
SPF	1467.70	1689.55	2447.58
LCFS	1739.24	1940.71	2471.02
LEXP	1936.93	1955.04	2496.97
SQF	1988.62	2082.25	2579.47
SRF	2426.40	2475.90	2818.35
SEXP	2519.32	2484.96	2864.63
LRF	2726.74	2517.65	2875.95
FCFS	**2864.63**	**2864.63**	**2879.22**
LPF	3277.66	3020.96	3064.14
LQF	4188.31	4159.97	3346.24
LAF	5322.73	4811.85	3525.13

Tab. 5.2: Comparing thresholding values for SDSC-SP2

policy	No threshold	2.31 days	20 hours
SAF	302.73	311.83	344.51
SPF	329.95	328.88	350.36
LEXP	384.27	383.99	396.27
LCFS	388.40	392.03	407.91
SQF	395.27	401.76	416.65
SRF	409.12	418.78	431.64
SEXP	468.82	473.94	465.08
FCFS	**487.37**	**487.37**	**487.37**
LQF	494.14	493.77	508.77
LRF	522.95	526.12	514.99
LPF	566.17	554.11	563.42
LAF	582.87	578.66	563.43

Tab. 5.3: Comparing thresholding values for SDSC-BLUE

policy	No threshold	2.31 days	20 hours
SAF	501.16	507.76	632.93
SPF	554.63	571.57	638.55
LEXP	568.07	573.80	651.27
SRF	600.80	590.25	651.45
LCFS	679.75	692.97	768.40
SQF	772.55	775.86	784.52
LQF	797.87	796.77	786.36
FCFS	**850.16**	**850.16**	**850.16**
SEXP	882.39	886.61	889.64
LRF	1001.39	961.17	917.95
LPF	1066.83	1023.84	963.46
LAF	1076.52	1026.10	976.46

Tab. 5.4: Comparing thresholding values for KTH-SP2

policy	No threshold	2.31 days	20 hours
SAF	190.95	191.24	219.51
SQF	238.50	235.10	248.85
SPF	254.26	251.65	251.64
LEXP	273.20	273.20	252.27
LCFS	276.91	277.79	273.96
SRF	281.82	280.94	283.89
SEXP	330.40	325.02	292.36
LQF	346.15	346.27	292.84
LPF	356.77	357.75	313.92
FCFS	**357.75**	**358.56**	**341.83**
LRF	366.45	370.98	357.75
LAF	466.47	466.42	386.57

Tab. 5.5: Comparing thresholding values for CTC-SP2

job_ID	submit_time	*run_time*	number_processors
28663	14837924	510209	9
28664	14837963	508420	16
8040	7660725	474510	6
25891	13785409	452520	60
9036	8427638	227033	64
31620	17005431	162564	4
31623	17005453	162536	32
17	577685	118561	5
14408	12186754	114369	8
34441	19458401	112013	64

Tab. 5.6: The jobs in SDSC-SP2 with the highest run_time that make the 20h threshold unreasonable

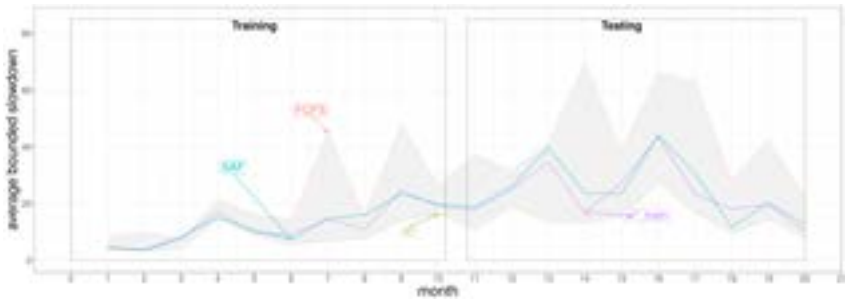
second half.
Figure 5.15 illustrates the results. **w***, optimal mixed policies, still outperform all other policies by a large margin. $\mathbf{w}^*_{train}$ and SAF are still comparable to each other. As far as we can observe changing the granularity does not change the results.

5.10 Conclusion

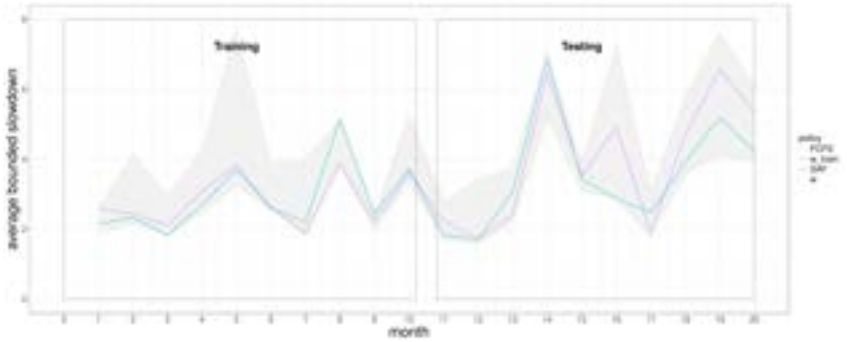
Scheduling parallel jobs in a real HPC platform is a complex task plagued with many uncertainties. Determining an efficient scheduling strategy is difficult due to the volatile nature of the workload. The main result of this work was to optimize the EASY-Backfilling algorithm by reordering the primary queue using policies learned from historical data.
We looked at the scheduling problem from a new perspective by studying a larger class of heuristics obtained from mixed policies that enable us to move from a

discrete to a continuous search space. We combined several characteristics extracted from the jobs in a linear expression and we determined the best weight for each characteristic.
We showed that pure policies are far from the optimal and that important gains can be obtained by using mixed policies. For some weeks in the simulation, we obtained results that are up to 3 times better than the best pure policy. Unfortunately, we observed that the structure of the workload changes too much over time and that whenever a policy performs well on a part of a trace, it does not mean necessarily that it will be efficient on another part of the trace.

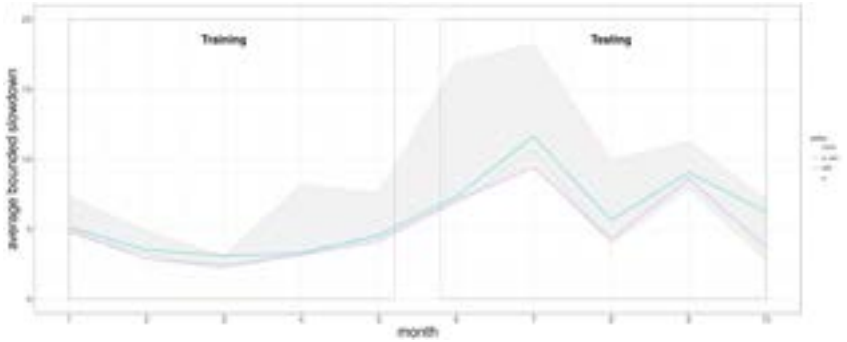
Using historical data to predict good scheduling policies for future jobs is not a straightforward task. We observed that the workload itself changes drastically from one time period to the next. We have yet to identify any meaningful pattern to these changes, which raises the question of whether it is possible to apply machine learning on real execution logs or not.



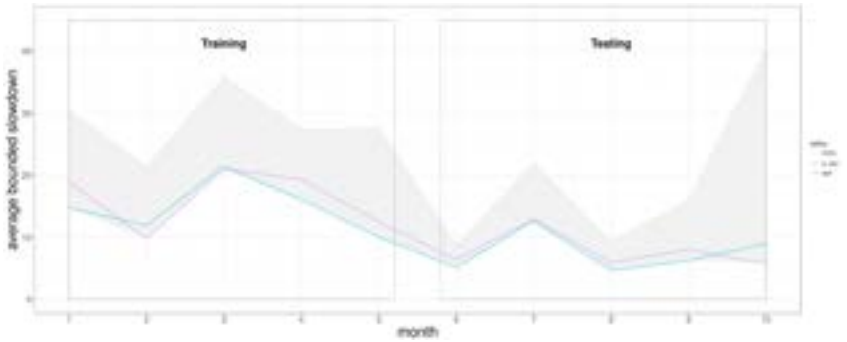
(a) SDSC-SP2



(b) SDSC-BLUE



(c) CTC-SP2



(d) KTH-SP2

Fig. 5.15: Comparing the performance on mixed and pure policies on the scale of month

6 Improving Online jobs scheduling via Classification

6.1 Introduction

In online scheduling, exact runtimes of jobs are never known in advance. Thus, in the majority of online scheduling scenarios, users are required to provide an estimation for the runtimes of their jobs. The scheduler uses these estimations as an upper-bounds to how long a job is allowed to run on the platform. To avoid having their jobs prematurely killed users tend to give highly overestimated values [90]. However, such overestimations negatively impact the performance of schedulers [9, 34]. Consequently, obtaining reasonable runtime estimates would be very valuable when designing HPC system schedulers.

Machine-learning techniques have emerged as a suitable tool to predict job execution times [90, 42, 49, 21]. However, it is difficult to estimate the execution times from historical data present in workload logs using regression-based techniques [64]. Such difficulty arises from the fact that crucial information, such as job dependencies, parameters, and even names, are often missing. Moreover, runtime information such as job placement and machine utilization are available only *a posteriori*. Consequently, although regression may allow better implementations of heuristics and tighter backfilling choices, obtaining reliable execution time estimates is rarely possible.

One insight one can have is that the key factor of heuristics that favor shorter jobs, such as SPF, is that executing small jobs first improves the metrics, such as the job flow time and the slowdown. In this work, we follow this insight and we propose to apply a simple two-class classification instead of regression. We classify the jobs into two general classes, namely small and large, and prioritize the execution of small jobs. Since performing two-task classification is easier than full regression, we expect to obtain better classification performance with less training data. We perform a thorough evaluation using six workload traces from actual HPC platforms and four scheduling policies, comparing the results of schedulers that use our job class classification with (i) standard schedulers, which rely only on user-provided

information, and (ii) clairvoyant schedulers, which have perfect knowledge of actual job execution times. We show that:

- The a priori knowledge of whether the job is small or large is sufficient to generate scheduling improvements close to those obtainable using fully clairvoyant schedulers;
- Our online classification algorithm achieves a precision ratio between 78% and 89% in all workload traces, which is sufficient to improve scheduling performance in all scenarios;
- Adding a safeguard mechanism that kills the jobs that are missclassified as small results in improvements similar to those obtainable using fully clairvoyant schedulers.

The remainder of this chpter is organized as follows. Sections 6.2 and 6.3 describe the method used for the classification. Section 6.4 describes the experimental protocol, followed by the experimental evaluation (Section 6.5). Finally, in Section 6.6, we give some concluding remarks.

6.2 Preliminary Observations

We use a data-driven approach, which relies on the characterization and identification of workload patterns from execution logs of HPC platforms. To ensure that our approach can be generalized and is not specific to a particular cluster or machine, we used datasets from six HPC platforms available from Table 3.1.

Job runtime distributions change from one system to another, and building a unified runtime distribution model has proven to be a challenging task [67]. Nevertheless, the density of requested runtimes for all six traces shows one or two peaks at small values, showing that most jobs have relatively small processing time requirements (Figure 6.1, upper row). Other peaks also appear in some traces, with some containing a peak near the maximum allowed processing time. However, when comparing to the *actual* runtimes (Figure 6.1, bottom row), we can easily see the well-known mismatch between the requested and actual runtimes. We also notice that the six traces share an interesting similarity, with all actual runtime distributions having a sharp peak at the small values and a large tail towards longer execution times. These distributions indicate that we can always divide jobs into two classes: (i) small, encompassing the jobs at the peaks of the distributions, and (ii) large, comprising jobs at the tails of the distribution.

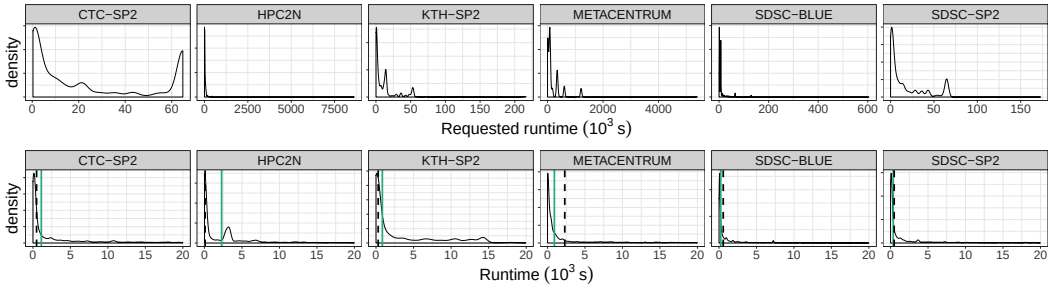


Fig. 6.1: Distribution of requested (upper row) and actual (bottom row) execution times of jobs for the six workload traces. (1) Note the scale/range difference on the X-axis, which indicates how the distribution of both variables are very different and shows that the requested runtime is a quite unreliable information. (2) The distribution of the actual runtime exhibits a sharp spike toward short jobs for all workloads. The green vertical line and the dashed black vertical line respectively represent the median value of the runtimes and the result of a clustering algorithm (Section 6.2) and allow to easily discriminate between "small" and "large" jobs.

Trace	Divider (s)	Small non premature (%)	Small premature (%)
CTC-SP2	1,114	17.11	32.89
HPC2N	2,287	27.63	22.38
KTH-SP2	847	15.37	34.63
MetaCentrum	915	3.94	46.07
SDSC-BLUE	229	0.36	49.70
SDSC-SP2	359	12.02	38.01

Tab. 6.1: Percentage of premature and non premature jobs: 22 to 49% of all jobs (Small premature jobs) requested their time allocation to be larger than the divider (5 to 20 minutes) but actually executed less than this

Characterizing Small and Large Jobs

From now on, we consider a job as small if its runtime is smaller than the median of the runtimes (green line on Figure 6.1), which we call the (*divider*), and we consider the job as large otherwise. For the sake of comparison, we also applied two clustering algorithms, DBSN [33] and EM [27], to divide the classes into two groups, which generated comparable divisions (dashed black line in Figure 6.1). Although divisions achieved by the median and clustering algorithms are not the same, they are relatively similar. As we generally aim for simplicity, we considered that the rolling median is sufficient to separate the initial peak from the rest of the distribution. Furthermore, having two classes with similar sizes simplifies the comparison in terms of fairness.

We further divide the small job class into two subclasses: (i) premature small jobs: short jobs that had requested runtimes larger than *divider* and which therefore terminate prematurely, and (ii) non-premature small jobs: those that also requested runtimes smaller than *divider* but managed to execute within this time bound. When

Tab. 6.2: Contribution of job size classes to platform resource usage: half of the jobs (Large) consume more than 98% of resources. Small jobs incur an unsignificant workload and running them first (provided they can properly be identified) should thus be harmless to large jobs

Trace	Large (%)	Small non premature (%)	Small premature (%)
CTC-SP2	98.37	1.34	0.29
HPC2N	99.35	0.50	0.15
KTH-SP2	99.59	0.36	0.05
MetaCentrum	99.33	0.20	0.47
SDSC-BLUE	99.32	0.57	0.11
SDSC-SP2	98.33	1.45	0.22

analyzing the traces from the six evaluated platforms, we notice that there is always a large fraction (22% to 50%) of premature jobs (Table 6.1). Premature small jobs have a wildly over-estimated processing time, causing them to wait longer for execution, which results in large slowdown values. This is problematic as it artificially inflates the overall slowdown. Moreover, we note that the total area[1] of these premature jobs represents a negligible fraction (less than 0.5%) of the total area (Table 6.2). If one could correctly detect these premature small jobs, we would obtain a significant reduction in the overall average slowdown in the platform. In the next section, we propose a method for performing this classification.

6.3 Job Size Classification

6.3.1 Classification Features

In this section, we detail the features we used for the classification and the reasoning behind our choices. A job is characterized by a set of *features*, which are pieces of information that can be used to predict the *class* of the jobs (Small or Large in our context). When a job is submitted, the scheduler has accesss to the following information: the id of its user, the dimensions of the jobs (requested number of processors, requested runtime), and the exact date of submission. We start by making two observations about the scheduling data: (i) it has been empirically observed that the runtime of a job is highly correlated with the user's submission history [64]; (ii) although there are clear regularities, the user identity is not sufficient to characterize job duration because the users often submit more than one *category* of job. For example, user_2 of the SDSC-SP2 submitted 796 jobs with 8 different requested node numbers and 11 different requested runtime values[2]. For a given user, the requested

[1] The area a_j of a job j is defined as its runtime multiplied by the number of resources it requested: $a_j = p_j * q_j$.

[2] Although $\tilde{p} \in \mathbb{R}^+$ may be arbitrary, in most HPC environments users tend to restrict themselves to a finite and small set of round values (e.g., 1 hour). This value can thus be treated as a factor.

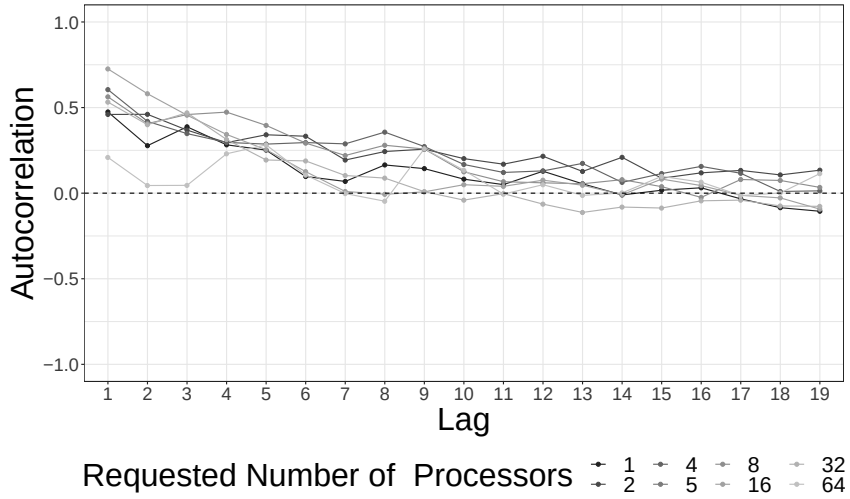


Fig. 6.2: Each category c allows to extract a series (ordered by submission dates) of actual runtimes $p^{(c)}$ for which we can estimate the autocorrelation coefficient for each lag value l as follows: $\rho_{p^{(c)}}(l) = \frac{\frac{1}{n-l}\sum_{i=1}^{n-l}\left(p_i^{(c)} - \mu_{p^{(c)}}\right)\cdot\left(p_{i+l}^{(c)} - \mu_{p^{(c)}}\right)}{\sigma^2_{p^{(c)}}}$, where $\mu_{p^{(c)}}$ and $\sigma_{p^{(c)}}$ are respectively the sample average and sample standard deviation of $p^{(c)}$. This autocorrelation coefficient lies in $[-1, 1]$ and indicates how strongly $p_i^{(c)}$ is correlated with $p_{i+l}^{(c)}$. The graph illustrates how the distribution of the autocorrelation coefficient evolves with the lag between the jobs that belong to the same category (u, q) of a specific user.

runtime of jobs, their size and the day when they are submitted are however a good indicator of the similarity of their actual runtime. We therefore introduce a category for each pair of factor (u, q), $(u, \tilde{p})$, and (u, d) and consider that two jobs from the same user u belong to *category* (u, q) (resp. $(u, \tilde{p})$, or (u, d)) if they have the same number of requested resources q (resp. same requested runtime $\tilde{p}$, or same submission day d).

To illustrate how useful such categories could be, let us come back to the user_2 of SDSC-SP2. Figure 6.2 illustrates how the autocorrelation coefficient decreases with the lag l: jobs that are very close in time have a relatively strong correlation and the first few lags have significantly higher correlation values than the rest. The notion of category therefore structures the job flow and can be used to perform online prediction of the jobs actual duration.

For each job, we extract all previous jobs that belong to the same categories and we derive the following features (Table 6.3):

Job features : The requested execution time and requested number of resources of the job.

Type	Feature	Description
Job features	$\tilde{p}_i$	user supplied runtime estimate
	q_i	user supplied number of resources
Temporal features	h	hour of the day
	D_{week}	day of the week
	d_{month}	day of the month
	m	Month
	w	Week
	Q	Quarter
Lag features	$C\tilde{p}_{i-1}$, $C\tilde{p}_{i-2}$, $C\tilde{p}_{i-3}$	Class of the previous, second to previous, third to previous jobs that belong to the same category $(u, \tilde{p})$
	Cq_{i-1},Cq_{i-2}, Cq_{i-3}	Class of the previous, second to previous, third to previous jobs that belong to the same category (u, q)
	Cd_{i-1},Cd_{i-2}, Cd_{i-3}	Class of the previous, second to previous, third to previous jobs that belong to the same category (u, d)
Aggregation features	$mean_{iq}$	percentage of jobs that belong to the same category $(u, \tilde{p})$ and are classified as small
	$mean_{i\tilde{p}}$	percentage of jobs that belong to the same category (u, q) and are classified as small
	$mean_{id}$	percentage of jobs that that belong to the same category (u, d) and are classified as small

Tab. 6.3: Features used for job classification

Temporal features : The hour of the day, the day of the week, the month and the quarter in which a job was submitted;

Lag features : contains the class (Small/Large) of the previous three jobs of the same category;

Aggregation feature : contains the percentage of jobs that belong to the same category and are classified as small. The goal of these features is to include the rest of the category's history. Although older jobs are less indicative of the class of the current job, they still contain information that is valuable to the learning process.

6.3.2 Classifier Training and Update

In an online scheduling context, the full information about the jobs is only known after their execution. Thus, the classical learning scheme, which consists in dividing the full dataset into a training and a testing set is not possible in this context. The learning process should adapt to the increasing amount of available data. We adopt a weekly training process illustrated in Figure 6.3:

- Training is performed at the end of every week.
- All the data gathered during the week is cleaned and processed to create the features presented in Table 6.3.1.

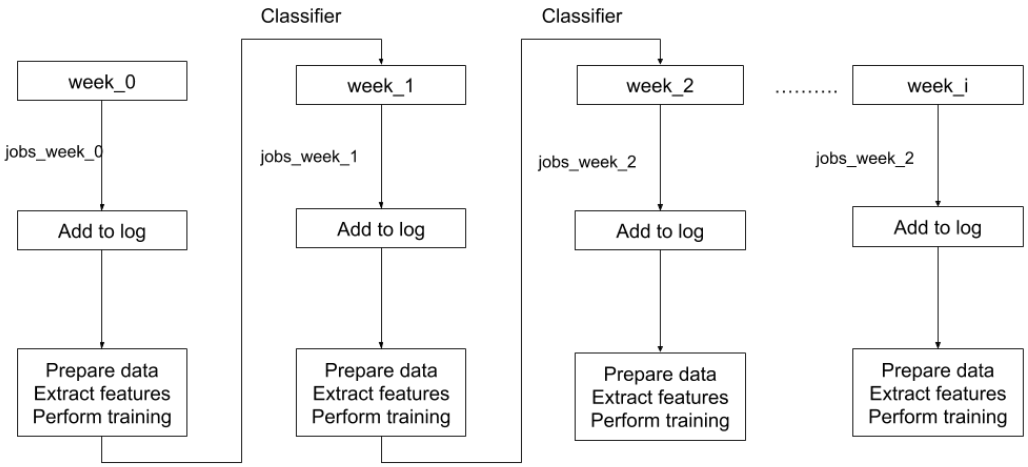


Fig. 6.3: Learning process: At the end of each week the new jobs are added to the dataset and a new training process is performed

- A new classifier is then trained and will be used during the next week.

We chose the period of one week because it seemed adequate. Indeed, retraining every day would be wasteful because in most cases a single day is not sufficient to generate enough new data to significantly change the output of the training. And retraining when the size of the new data reaches a certain threshold (e.g., training every 5000 new jobs) would cause the classifier to be updated at "unpredictable time", possibly in the middle of a workload spike, which is quite undesirable.

The data of the current week jobs are thus not added to the training data since we perform a weekly training. Note that, to simplify the implementation of our simulations and the tuning of the learning algorithms, we have decoupled the batch scheduling simulation from the learning and prediction mechanism. As a consequence, when performing predictions, the lag and the aggregation feature are also solely extracted from the jobs of previous weeks, which slightly decreases the reactivity of our predictions as they are built on information that is not the most up-to-date.

We use the Random Forest algorithm [17] to perform the classifications as it allows to easily combine numerical and categorical features. Random forests create multiple decision trees on randomly selected data samples, getting a prediction from each tree, and select the best solution by majority voting, which makes them particularly resilient to “outliers”.

6.3.3 Online Learning Quality

In this section we investigate the quality of the proposed online classification scheme and we explore some of the strengths and weaknesses while applying learning on scheduling data.

6.3.3.1 Accuracy, Precision and Recall

We measure the quality of our classifications using the three following indicators[3]:

- *Accuracy* is the ratio of correctly predicted observation over the total number of observations:

$$accuracy = \frac{TL + TS}{TL + TS + FL + FS} \tag{6.1}$$

- *Precision* is the ratio of correctly predicted small jobs to the total number of jobs that are predicted to be small:

$$precision = \frac{TS}{TS + FS} \tag{6.2}$$

- *Recall* is the ratio of correctly predicted small jobs to all observations in the small class:

$$recall = \frac{TS}{TS + FL} \tag{6.3}$$

As explained in section 6.3.2 the learning process is repeated at the end of every week and every week may thus have a different classification performance. Some week may suddenly behave very differently from the previous ones and thus, it is interesting to study the process in more detail. Figure 6.4 depicts the weekly quality of the learning through time for each platform, and from which several observations can be made.

Although the quality of the learning process varies between traces, which is expected because every trace has its own specific characteristics (number of jobs, number of users, frequency of job arrival, etc.) and may be more or less variable, for CTC-SP2, KTH-SP2, SDSC-SP2, and SDSC-BLUE all the weekly evaluation metrics maintain high values from the beginning to the end (with a few exceptions). For the other two traces; MetaCentrum and HPC2N, the results are not as stable as the others over time. For several weeks of MetaCentrum, the learning even seems to be very poor as

[3] TL –True Large– (resp. TS) represents the number of jobs correctly predicted as Large (resp. Small), while FL –False Large– (resp. FS) represents the number of jobs incorrectly classified.

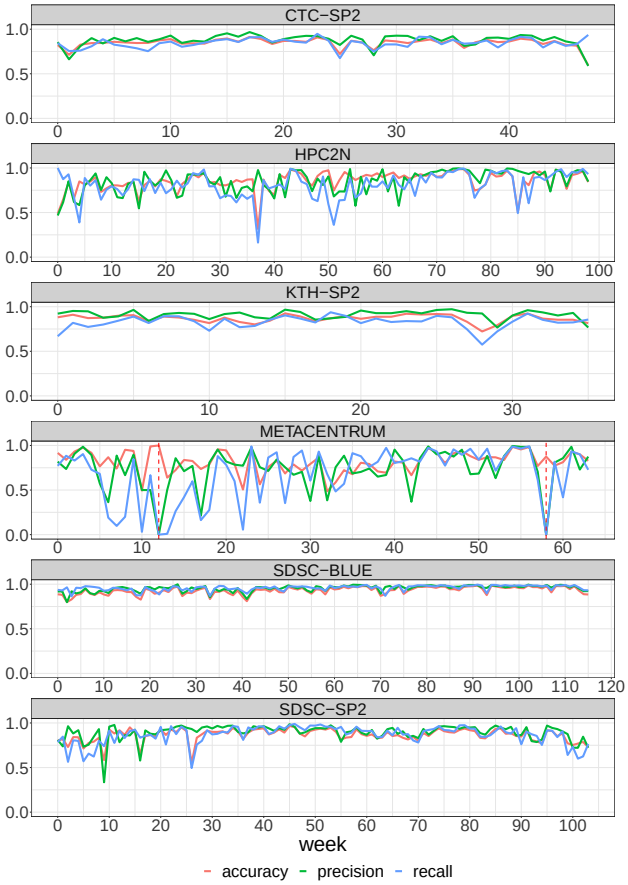


Fig. 6.4: Evolution of the quality of the learning for individual weeks

the precision and recall values are extremely low and even drop to 0 for some cases. A closer look at these weeks allows to understand why such low values occur.

Week 58 (identified by the rightmost red dashed line in the MetaCentrum trace of Figure 6.4) comprises 48 jobs, only 1 of which is a small job. The classifier in this instance made a single error and misclassified this single small job as large. This leads to a significant reduction in the accuracy value; 85% (due to the small number of jobs in that week), a 0% recall, and an undefined value for the precision ($TS = 0$ and $FS = 0$)

Week 12 (identified by the leftmost red dashed line in the MetaCentrum trace of Figure 6.4) comprises only 78 jobs but 0 small jobs and all the jobs are properly classified, which results in an accuracy of 100% but the values of the precision and recall (Equations (6.2) and (6.3)) cannot be computed because $TS = 0$ and $FS = 0$ and $FL = 0$.

The misclassification of a single job may thus significantly impact the learning metrics for a week comprising few jobs but it has a relatively low impact on a trace of more than 79,000 jobs (Table 3.1). These absolute fine grain weekly learning performance

Tab. 6.4: General classification performance: For each trace, we count the values of TS,FS,TL and FL for all the weeks then, we compute the general value of the accuracy, precision and recall

Trace	Accuracy(%) $TL+TS/Total$	Precision(%) $TS/(TS+FS)$	Recall(%) $TS/(TS+FL)$
CTC-SP2	85	82	86
HPC2N	90	87	89
KTH-SP2	86	79	90
SDSC-BLUE	80	78	83
SDSC-SP2	87	89	91
MetaCentrum	85	83	87

Tab. 6.5: Classification error per trace: 7–11% of Large jobs are misclassified while 4–8% of Small jobs are misclassified

Trace	False Large(%) $FL/(FL+TL)$	False Small(%) $FS/(FS+TS)$
CTC-SP2	8.16	6.29
HPC2N	7.59	5.05
KTH-SP2	9.27	4.03
MetaCentrum	8.43	6.36
SDSC_BLUE	11.29	8.52
SDSC-SP2	7.20	5.85

indicators should be interpreted with care, especially because some weeks with a relatively low workload or missing classes (e.g., week 30 of MetaCentrum) often make the learning look artificially inefficient.

Fortunately, the overall (i.e., when the ratios of Equations (6.1) to (6.3) are not broken down per week) performance of the classifier is particularly good. Table 6.4 shows the performance of all the jobs regardless of the week. The overall recall, precision and accuracy are always above 78% and even generally around 90%.

6.3.3.2 Classification Errors

During the training phase, the goal is to reduce prediction errors as much as possible. However, incorrect predictions and errors are an unavoidable part of any learning process. The two types of prediction errors are related to the number of false large (FL) and false small (FS) jobs. Table 6.5 shows the percentage of classification errors of each type. Although the values vary from one trace to another, the percentage of FS tends to be smaller than that of FL. We note that the percentages of FS jobs we obtained are comparable to the values presented in [49] where the authors use a method to specifically manage the problem of underestimation in runtimes predictions and reported an FS rate of $\approx$ 5–8%. We do not aim at designing a perfect

classifier nor at fine-tuning the learning algorithm parameters so we argue that such classification error is representative of what can be achieved with a reasonable effort. Although the predicted class is a precious qualitative information, a scheduler should thus be aware of potential prediction errors and manage them accordingly.

6.3.4 Feature importance analysis

In this section, we provide insight on the importance and the stability of each of the features during the learning process. We use the Gini impurity measure [71], which estimates the probability of miss-classifying an observation, to evaluate the importance of the constructed features. Note that it is also the measure used during the training phase of our classification. For an in-depth review of feature importance analysis, we refer the reader to [93]. This measure provides us for each week with a weight representing the importance of each feature in the classification.

Figure 6.5 represents the distribution (summarized with a box-plot) of the weekly weights of each feature for each trace. These box-plots reveal several useful information on the learning process.

- First, we observe similarities between the results of various traces. (i) The requested execution time appears to be the most important feature for the majority of the traces (expect METACENTRUM). This is expected since a sizable portion of the jobs belong to the class *Small* because the users themselves requested a processing time that is smaller than the divider (see table 6.1). (ii) For the lag features, the first lag always holds most of the weight followed by the second and the third lag respectively. (iii) Aggregation features are about as important as the first lag features. (iv) Temporal features generally hold the lowest weights for all the traces.
- This supports the idea that performing a single unified learning process for all traces like the ones presented in [41] and [21] is reasonable and can yield good results. However, there are also clear differences between the traces. For example, the requested time, the feature with the highest weight, holds different importance scores from one trace to another and it is outweighed by the lag features for in the case of the METACENTRUM trace. Also, For some traces (CTC-SP2 and KTH-SP2), we observe a very small difference between the weeks. For others (METACENTRUM and HPC2N), there is a noticeable difference between the weeks, which can be explained that this workload seems harder to predict than the others and supports the importance of constantly updating the learning process.

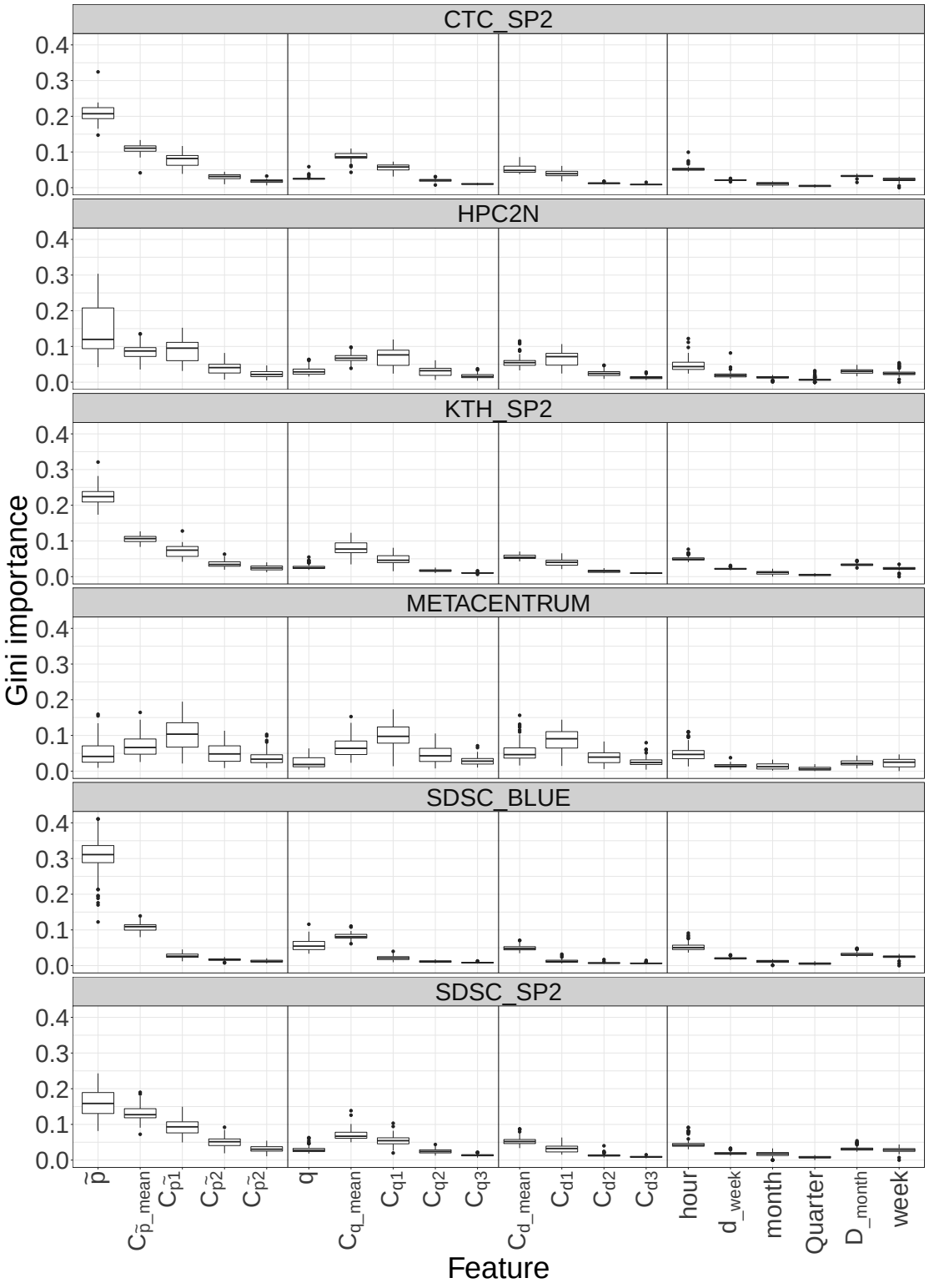


Fig. 6.5: Importance of individual features during the weekly learning process. The larger the weights, the more important the feature in the classification. Weights are normalized such that their sum equals 1.

6.4 Proposal

As indicated by the previous trace analysis, small jobs represent a negligible fraction of the total load of the platform (Table 6.2) but are quite numerous (Table 6.1) and often wildly over-estimate the runtime they request. Scheduling them using a policy based on this estimation leads to particularly poor slowdown and to an overall poor performance of the system. Fortunately, the learning algorithm presented in Section 6.3 allows to efficiently distinguish between small and large jobs. To improve the mean bounded slowdown, we propose that any job classified as small is executed with a higher priority than those classified as large. Specific care must be taken though to avoid starvation and to deal with classification errors. This section describes how this was done and how we evaluated our proposal.

We aimed our work to be as transparent and reproducible as possible [83]. We provide a snapshot of the workflow used throughout this work[4], which includes a nix [29] file that describes all the software dependencies and an R notebook that allows reproducing all the figures.

We consider HPC platforms as a collection of homogeneous resources, with jobs stored in a centralized waiting queue, following the submission described in the workload logs. We implemented all simulations using Batsim [31], a simulator based on SimGrid [22] that allows us to observe the behavior of scheduling algorithms under different conditions. We evaluate our method using the six traces presented in Table 3.1 and four scheduling policies presented in the following section.

6.4.1 Scheduling Policies

We considered three of the basic scheduling policies introduced in Section 3.1.4; FCFS, SPF, and SAF. And one extra popular policy known as WFP.
WFP: is a scheduling policy adopted by the Argonne National Labs [86] and is given by: $WFP_j = (\frac{wait_j}{\tilde{p_j}})^3 * q_j$. This policy attempts to strike a balance between the number requested resources, the requested runtime and the waiting time of jobs. It puts emphasis on the number of requested resources while preventing small jobs from waiting too long in the queue.

We chose FCFS and WFP because several existing HPC systems use them. SAF and SPF are less common, mostly because they are perceived as too risky since they could potentially induce job starvation. *Starvation* occurs when a job waits for an indefinite or a very long time in the queue. However, some studies [21] show that SAF and

[4] `https://gitlab.inria.fr/szrigui/job_classification/`

SPF provide better results on performance metrics in almost all cases. Furthermore, one can prevent starvation by putting a *threshold* on the waiting time [65]. When the waiting time of a job surpasses the threshold, the scheduler transfers the job to the head of the queue. In [65] the authors perform a detailed analysis of the thresholding mechanism and of its impact.

We implemented the four aforementioned policies in conjunction with the EASY backfilling heuristic and the thresholding/starvation prevention mechanism. The scheduler orders and executes the jobs following the order set by the chosen policy. When it reaches a job that cannot start immediately, it makes a reservation for that job. The scheduler then allows the next tasks to skip the queue if they do not delay the initial reservation.

6.4.2 Learning and Scheduling Algorithms

When a user submits a job for execution, the classifier uses the job features to assign it to the small or large classes, represented by queues Q_{small} and Q, respectively. In the first week of the trace, since the classifier does not have prior data to learn the classification task, it classifies all jobs as large and does not behave differently from a classical policy. After that, we update the classifier at the beginning of every week, with data from all previous weeks as explained in Section 6.3.

The resource manager calls the scheduler whenever a job finishes its execution, and computational resources become available. The scheduler then sorts the two queues, Q and Q_{small}, independently, according to a chosen policy (FCFS, SAF, SPF, or SQF), and merge them in a single queue Q_{total}, with the jobs belonging to the small class first. Finally, resource allocation is done using the EASY heuristic, as shown in Algorithm 1.

Algorithm 3: Perform scheduling

Input : Queue of large jobs Q
Queue of small jobs Q_{small}
Scheduling policy $Policy$ (FCFS|WFP|...)

1 Order Q according to $Policy$
2 Order Q_{small} according to $Policy$
3 $Q_{total} =$ **concat**(Q_{small}, Q) *# small jobs are always put in the head of the final queue*
4 **EASY**(Q_{total}) *# Schedule the jobs in the final queue using the EASY heuristic*

The only additional relevant overhead compared to the basic EASY scheduling heuristic is the cost of updating the classifier. The update includes finding the median execution time over the workload log of the previous week and training

Algorithm 4: Kill False Small jobs

```
1 Q = {} # queue of large jobs
2 Q_small = {} # queue of small jobs
3 job_counter = 0  # number of submitted jobs
4 while Running do
5     # go through all jobs currently running
6     if job_j.class == "small" & job_j.runtime > divider then
7         kill(job_j)
8         Q_small.remove(job_j) Q.add(job_j)
```

the classifier using the pairs $(features, jobclass)$. The full execution of this procedure takes only a few seconds and occurs only at the end of every week. Moreover, it runs independently from the scheduler, without blocking it.

6.4.3 Dealing with Classification Errors

As explained in Section 6.3.3.2, no classifier is perfect and some jobs will inevitably be wrongly classified. False Small jobs are large jobs that were wrongly classified as small. This is similar to runtime underestimation in the case of exact runtimes prediction. Although the resource manager may allow these jobs to execute until completion, it can significantly impact performance in some cases, e.g, when there are many True Small jobs following the misclassified large job. This type of misclassification is arguably more dangerous than False Large: if a small job is classified as large it will be delayed but it will not impact the waiting time of other jobs.

One way to correct this problem is to kill false small jobs. When the execution time of a job classified as small exceeds the divider value, it is killed and put back to the waiting queue as a large job. To ensure that the killing and restart process happen without problems, we consider that jobs are **idempotent**. Formally an idempotent operation is defined as an operation that can be applied multiple times without changing the result from the initial application. In this context, we consider a job to be idempotent if it can be killed and restarted without changing the final execution outcome.

The scheduler periodically goes through the list of running jobs (Algorithm 4). If it detects a job classified as small and has been executing for a period longer than the divider value, it kills the job and classifies it as large.

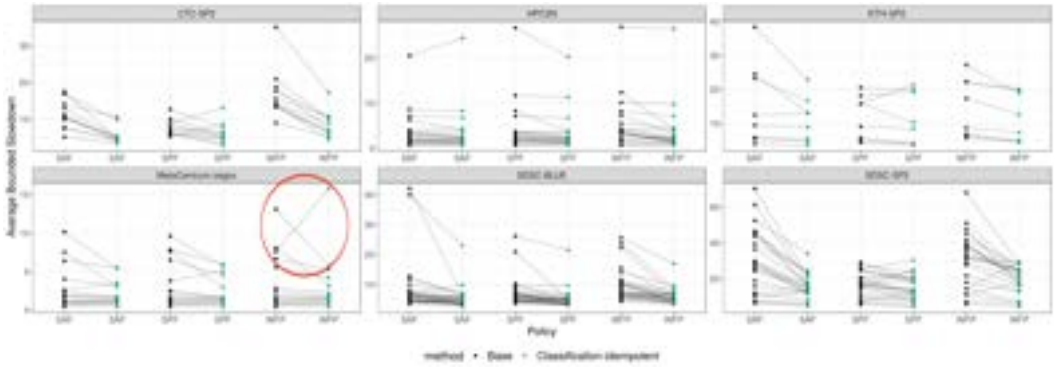


Fig. 6.6: Monthly average bounded slowdown. Each line links the values from the same month when using the base and classification-idempotent schedulers.

6.5 Experimental Results

In Section 6.3, we presented the job size classifier and showed its accuracy from a pure learning perspective. However, achieving a high-quality classification is not our final goal. In the scheduling context, the effectiveness of an approach is measured by how much it improves the end-to-end performance metrics, such as the average bounded slowdown.

6.5.1 Overall Impact on Scheduling Performance

We evaluate the impact of the cumulative bounded slowdown when applying the EASY-backfilling with the four scheduling policies (FCFS, WFP, SAF, and SPF). Figure 6.7 shows the results for the scenarios with the job size classification and job-killing mechanism (in cyan) and without them (in black).

Comparing the curves for the four basic scheduling policies, we note that SPF and SAF generate the lowest cumulative slowdown in all platforms. WFP has cumulative values close to SPF and SAF, while FCFS has the worst values by a large margin in all cases. These results are consistent with previous comparisons of scheduling policies (Chapter 5).

Applying the job size classification reduced the cumulative slowdown values in all scenarios, with the improvement depending on the trace and scheduling policy. For FCFS, we observed substantial improvements for all the six traces, ranging between 33% to 79%. For the other policies, we observed smaller, albeit consistent, improvements in performance, ranging from 3% to 32% for SPF and 10% to 51% for SAF. We explore these results further in Section 6.5.5.

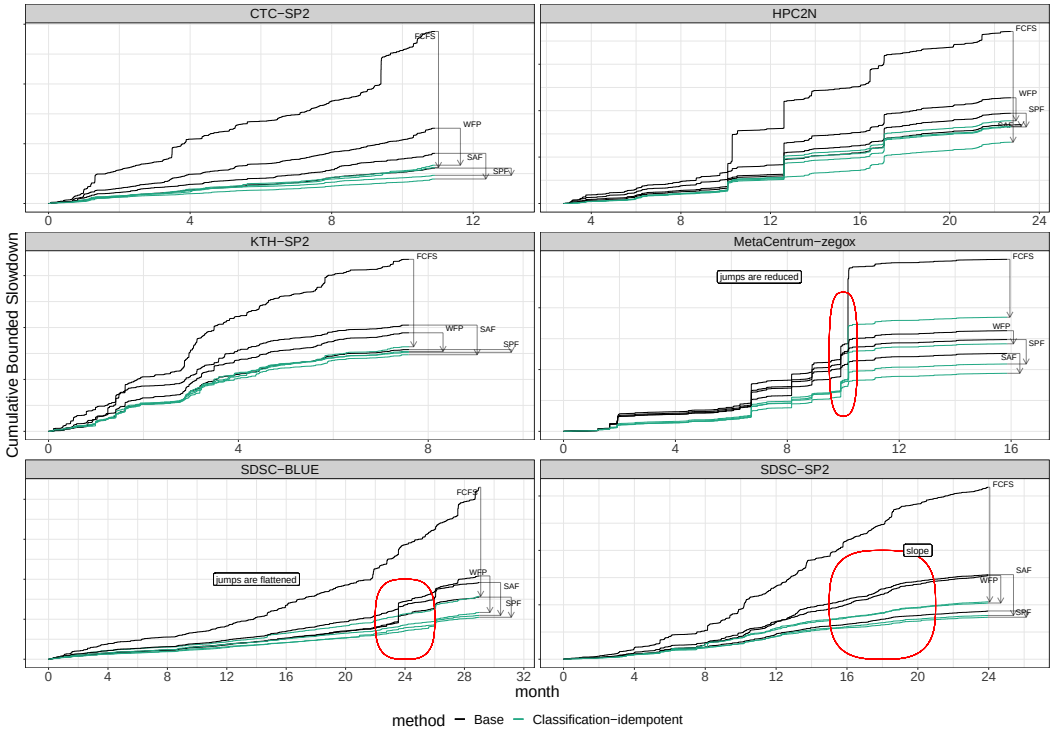


Fig. 6.7: Evolution of the Cumulative Bounded Slowdown for the six platforms, using the base policies (black) and the same policies augmented with job size classification and idempotence (cyan). The cumulative bounded slowdown is always such that SAF≈SPF<WFP<FCFS, which is expected as prioritizing small jobs is known to optimize the average slowdown whereas FCFS rather bounds the largest waiting time. Since these heuristics solely rely on the requested runtime, they cannot be very efficient. Activating our classification-based prioritization systematically and significantly improves the performance of all heuristics at any point in time and not simply at the end of the evaluation period. In steady state (see SDSC-SP2), it is clear that the cumulative bounded slowdown increases more slowly when our classification-based mechanism is activated. It may happen that burst of jobs are submitted and incur sudden and large jumps in the cumulative bounded slowdown. These jumps are always significantly reduced (see Megacentrum-zegox) with our mechanism and even sometimes completely avoided (see SDSC-BLUE).

The cumulative slowdown increases most of the time smoothly, with some sharp rises. The slower increments occur during lightly or moderately loaded periods, in which we see steady increments in the gap between the scenarios with and without job size classification. The jumps are the result of high load periods and seem unavoidable, as they occur with all policies. However, regardless of the policy and the trace used, our method always results in smaller cumulative slowdowns.

Since FCFS performed poorly compared to other policies, we decided to exclude it from the subsequent analysis. However, we note that the observations in the next sections also apply to FCFS.

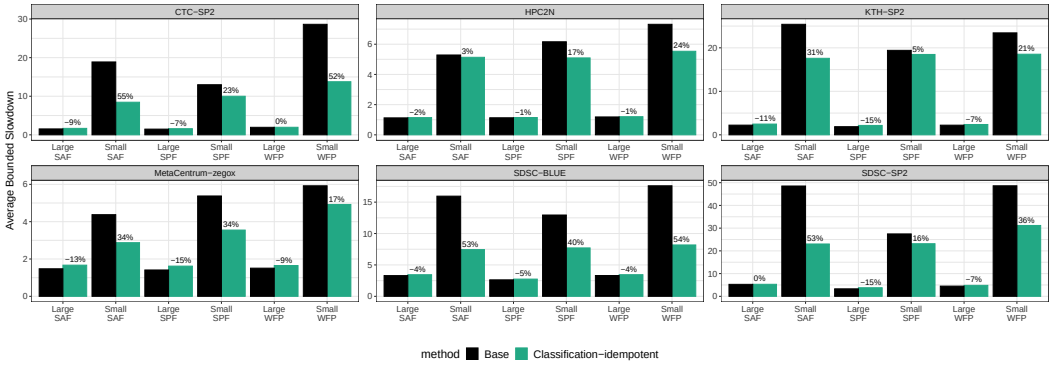


Fig. 6.8: Average bounded slowdown for small and large jobs, using the four base policies and the corresponding classification-idempotent schedulers. Breaking down the average bounded slowdown between small and large jobs allows to evaluate how both classes benefit from the classification and whether one is unfairly treated compared to the other. The benefit for the Small job class is huge and can go up to 55% while the loss for the Large job class never exceeds 15% (the higher losses always occur in trace/policies with extremely small base slowdown). The difference for Large jobs is therefore negligible and would be barely noticeable by users. Last, note that, although there are visible differences between the base policies (SAF, SPF, WFP, in black), they tend to vanish whenever using our classification (in green).

6.5.2 Impacts on Individual Months

The evolution of the cumulative bounded slowdown over long periods, although informative, can mask important details about the behavior of a scheduler at a smaller time scale, such as individual weeks or months. Ideally, a good scheduler should provide improvements that are somewhat equitably distributed throughout the evaluation period.

We investigate the effects of our approach on individual months in Figure 6.6. Each pair of connected points represents the average bounded slowdown of a single month from the full workload execution, for the *base* and *classification-idempotent* cases. We note a reduction in the slowdown in most cases, with a decrease proportional to the base value. There are a few months where our approach seems to degrade performance substantially, such as in MetaCentrum/WFP. These are artifacts that emerge from splitting the results into one month periods, where workloads “spills” from one month to the other during periods of very high load. Overall, the results show that improvements are fairly distributed between months, even for the clusters that have large jumps in the cumulative slowdown, such as MetaCentrum and HPC2N.

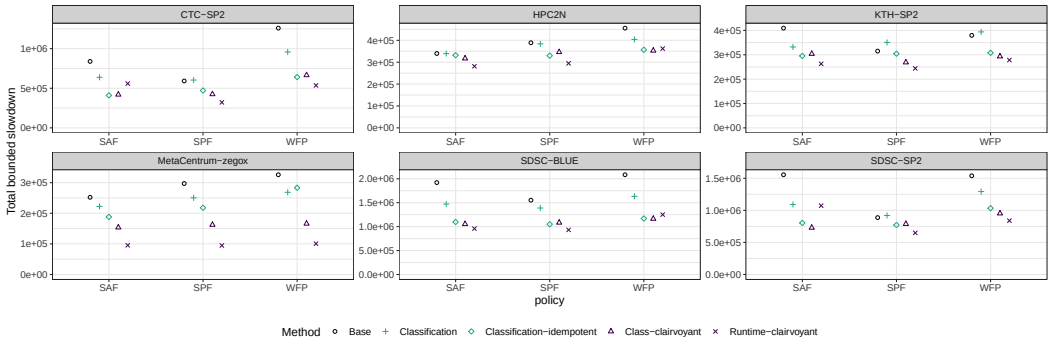


Fig. 6.9: Total accumulated bounded slowdown for the base schedulers (base), schedulers with perfect classification (class-clairvoyant), schedulers with classification and job-killing mechanism (classification-idempotent), and schedulers with perfect execution time information (runtimes-clairvoyant). Regardless the heuristic, it is interesting to note that, in general, Base $\gtrapprox$ Classification $>$ Classification-Idempotant $\gtrapprox$ Class-clairvoyant $>$ Runtime-clairvoyant, which is consistant with the fact that more accurate information allow to produce better schedules

6.5.3 Impact of Small Job Prioritization over Large Jobs

Our algorithm reduces the overall bounded slowdown by prioritizing small jobs. This mechanism naturally raises one crucial question: What is the impact of favoring small jobs over the jobs classified as large?

We compute the average bounded slowdown of the jobs for each of the two classes (Figure 6.8). As expected, the small jobs have the most substantial reductions in the average slowdowns. The extent of the reduction depends on the platform and policy and is mostly proportional to the improvements in the cumulative bounded slowdown, shown in Figure 6.7. More importantly, there is only a small increase in the average slowdown of large jobs.

The use of the job size classifier results in substantial improvements for small jobs, with little or no impact on large jobs. Consequently, we argue that there are no perceivable hidden costs for large jobs when prioritizing small jobs.

6.5.4 Impact of the Safeguard Mechanism

Assigning a large job to the small class can cause an overall increase in the average bounded slowdown of other jobs since it occupies resources for an extended period. We prevent this problem by killing the job when it reaches the job size divider value. But we cannot apply it for non-idempotent jobs. A subsequent question that arises is: can we still improve the performance if we allow miss-classified jobs to run until completion?

We compared the cumulative bounded slowdown values at the end of the full workload trace simulations, for the six platforms, for three scenarios: (i) *base*, (ii) *classification-idempotent*, where we kill false small jobs, and (iii) *classification*, where we use classification without job-killing.

Preventing job-killing reduces the effectiveness of the classification in almost all scenarios (Figure 6.9). We note, however, that *classification* without job-killing still managed to improve the total slowdown for most cases, but to a worse extent than *classification-idempotent*. The exceptions are the combinations where the *classification-idempotent* only managed to improve results by a small margin. In these cases, the *classification* without job-killing did not improve or caused a negligible degradation in performance. Removing the safeguard mechanism reduces the effectiveness of our method without rendering it completely useless. Note that only False Small jobs are restarted, which corresponds to 2-4% of all jobs at most (See Table 6.5).

6.5.5 Comparison with Clairvoyant Schedulers

Finally, we evaluate the hypothetical optimum obtainable by a clairvoyant scheduler that knows the actual execution times of each job in advance. We compare three strategies that build the base policies (SPF, SAF, and WFP): (i) *runtimes-clairvoyant*, where the scheduling heuristic is provided with the actual p_j, instead of the requested processing times $\tilde{p}_j$, (ii) *class-clairvoyant*, where the scheduler is indicated which class the jobs belong to (i.e., as if a perfect job class classification was achieved), and (iii) *classification-idempotent*, the method we propose and which only uses estimated execution times. Although the clairvoyant versions cannot occur in practice, they provide us with a useful benchmark on the achievable improvements.

Using the *classification-idempotent* results in improvements comparable to the *class-clairvoyant* (Figure 6.9), except for MetaCentrum. This result indicates that the job-killing mechanism is effective in counteracting the misclassifications and that the overhead of job-killing has a small impact on performance. Moreover, it shows that our strategy of combining classification with job-killing is already very efficient and has little room for further improvements.

The two clairvoyant strategies, *class-clairvoyant* and *runtimes-clairvoyant*, also have comparable performance, with slightly better results when using *runtimes-clairvoyant*. This result shows that a simple classification in two categories is, in most cases, sufficient to obtain important improvements for the bounded slowdown metric. It indicates that trying to predict job execution time accurately with elaborate

regression techniques will not bring large improvements over the use of a simpler binary job size classification.

The most notable exception to the conclusions above is the MetaCentrum trace. We observe consistent improvements when moving from *base* to *classification-idempotent*, *class-clairvoyant*, and *runtimes-clairvoyant*. For this particular trace, there were several jumps in the cumulative bounded slowdown (Figure 6.7), caused by abnormally high loads. In these situations, a perfect knowledge of execution times appears to have a larger impact on scheduling performance.

Finally, we look at the cases were *class-clairvoyant* provided minor improvement: SDSC-SP2/SPF and KTH-SP2/SPF. In Figure 6.9, we can see that even with full knowledge, there were no significant improvements. *class-clairvoyant* only improved over *base* SPF by 10% and 13% for SDSC-SP2 and KTH-SP2 respectively indicating that, for these two traces, SPF was already a very good policy.

6.6 Conclusions and Discussion

In this work, we showed that a coarse classification of jobs into small and large is sufficient to improve scheduling performance. A simple safeguard mechanism that kills large jobs misclassified as small is important to prevent these jobs from unduly delaying others. Since the misclassification is detected very early, when the job execution time reaches the divider value between classes, which is never more than a few minutes, it results in a small overhead over the average slowdown metrics. We obtained improvements in scheduling performance for all combinations of six workload traces and four scheduling policies evaluated (see Figure 6.7). Moreover, in most scenarios, we managed to obtain improvements in scheduling performance comparable to that of clairvoyant schedulers with perfect knowledge of job execution times. Finally, we showed that our performance not unfair (Figure 6.8) in the sense that although the performance gain mostly targets small jobs (which are prioritized), it is not detrimental to large jobs.

We claim that in this context, using a classification approach is more effective than using regression for improving scheduling performance. Compared to regression-based techniques, our approach has two major advantages: (i) a two-class classification task is easier to learn than regression, requiring less training data, and (ii) misclassification of large jobs as small is detected very quickly during execution, opposed to regression, where underestimates are evident only after the job executed for the entire actual period. To substantiate this claim, we can compare the improvements obtained by our approach with two regression-based approaches: the relatively

Tab. 6.6: Improvement (in %) over EASY-FCFS using regression ([90] and [42]) and classification (SPF-CI and FCFS-CI). Values between brackets correspond to the evaluation performed by the original authors whose methodology may slightly differ from ours. Our classification based approach systematically and significantly improves upon the previous strategies, regardless of the the base scheduling heuristic (FCFS or SPF)

			Classification-Idempotent	
	EASY++ [90]	Gaussier et. al. [42]	FCFS-CI	SPF-CI
KTH-SP2	23 [36]	[44]	50	59
CTC-SP2	1 [37]	[59]	79	85
SDSC-BLUE	38 [47]	[05]	63	74
SDSC-SP2	32 [29]	[15]	66	75

simple EASY++ [90], which replaces user-provided runtimes estimates by the average runtime of the two previous jobs from the same user, and the one proposed by Gaussier *et al.* [42], which relies on more elaborate regression technique using an asymmetrical loss function. Both works used the workload traces from SDSC-BLUE, SDSC-SP2, KTH-SP2, and CTC-SP2 and reported improvements over the base EASY-backfilling with FCFS ordering policy (see Table 6.6). Although there are a few methodological differences (simulation technique, trace cleanups, etc.) between our evaluations, our classification approach combined with FCFS reduced the cumulative bounded slowdown by 50–79%, compared to 29–47% from EASY++, and 5–59% from Gaussier *et al.*. Relying on SPF instead of FCFS allows decreasing the cumulative bounded slowdown even further (59–85%), with most of the gain provided by the classification mechanism. Finally, our mechanism greatly reduces the performance difference between heuristics (without classification, FCFS is significantly worse than SPF or SAF) without loosing most of the fundamental properties that make FCFS an appealing option: its simplicity in terms of explainability to users and its predictable behavior. Consequently, we believe that using the proposed scheme of job size classification is more appropriate for deployment in real HPC platforms than regression-based approaches.

Since the gains we report are particularly substantial, one may wonder whether further gain can still be expected or not. For most of the traces we studied, not only the learning is very good (Table 6.4) but there is almost no difference between the performance of our classification-based scheme and the one a fully informed (clairvoyant) approach would give (Figure 6.9), which means that very little gain may be expected from the learning perspective.

Note that a potential improvement perspective can be foreseen by closely inspecting the only trace (METACENTRUM) where the weekly performance of the learning did not seem stable (Figure 6.4). This trace exhibits particularly irregular job submissions with burst of jobs that lead to sudden jumps in the cumulative bounded slowdown

(Figure 6.7). We suspect that our batched (weekly) learning strategy may not be able to adapt well to such rapidly changing situations. Online monitoring of the classification error may then be a good indication that the situation has evolved and that the learning should quickly be updated.

7 Energy profiling and classification

7.1 Introduction

With the ever-increasing size and complexity of data centers, power consumption has become a dominating factor in the total cost of ownership of supercomputers. Exorbitant power consumption does not only impact the budge of supercomputer operations, but it also translates to a high carbon footprint which has a detrimental impact on the environment. All these factors make efficient power management a necessity, especially with the anticipated rise in the operating coast that will come with exascale machines.

In this chapter, we focus on energy consumption. More precisely the energy consumption of individual applications. We use RAPL to collect the energy measurements and we couple them with basic jobs information similar to the ones used in all the previous chapters (job resources, runtimes, allocations ...). We observe that the energy profile of HPC application is not as chaotic and unpredictable as initially perceived and that certain patterns are common between the jobs. Such can be observed quantified and even predicted. In this chapter, we offer:

- A detailed characterization of the energy behavior of an HPC complete workload using "simple" non-intrusive tools such as RAPL and built-in performance counters.
- An insight into the energy behavior of HPC jobs. We observe that such behavior is, for the most part, predictable. More precisely we classify the jobs into constant (energy consumption remain constant thought the execution), periodic (alternating phases of low and high consumption), and non-stationary (no particular pattern can be discerned)
- A classification tree based on a series of statistical tests that automatically detects the energy profile of any job.

Such knowledge can play a major role in optimizing energy consumptions. It will not only allow us to predict the general energy consumed by the workload but also to detect the different consumption phases of an application and implement energy-aware scheduling and allocation policies.

7.2 Data sources

7.2.1 Machines

For this work, we use the GRIDCAD Infrastructure [2]. We chose to focus on a single cluster to prepare the pipeline for the data processing step. We use Dahu Cluster, more precisely we extract all of our data from 40 machines (Dahu_33 to Dahu_72). The 40 machines are homogeneous and have the following characteristics: Dell C6420 bi-xeon SKL Gold 6130 (16 cores, 2.1Ghz).
The Dahu cluster is managed by the OAR Resource and job Manager (RJMS) and monitored by Colmet. We introduce the two tools in the following sections.

7.2.2 OAR

OAR [3] is an RJMS for HPC clusters and distributed platforms. Its main tasks include job scheduling, reservation management, resource allocations, resources visualization ...
OAR give the user the option to submit their jobs as one of the following types:

- parallel jobs: are traditional HPC jobs that request several cores
- serial: jobs that request and execute on a single core
- interactive: jobs where the user can interact with his allocated machines, via a shell interface. generally used for debugging purposes.
- best-effort: jobs with the lowest constraint and lowest priority, can be pre-empted, can allocate any number of cores, generally CPU intensive.

OAR logs the execution history of all the submitted jobs within its attached database. In this work, we use this functionality to extract job-related information such as the request as submitted by the users, job dimensions, the resources that were allocated to the job,...

7.2.3 Colmet

Colmet is a resource monitoring and collection tool used by the Ciment-Dahu cluster. Its major task is to monitor the resources used by OAR jobs and identify job profiles in relation to resources.
Colmet relies on the Linux taskstats [4] accounting feature coupled with the cgroup [5] kernel isolation mechanism to retrieve consumption counters at low-cost.

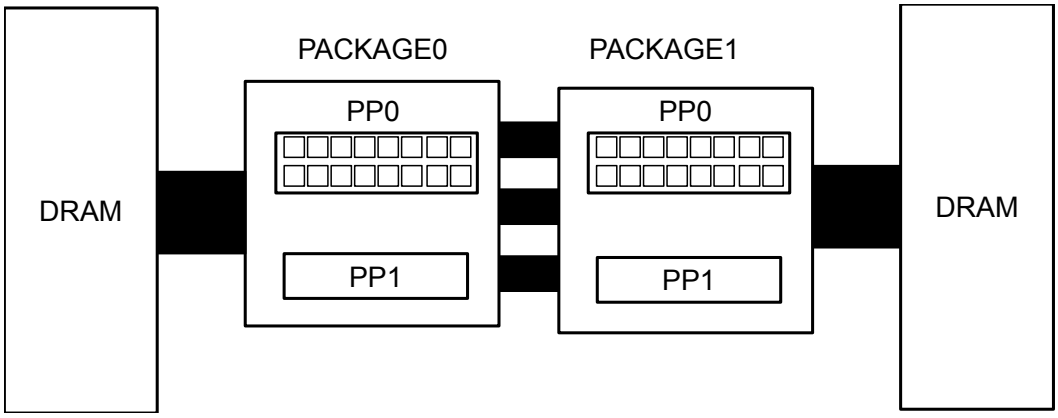


Fig. 7.1: Overview of the Intel RAPL architecture of a dual-socket system

Collected data is then stored on a dedicated node in a file structured in Hierarchical Data Format v5 (HDF5). Among the various counters Colmet monitors, we focus on the RAPL counter.

7.2.4 RAPL

RAPL was introduced in Section 2.3.2. In this section we give more details On its packages and architecture.
All the machines we used have a dual-socket system, So we limit this section to detailing RAPL for the dual-socket architecture. Figure 7.1 depicts the various component.

- RAPL has a hierarchical architecture. It supports an independent package for each processor.
- A package measures the energy consumption of the whole processor. In Figure 7.1 We identify two packages; PACKAGE0, PACKAGE1.
- Each package contains two power plane (PP) and has an attached memory (DRAM).
- The power Plan 0 (PP0) domain monitors the collective Energy consumptions for the processing cores (16 cores in this care).
- The power Plane 1 (PP1) domain monitors the energy consumption of integrated graphical processors (GPU). It is only active if the monitored machines include GPUs (not used in this work).
- The DRAM domain measures the energy consumption to the attached memory.
- The measurement used throughout this work includes the PP0 and The DRAM of both processors of every machine.

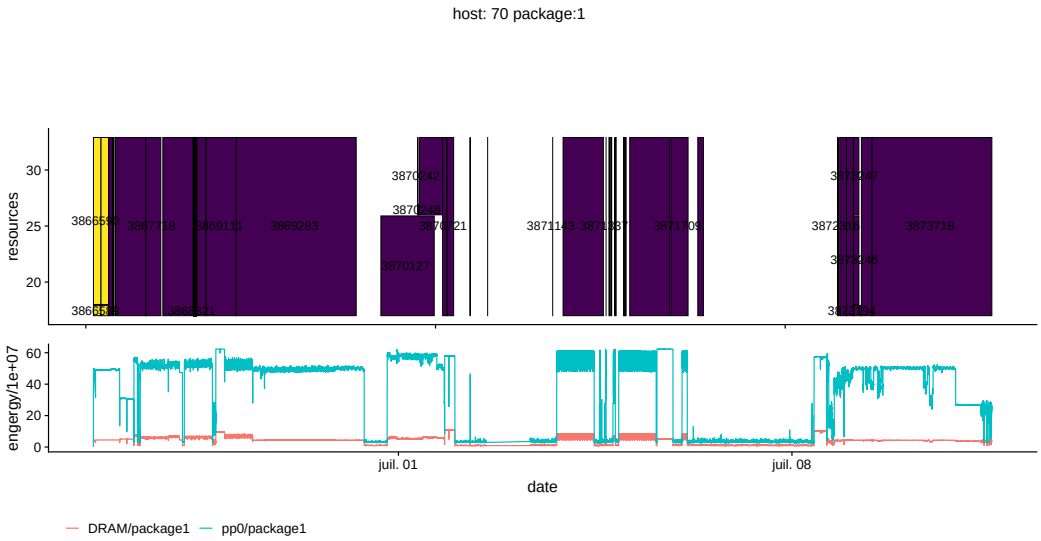


Fig. 7.2: Execution history of a single processor

7.3 Combining the different data sources

Basic job descriptions are stored in the database of the OAR RJMS, and RAPL energy measurements are logged via the Colmet monitoring tool (within the HDF5 format). By Combining and synchronizing data from the two sources, we can visualize the execution history of any monitored machine. Figure 7.2 presents the execution log of processor 1 of the Dahu 70 machine. The upper figure is the Gantt chart of the jobs that were executed during a certain period and the lower figure shows the energy consumption as logged by RAPL for the corresponding period.
Several Remarque can be made:

- There is a clear correlation between the jobs and the changes in the energy patterns.
- jobs have different energy profiles.
- Just because a job uses a full processor, it does not mean that it uses the maximum amount of energy allowed.
- the length of the jobs varies greatly. from few seconds to multiple weeks
- the consumption of the pp0 and DRAM are heavily correlated.
- the pp0 consumption is approximately one order of magnitude larger than the DRAM consumption.
- **Energy Profile:** An energy profile of a job corresponds to all the RAPL measurements that were taken during the job executions. Jobs that use a full

processor have an individual profile. Jobs that share a processor have a shared profile that corresponds to the total energy used by the processor for the duration of the execution. In this work, we also refer to the energy profile of a job by the term signal or series (a more detailed explanation of the reason will be given in Section 7.4.2)

- Since RAPL gives measurements for one full processor, jobs that don't use a full processor do not have a specific energy profile. They share the same profile with other jobs that are running at the same time.

7.4 Preprocessing and job distribution

As we are working with real word logs, We are bound to encounters anomalies and inconsistencies in the data we collect. In this section, we perform several filtering steps to extract clean profiles which will be used letter in this work.

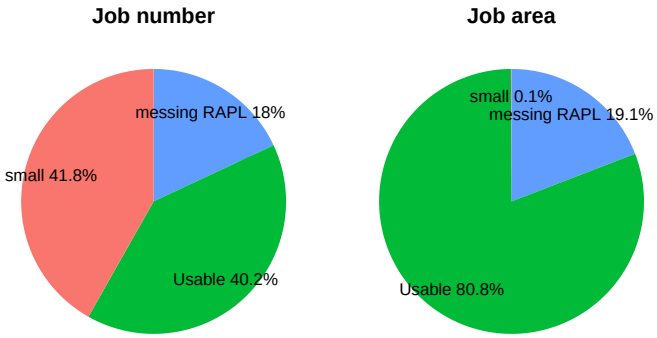
7.4.1 Sample

For this work, we use only a sample of the available data; about 16 days, from June 25, 2019, to July 11, 2019. In this period, 21,885 unique jobs we submitted and executed on the Dahu platform.

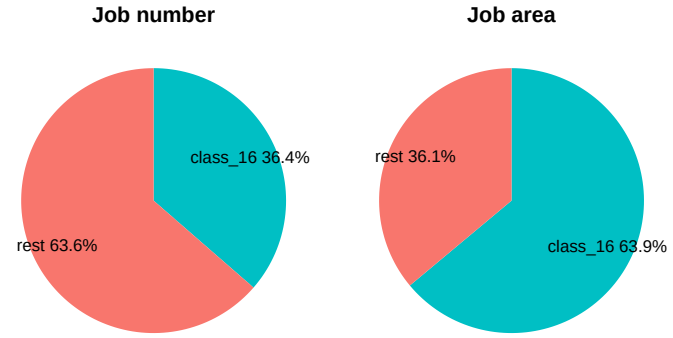
First, we remove jobs with no usable energy profile. They include jobs that are too short (less than 1 minute) and jobs with messing energy measurements. Figure 7.3 show the percentage of discarded jobs.

- Very short jobs (less than one minute of execution time): The number of RAPL observations is too small to make any use of. We cannot extract any meaningful pattern (even if we can the number of observation is too small to be of any use in this study). Also, they have no impact on scheduling performance. Thus, we discard them. Figure 7.3a reveals that those jobs account for 41.8% of total number of jobs. But $\approx$ 0% of the total processing time. Also, They account for $\approx$ 0% of the total recorded energy consumption.

- Jobs with missing energy measurements. This could be due to a kernel error, specific administrator configuration, ... This is a study of energy consumption. Jobs with missing data are of little interest to us. Thus, we discard them as well. In Figure 7.3a, we observe that jobs with missing energy measurement account for 18% (19%) of the total number of jobs (total execution time).



(a) Selcting jobs with a usable energy profile



(b) Selecting jobs that use a full processor.

Fig. 7.3: Job filtering process; The jobs number and the jobs area represents respectively the total number of unique jobs and the total execution time.

The next filter concerns the number of nodes a job uses.

- The reservations in OAR can be core-based. Jobs of type serial, use a single core, and other jobs can also allocate any number of nodes that may not cover an integer number of processors. At the same time, RAPL measure the energy consumption of a full processor. Although it is one of the finest energy measurement tools, it does not reach the level of individual cores. Extracting the energy profiles of jobs that share a processor is a very complicated task and requires a full study which is beyond the scope of this work. We focus on jobs that reserve and use one or more full processors (a processor has 16 cores) for the full duration of their execution.
  Figure 7.3b shows that jobs that use full processor account for 36,6% of the total number of jobs and 63.9% of the total execution area.
  Although the jobs that use less than a full processor represent a sizable portion of the total jobs, we are obliged to discard them.

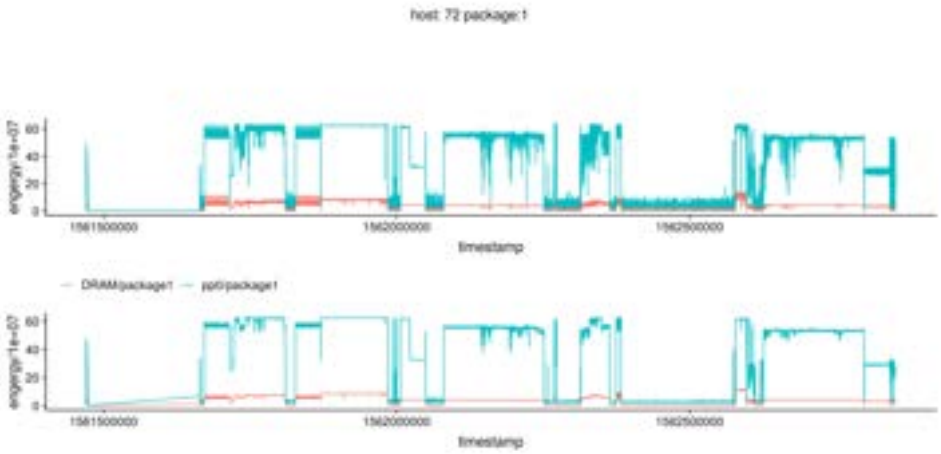


Fig. 7.4: Energy measurement aggregations; The upper figure represents the measurements as taken from Colmet. The bottom figure represents the results of the aggregation process.

The goal of this work is the analysis of the energy profile of jobs for the goal of using them in a meaningful way. In order to detect a meaningful pattern, the energy profile must sufficiently long. We use jobs that are longer than 30 minutes.

7.4.2 Energy Data preprocessing

For the remainder of this works, we will use the terms signal or series to refer to an energy profile. They come respectively from the worlds of signal analysis and time series analysis. We use these terms because most of the techniques we rely on in the following sections are derived from these fields.

7.4.2.1 Signal aggregation

Energy measurements are reported once every 5 seconds. Measurements on such a narrow timeframe are very prone to random noise and fluctuations that are more related to the external state of the machine (such as ambient temperature, system interruptions, ...) than job patterns themselves.
To reduce the noise, we decide to perform a minute based aggregation. We take all the RAPL measurements of any given minute and we average them. We compute the average of every 12 observations.
Figure 7.4 shows an example of the aggregation process.The upper figure represents the original signal, and the lower figure represents the aggregated signal. The aggregated signal has less fluctuation than the original. Also, we note that the general patterns (Constance, periodicity) of the signal become more visible thanks to the aggregation. Also, the aggregation reduces the data set size by a factor of 12. Thus, making subsequent computations faster.

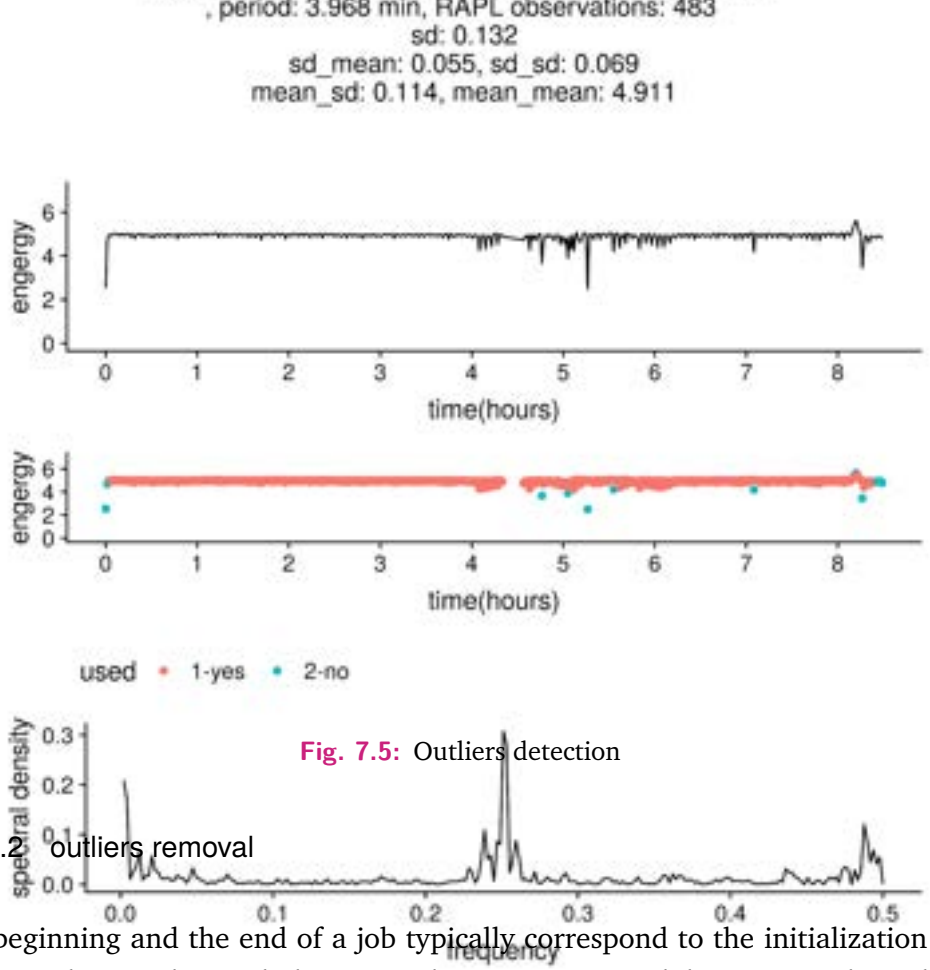


Fig. 7.5: Outliers detection

7.4.2.2 outliers removal

The beginning and the end of a job typically correspond to the initialization and finishing phases. They include actions that are unique and don't repeat through the life of the application such as initializing kernels, loading libraries, saving results. These phases are usually short (a few minutes in the beginning and the end), and they are not representative of the general behavior of the signal. Thus we decided to cut them.

Also, we observed points in the middle of the signal that are very different values from the rest. They are larger (or smaller) and are several orders of magnitude compared to the average variance of the rest of the points, which causes heavy statistical interference. Thus we decided to remove them as well.

Figure 7.5 shows an example of the filtering process. the blue points correspond to the points we have filtered. We observe that the values at the beginning and the **end** are different from the rest of the signal. Also, in the middle of the signal, there are outliers which we filter using the z-score [16] with a threshold of 3, which allows us the filter out the points that are outside the 3rd standard deviation. Those points correspond to all the blue points in figure 7.5 other than the one at the beginning and end.

7.4.3 Classification Tree

After performing the preprocessing steps, we obtain energy profiles that are complete and long enough to study. To the extent of our knowledge , this is the first work of its kind. We have little idea about the types of signals we will encounter. Thus, we start by performing some observational work. We visualize the energy profiles and make

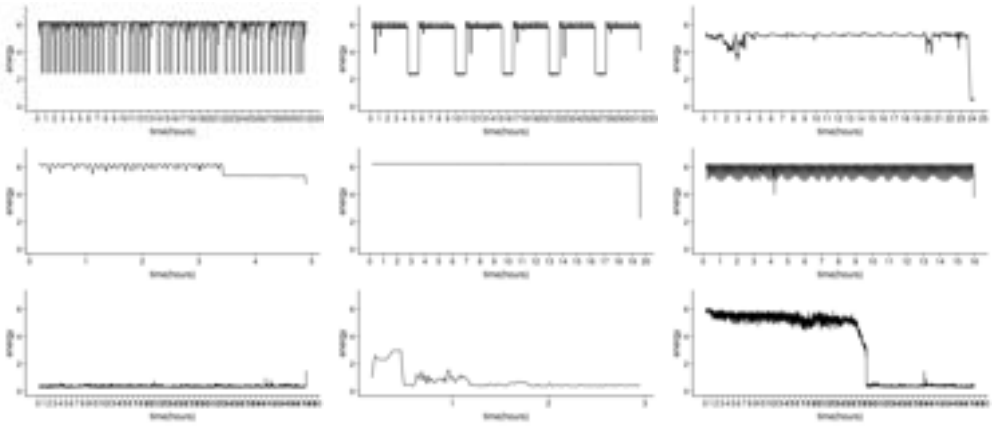


Fig. 7.6: Energy profiles

some hypothesizes. In Figure 7.6 ,we show some examples of energy profiles. The x-axis represents the running time of the job (in hours), and the y-axis represents the energy value.

- There are different types of signals with few commonalities. The variations, patterns, amplitudes, and durations are all changing from one profile to another.
- Through this variability, There major classes can be identified; periodic signals (upper left, upper center, middle right), constant signals (middle center), and chaotic signal with no particular characteristics (lower center and lower left.)
- Also, we note that these classes are not mutually exclusive. An energy profile can have a mix of two or more forms (upper right (periodic+ chaotic)), or it can start with one form and finish with another (middle left (periodic then constant)).
- Direct observation is the most reliable method. But it is not practical in large production systems that receive hundreds or even thousands of jobs daily. Thus, an automated procedure to analyze and classify energy profiles must be put in place.
- Throughout this work, many of the values we chose to perform the classification are hand chosen. The choices are directly related to the platform and the "tolerance level" of the user of the classification. In the subsequent sections, we give detailed justifications for our choices.

Finally, we note that there is no such thing as a perfectly constant signal. All the signals exhibit variability. Sometimes this variability is too minor that it is not visible in the figures. The scale is what makes the distinctions. It falls on us (and subsequently the user of this classification) to decide from which *Threshold* value the variability is sufficiently large to be taken into consideration.

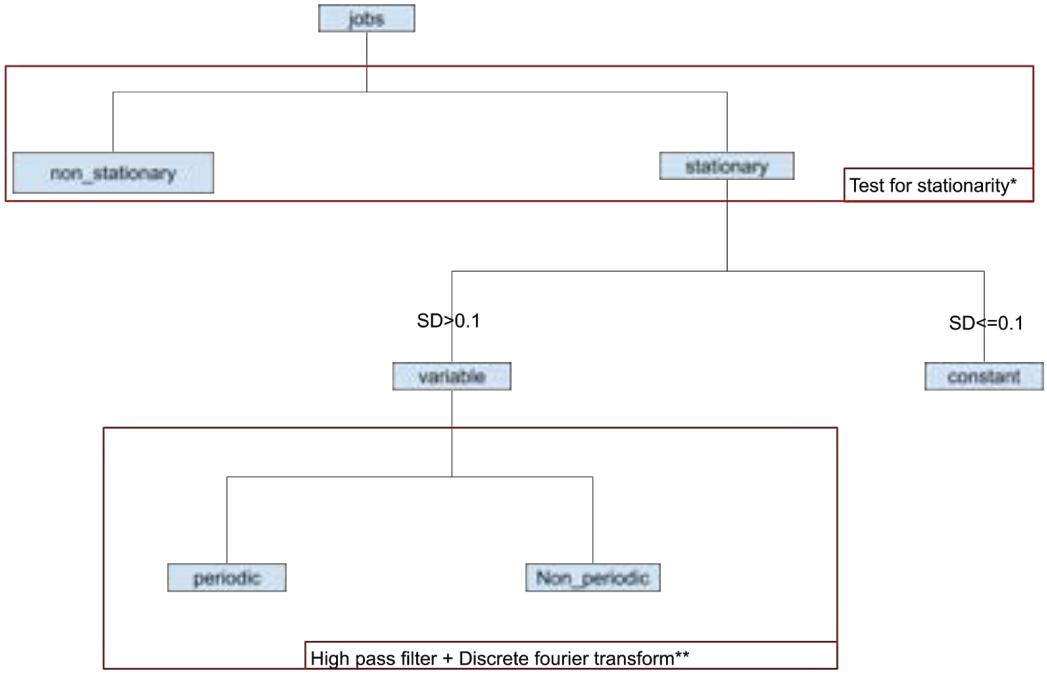


Fig. 7.7: Energy profile Tree

We perform three tests. To build what we will refer to as a classification tree. The tree is presented in Figure 7.7. The first test allows users to isolate stationary energy profiles (test of stationarity). The second test checks if a stationary signal is constant or not (test of variability). The last test checks whether the stationary-non-constant signals are periodic or not (test of periodicity).
The classification tree is built with the idea of utility in mind. This study aims to identify and isolate jobs that can be exploited to optimize energy consumption. This is done by identifying and predicting instances that have low energy consumption. These instances include jobs with constantly low energy usage thought all of their execution. and periodic jobs that have low energy consumption periods that are "long" enough to be exploited properly.
(i)The first step is to detect whether a job is stable through time or not. Stability means predictability which translates to the possibility of exploitation. (ii)In the second level, we isolate constant jobs. the third and the final level we determine the periodic jobs.
In the following sub-sections, we detail the tests we used for this classification

7.4.4 Test for stationarity

Stationarity is a term often used in the field of time series analysis [19]. A time series is said to be stationary if its statistical properties such as the mean, the variance, the standard deviation (SD), autocorrelation, etc. are all constant through time.
Throughout the literature, many tests have been devised to check if a series is

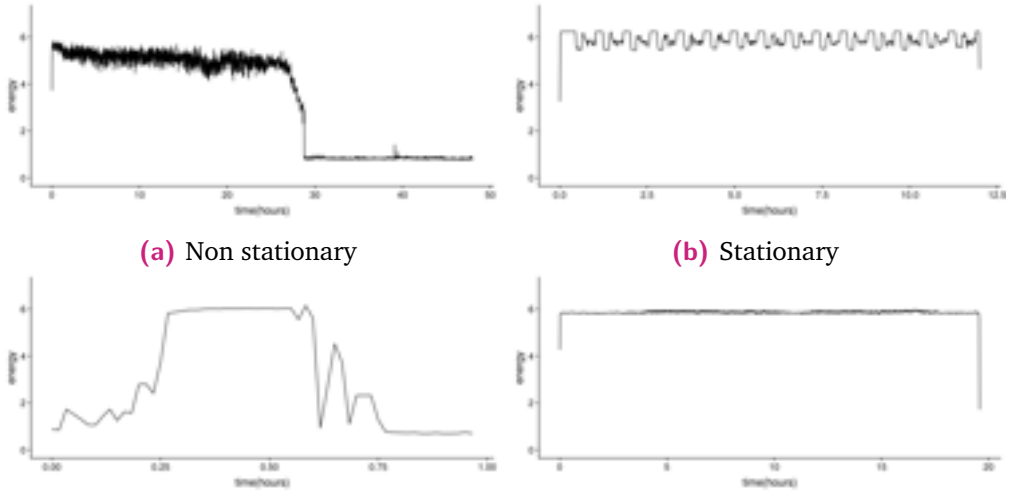
stationary or not. The most popular are parametric tests like the Dicky-fuller test [53], and the KPSS [11] test. However, parametric tests are limited and they cover only a narrow sub-class of possible cases encountered in real data. They are meant to detect specific types of stationarity, namely those brought about by simple parametric models of a generating stochastic process Plus, they do not take into consideration the relative differences between the signals (the scale). In this work, we are dealing with a very wide spectrum of signal types. Many of them have no common proprieties. Thus, we adopt a basic, more general approach to detect stationarity:

- Divide the series into n (6 in this work) parts, we call windows and compute the mean of each window
- Check if there are significant changes between the windows.
  - Compute the SD of the means.
  - If it is above a certain threshold then we consider the series non-stationary and we stop there.
  - If not, we consider the series stationary and we move to the next test.
- The threshold value is chosen based on the maximum and the minimum energy that a job can consume. We decided to take $threshold = \mathbf{0.1}$ since it appears to be a good value to separate minor variations from significant variations that significantly impact the form of the signal

Based on our observations, the presented approach seem to the most effective for detecting stationarity but it sill had shortcomings.
When it comes to stationarity testing, however, the reality is more complex than any model. At the moment of writing this text and to the extent of our knowledge there exist no widely-applicable tests that encompass all real-life scenarios.

Figure 7.8 provides examples of stationary and non-stationary signals. For the non-stationary profiles. Two forms emerge. The first (Figure 7.8a are profiles that appear stationary for the most part but with a sudden change in the regime that comes at a random time. The second (Figure 7.8c are profiles that exhibit chaotic variations with no observable pattern. For the stationary profiles, We also observed two shapes . Periodic signals (Figure 7.8b) and constant (or near constant) signals (Figure 7.8d).

As far as non-stationary signals go, There is very little that can be done about them. So we stop the analysis for this class. In the next levels of the tree, we focus our attention on the stationary signals.



(a) Non stationary (b) Stationary

(c) Non stationary (d) Stationary

Fig. 7.8: Results of the stationarity test

7.4.5 Test of variability

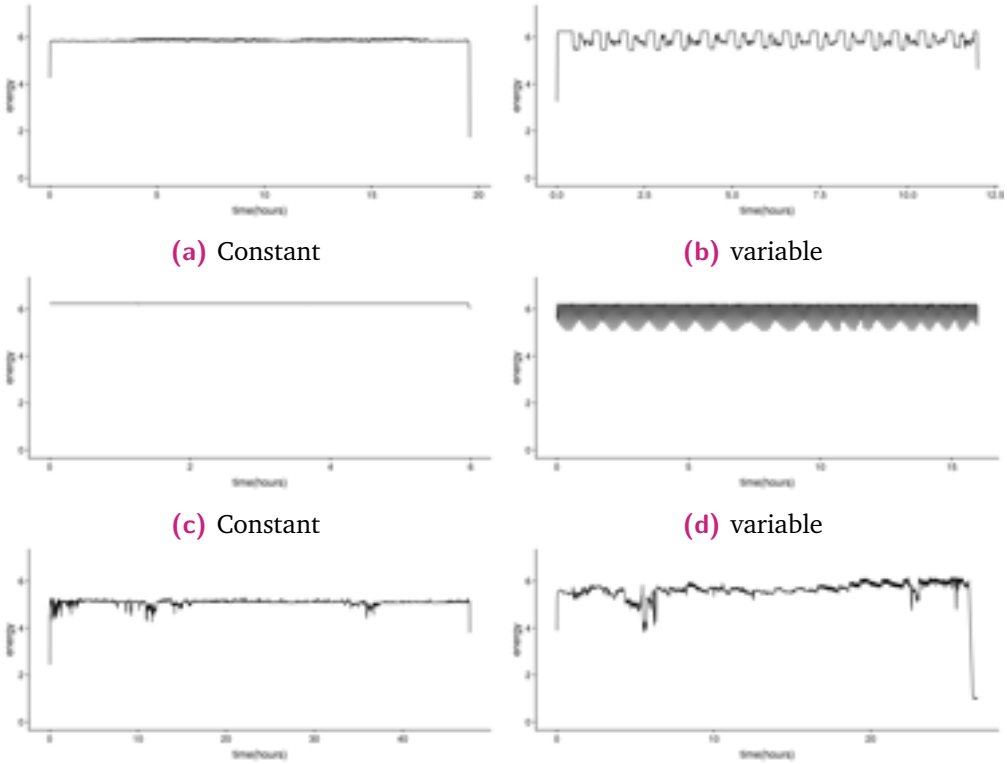
This test includes stationary signals only, which corresponds to the right branch of the tree in Figure 7.7. We measure the amount of variation a signal exhibits.
We define a signal to be constant if its variation through time is too minor to have any significant impact or to be exploited. These variations could be periodic in nature, noise, or a mix of both. Regardless of their nature we still label the signal as simply constant or variable. We use The same threshold value of this test as well ($threshold = \mathbf{0.1}$). The test is simple. We compute the value of the standard deviation. the standard deviation (σ) is defined as follows:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2} \tag{7.1}$$

where N is the number of observations, x_i is the value of observation i, and $\overline{x}$ is the signal mean

- If $\sigma < 0.1 \rightarrow$ signal is constant
- If $\sigma >= 0.1 \rightarrow$ signal is non-constant

Figure 7.9 gives examples of the constant and non constant signals. The constant profiles generally include jobs with no visible changes in the energy consumptions throughout the execution (Figure 7.9c) and jobs that exhibit minor fluctuation (Figure 7.9a). The non constant job mostly includes jobs that are periodic (Figures 7.9b and 7.9d).



(a) Constant (b) variable

(c) Constant (d) variable

(e) Constant (f) variable

Fig. 7.9: Results of the variability test

Figures 7.9e and 7.9f exhibit interesting patterns. We can see that they are neither constant nor periodic (in the strict sense of the words).

- Profile 7.9e: is mostly constant with some small spikes now and then and three distinct distortions of the signal. However, such distortion is too small (below the σ threshold).
- Profile 7.9f is a mix of all the classes. It has some traces of periodicity, fluctuations that are too small and short to be labeled as non-stationary and is not constant. We argue the class that this job should belong to is very subjective.

7.4.6 Test of periodicity

This test corresponds to the final branch of the tree in Figure 7.7. The profiles we have in this part of the tree are stationary and exhibit significant variations. The goal of this test is to check whether these variations are periodic or not.

To detect the periodicity of a signal, we use Discrete Fourier Transformation(DFT) [18]. The Fourier transform in general, and DFT in particular, are at the heart of many signal treatment methods. DFT reveals periodicities in input data as well as the

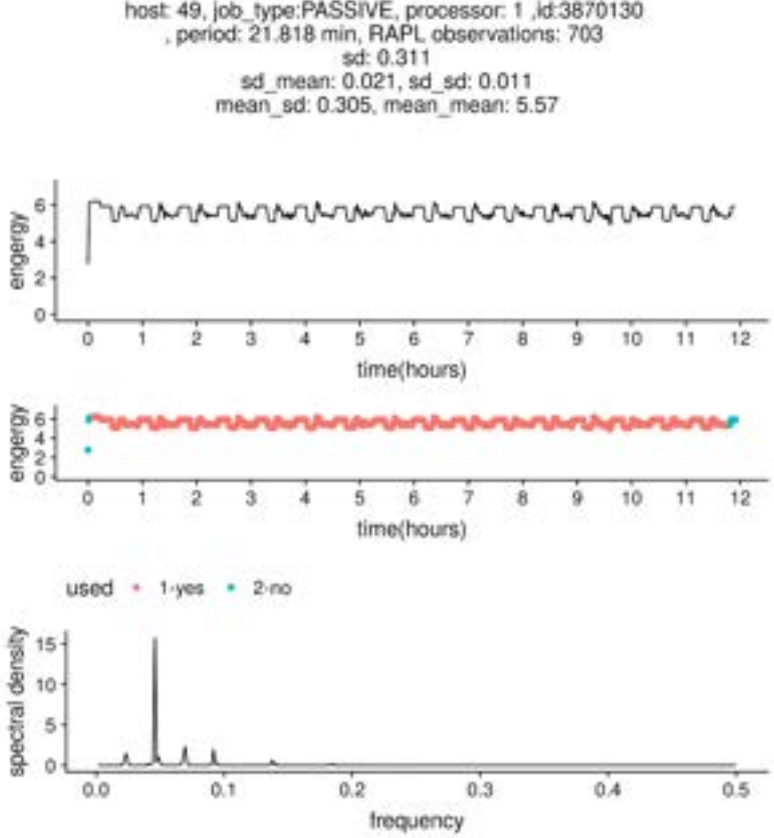


Fig. 7.10: Discrete Fourier Transform

relative strength of any periodic components. Plus, the computational overhead is low because it uses the Fast Fourier transform.
We perform the following step:

- For each signal, we apply a DFT: which allows us to determine the frequencies that appear particularly strong or important.
- From that, we obtain a periodogram that quantifies the contributions of the individual frequencies to the signal.
- We extract the frequency of with the highest contribution: max_freq
- If max_freq $>$ threshold (1 in the examples), then the signal exhibit strong cyclic/periodic behavior.
- If not, then the signal is stationary and variable but with no clear periods.

Figure 7.10 gives an example of the DFT of an energy profile. The x-axis denotes the frequency (relative to the length of the signal). And the y-axis (spectral density) shows the power or contribution of each frequency relative to the rest of the signal. For this particular signal, we can observe a single dominant peak and a few small peaks. The dominant peak corresponds to the distinct periodic behavior we observe in the signal.
Figure 7.11 is an example of a signal that is not stationary and non-constant but not periodic either. It corresponds to a constant signal with distortions in some parts.

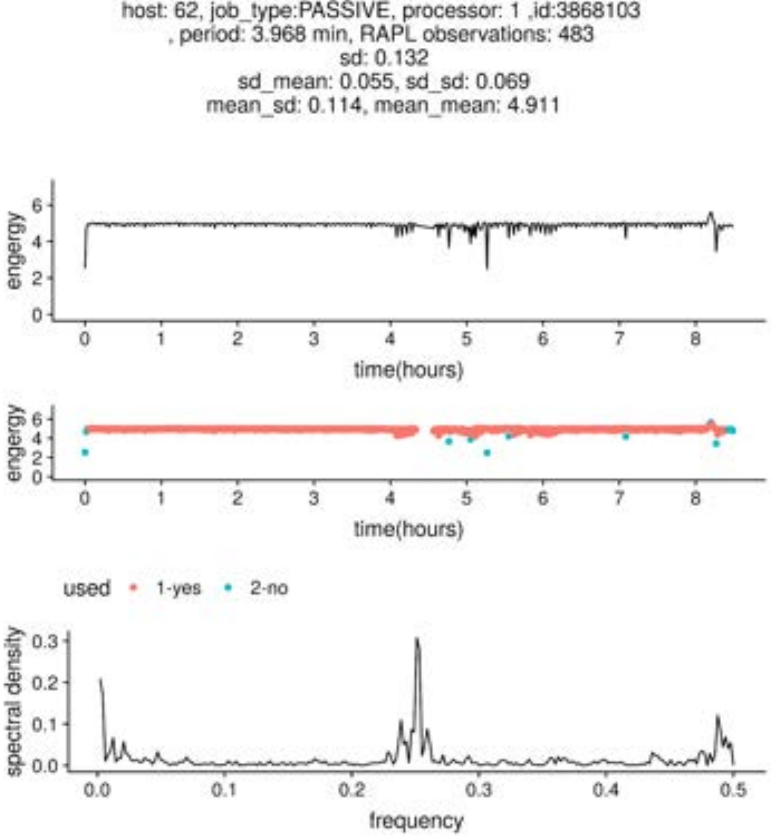


Fig. 7.11: Discrete Fourier Transform

This is reflected in its DFT, where the spectral density of all the frequencies is very low (0.3 being the highest value).

In this section, we classified the profiles into three major families; periodic, constant, and non-stationary. We note that this classification is subjective as it relies on our definitions of the different classed on the choice of the threshold values.
Figure 7.12 shows the distribution of jobs after applying the classification tree. Unlike Figures 7.3a and 7.3b, the job number and the area distributions are similar. Around half of the jobs belong to the constant class and a quarter of the jobs belong to the periodic class.

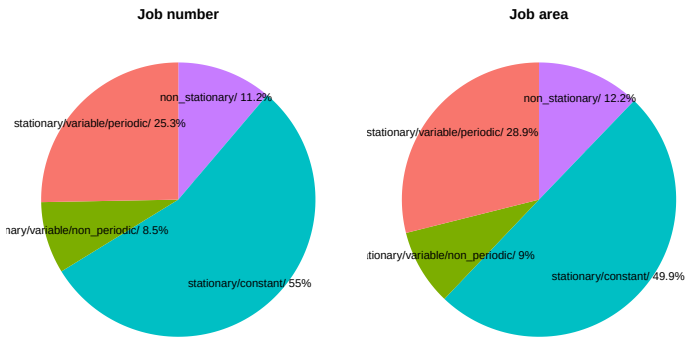


Fig. 7.12: Job distribution after applying Tree 7.7 classification

For the studied sample Around 75% of the total jobs (and respectively total execution area) are predictable.

7.5 Conclusion

In this chapter, we studied the energy consumption of HPC jobs. More precisely, we focused on the different patterns an energy profile can have. We observed that energy profiles although unique to each job share certain common traits that allow us to classify them into specific classes. We Relied on the RAPL energy monitoring tools to collect energy measurements. Then, we proceeded to build a classification tree that sorts jobs into periodic, constant, and non-stationary. We observed that constant and periodic jobs make up around 75% of the total execution time. This indicates that the majority of the trace have predictable energy patterns. This shows great promise since predictability could potentially lead to exploitability.

In this work, we limited our analysis to include only the jobs that execute on a full processor. We note that reserving and running an entire processor does not necessarily mean that it is used to its fullest potential. This could be due to various reasons like a job going through multiple phases of execution, bottlenecks, or simply sub-optimal code that lead to unused (but reserved) cores. We take another look at Figure 7.6. The bottom left sub-figure is an example of a job that reserved the full processor but its energy consumption is almost none. The upper-middle figure is an example of periodic jobs that have phases of low energy consumption.

Two steps might follow. Predicting future energy consumption from user history and coupling low consumption jobs together.

Predicting future energy consumption: In previous works we have shown that we can predict job runtimes via user history. The same can be done for energy profiles. More specifically for the jobs with a predictable pattern.
couple low consumption jobs together: This approach will specifically target constant jobs with low energy consumption and periodic jobs with sufficiently long low phases.

8 Conclusion

Scheduling, even its most basic form, is an NP-hard problem, and finding the optimal solution requires an exponential computation time. Online scheduling is even harder. It adds several layers of uncertainty to an already complex problem. Faced with such complexity, system administrators opted for the safest and the simplest solution, using basic ordering heuristics.
Simple ordering heuristics have been, for the longest time, the go-to method to sort jobs in HPCs. However, with the increase in size and complexity of supercomputers, such elementary methods are no longer able to satisfy the numerous constraints a scheduler must contend with. Thus, alternative, more promising methods have been gaining popularity. Machine learning is one such method. ML studies focus on investigating and analyzing patterns and anomalies in data and building solutions based on these investigations. ML has been applied with various degrees of success on the scheduling problem, from building fully automated schedulers to reducing the inherent uncertainly in some parameters.

In this thesis, we study the interactions between Online scheduling and machine learning. We unravel some of the gains, pitfalls, and promising directions that come with the application of data analysis and machine learning techniques to the field of HPC online scheduling.

We start our experimental campaign with a comparative study. We evaluate scheduling policies by going beyond what we learn from basic metrics and popular perceptions of fairness. We study some of the principal weaknesses of the evaluation process. The first weakness is using a single metric or a similar set of metrics (like the waiting time and the slowdown). The second is reducing a metric to a single number (the average or the max). We show that such simplified assessments of scheduler performance tend to be misleading as they showcase a very narrow view of the performance. We also argue about the importance of area property when scheduling and we show the robustness of this properly under different circumstances.

Then we proceed to propose two methods to explore logs and machine learning to improve performance. First, we conceive and test a method to generate more expressive policies. We build mixed policies, a larger family of policies that are a combination of several job characteristics. We tune them using historical logs and

black-box optimizers.
We observe that for any set of jobs of any given period, there exists a mixed policy that offers performance improvements that are far superior to anything simple policies can offer. On the other hand, such policies tend to be very specialized to a single set of jobs as their performance significantly degrades when tested on another job set.
This excellent performance is attributed mainly to overfitting. By overfitting, mixed policies managed to circumvent several problems caused by the lack of knowledge about incoming jobs such as unknown runtimes problem.
Second, we address the problem of unknown runtimes. Although sound approximations can be found throughout the literature, the exact value of job runtimes remains near-impossible to predict. Extreme factors, such as the machine state and the machine location in the rack can have subtle and cumulative effects on the runtime. Such factors are too complex and volatile to account for. We show that a simple classification coupled with a contingency mechanism offers improvements in performance that are on par with what scheduling with exact runtimes offers. This type of simplification is not particularly new. Other fields adopted similar approaches long before this work. For instance, in stock market analysis, Predicting the exact value of a stock is an impossible task due to the large number of external factors that cannot be measured or quantified. Thus, it is common to predict if the stock is going up or down instead. This simple classification has proven to be quite helpful.

Finally, we focus on the energy consumption of individual HPC applications. We show that most jobs follow certain energy consumption patterns. These patterns could be either constant or periodic. Then we build a classification tree using several statistical tests to automatically identify the energy patterns of a job. This knowledge can be used in several ways. (i) To predict the energy profiles of waiting jobs using techniques and features that are similar to the ones presented in Chapter 6. (ii) Or To use the knowledge about instantaneous consumption of waiting jobs to build and energy aware scheduler.

We briefly discuss three directions for future research:

- **Mixed policies:** In Chapter 6 we chose to limit our search to linear expressions. We made THIS design decision for several reasons. It is simpler to reason about, and the resulting expressions are interpretable. Also, it gave us the possibility to visualize the search space (Figure 5.4), which allowed us to gain valuable insights. However, it is possible to expand the search space to including non-linear combinations of the job characteristics (quadratic, polynomial, ...). We believe this expansion will allow us to generate policies that perform better, at least for the offline case. We are also interested in the pattern that will emerge and weight distribution when non-basic characteristics are involved.

- **Combining machine learning and energy profiling**. In section 7, we have shown that most jobs follow an identifiable energy consumption pattern. This pattern could be either constant or periodic. We can use this knowledge in conjunction with the runtimes prediction techniques presented in Chapter 6 to estimate the energy consumption of incoming jobs. We can take this research direction even further by building an energy-aware scheduler.

- **Using Deep Reinforcement learning:**. Deep learning in general and deep RL represent the pinnacle of automatic scheduling policies. Full control is given to the RL algorithm to take whatever ordering decision it deems fit. A plausible research direction is to build an RL model (possibly with policy gradient methods). And try to study its behavior in detail. It is interesting for two reasons. First, there is the promise of greater improvements. By having access to larger search space, we believe that RL has a great potential to achieve unprecedented performance improvements. The second reason is that we believe that analyzing the behavior of a fully automated model can give us insights into scheduling practices that were unknown before.

Bibliography

[1]`https://www.top500.org/` (cit. on pp. 1, 16).

[2]https://gricad.univ-grenoble-alpes.fr/ (cit. on p. 84).

[3]https://oar.imag.fr/ (cit. on p. 84).

[4]https://www.kernel.org/doc/Documentation/accounting/taskstats.txt (cit. on p. 84).

[5]https://man7.org/linux/man-pages/man7/cgroups.7.html (cit. on p. 84).

[6]Anurag Agarwal, Selcuk Colak, Varghese S. Jacob, and Hasan Pirkul. „Heuristics and augmented neural networks for task scheduling with non-identical machines“. In: *European Journal of Operational Research* 175.1 (2006), pp. 296–317 (cit. on p. 7).

[7]D. H. Ahn, J. Garlick, M. Grondona, et al. „Flux: A Next-Generation Resource Management Framework for Large HPC Centers“. In: *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 9–17 (cit. on p. 22).

[8]Francisco Almeida, Marcos Dias de Assuncao, Jorge Barbosa, et al. „Energy Monitoring as an Essential Building Block Towards Sustainable Ultrascale Systems“. In: *Sustainable Computing : Informatics and Systems* 17 (Mar. 2018), pp. 27–42 (cit. on p. 12).

[9]Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. „Are User Runtime Estimates Inherently Inaccurate?“ In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 253–263 (cit. on pp. 8, 59).

[10]Frank Bellosa. „The Benefits of Event: Driven Energy Accounting in Power-Sensitive Systems“. In: *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. EW 9. Kolding, Denmark: Association for Computing Machinery, 2000, pp. 37–42 (cit. on p. 13).

[11]Alok Bhargava. „On the Theory of Testing for Unit Roots in Observed Time Series“. In: *The Review of Economic Studies* 53.3 (July 1986), pp. 369–384 (cit. on p. 93).

[12]Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. „A survey on metaheuristics for stochastic combinatorial optimization“. In: vol. 8. 2. June 2009, pp. 239–287 (cit. on p. 49).

[13]Peter Bodik, Moises Goldszmidt, Armando Fox, Dawn B. Woodard, and Hans Andersen. „Fingerprinting the Datacenter: Automated Classification of Performance Crises“. In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: Association for Computing Machinery, 2010, pp. 111–124 (cit. on p. 10).

[14]Marin Bougeret, Pierre-François Dutot, Klaus Jansen, Christina Otte, and Denis Trystram. „Approximation Algorithms for Multiple Strip Packing“. In: *Approximation and Online Algorithms, 7th International Workshop, WAOA 2009, Copenhagen, Denmark, September 10-11, 2009. Revised Papers*. 2009, pp. 37–48 (cit. on p. 7).

[15]Raouf Boutaba, Mohammad A. Salahuddin, Noura Limam, et al. „A comprehensive survey on machine learning for networking: evolution, applications and research opportunities“. In: *Journal of Internet Services and Applications* 9.1 (June 2018), p. 16 (cit. on p. 11).

[16]C.H. Brase and C.P. Brase. *Understanding Basic Statistics*. Cengage Learning, 2015 (cit. on p. 90).

[17]Leo Breiman. „Random Forests“. In: vol. 45. 1. Hingham, MA, USA: Kluwer Academic Publishers, Oct. 2001, pp. 5–32 (cit. on p. 65).

[18]William L. Briggs and van Emden Henson. *The DFT: an Owner's Manual for the Discrete Fourier Transform*. Philadelphia: Society for Industrial and Applied Mathematics, 1995 (cit. on p. 95).

[19]P.J. Brockwell and R.A. Davis. *Introduction to Time Series and Forecasting*. Springer Texts in Statistics. Springer International Publishing, 2016 (cit. on p. 92).

[20]Nicolas Capit, Georges Da Costa, Yiannis Georgiou, et al. „A batch scheduler with high level components“. In: *Cluster computing and Grid 2005 (CCGrid05)*. Cardiff, United Kingdom: IEEE, 2005 (cit. on p. 2).

[21]Danilo Carastan-Santos and Raphael Y. de Camargo. „Obtaining Dynamic Scheduling Policies with Simulation and Machine Learning“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: ACM, 2017, 32:1–32:13 (cit. on pp. 4, 10, 23, 25, 30, 36, 42, 59, 69, 71).

[22]Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. „Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms“. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917 (cit. on pp. 22, 71).

[23]Minh Thanh Chung, Kien Pham, Nam Thoai, and Dieter Kranzlmueller. „A New Approach for Scheduling Job with the Heterogeneity-Aware Resource in HPC Systems“. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2019, pp. 1900–1907 (cit. on p. 10).

[24]Ira Cohen, Steve Zhang, Moises Goldszmidt, et al. „Capturing, Indexing, Clustering, and Retrieving System History“. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. SOSP '05. Brighton, United Kingdom: Association for Computing Machinery, 2005, pp. 105–118 (cit. on p. 10).

[25]Pawel Czarnul, Jerzy Proficz, and Adam Krzywaniak. „Energy-Aware High-Performance Computing: Survey of State-of-the-Art Tools, Techniques, and Environments“. In: *Scientific Programming* 2019 (Apr. 2019), p. 8348791 (cit. on p. 12).

[26]H. Al-Daoud, I. Al-Azzoni, and D. G. Down. „Power-Aware Linear Programming based Scheduling for heterogeneous computer clusters“. In: *International Conference on Green Computing*. Aug. 2010, pp. 325–332 (cit. on p. 7).

[27]A. P. Dempster, N. M. Laird, and D. B. Rubin. „Maximum likelihood from incomplete data via the EM algorithm“. In: vol. 39. 1. 1977, pp. 1–38 (cit. on p. 61).

[28]Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. „A Validation of DRAM RAPL Power Measurements“. In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: Association for Computing Machinery, 2016, pp. 455–470 (cit. on p. 13).

[29]Eelco Dolstra, Eelco Visser, and Merijn de Jonge. „Imposing a Memory Management Discipline on Software Deployment“. In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 583–592 (cit. on pp. 40, 71).

[30]Jack Dongarra, Hatem Ltaief, Piotr Luszczek, and Vincent M. Weaver. „Energy Footprint of Advanced Dense Numerical Linear Algebra Using Tile Algorithms on Multicore Architectures“. In: *Proceedings of the 2012 Second International Conference on Cloud and Green Computing*. CGC '12. USA: IEEE Computer Society, 2012, pp. 274–281 (cit. on p. 14).

[31]Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. *Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator*. Ed. by Narayan Desai and Walfredo Cirne. Cham, 2017 (cit. on pp. 22, 26, 71).

[32]Daniel Ellsworth, Tapasya Patki, Martin Schulz, Barry Rountree, and Allen Malony. „Simulating Power Scheduling at Scale“. In: *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*. E2SC'17. Denver, CO, USA: Association for Computing Machinery, 2017 (cit. on p. 22).

[33]Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. „A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise“. In: *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, pp. 226–231 (cit. on p. 61).

[34]Y. Fan, P. Rich, W. E. Allcock, M. E. Papka, and Z. Lan. „Trade-Off Between Prediction Accuracy and Underestimation Rate in Job Runtime Estimates“. In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 530–540 (cit. on pp. 8, 59).

[35]Dror G Feitelson. „Resampling with feedback — a new paradigm of using workload data for performance evaluation“. In: *European Conference on Parallel Processing*. Springer. 2016, pp. 3–21 (cit. on pp. 21, 26).

[36]Dror G Feitelson and Larry Rudolph. „Metrics and benchmarking for parallel job scheduling“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1998, pp. 1–24 (cit. on pp. 20, 29).

[37]Dror G Feitelson, Larry Rudolph, Uwe Schwiegelshohn, Kenneth C Sevcik, and Parkson Wong. „Theory and practice in parallel job scheduling“. In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 1997, pp. 1–34 (cit. on p. 25).

[38]Christodoulos A. Floudas and Xiaoxia Lin. „Mixed Integer Linear Programming in Process Scheduling: Modeling, Algorithms, and Applications“. In: *Annals of Operations Research* 139.1 (Oct. 2005), pp. 131–162 (cit. on p. 7).

[39]Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. „Estimation of energy consumption in machine learning“. In: *Journal of Parallel and Distributed Computing* 134 (2019), pp. 75–88 (cit. on p. 13).

[40]Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W.H.Freeman & Co., 1990 (cit. on p. 7).

[41]E. Gaussier, J. Lelong, V. Reis, and D. Trystram. „Online Tuning of EASY-Backfilling using Queue Reordering Policies“. In: *IEEE Transactions on Parallel and Distributed Systems* 29.10 (Oct. 2018), pp. 2304–2316 (cit. on pp. 4, 25, 26, 69).

[42]Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. „Improving Backfilling by Using Machine Learning to Predict Running Times“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 64:1–64:10 (cit. on pp. 4, 8, 9, 23, 25, 39, 42, 59, 80).

[43]R. Ge, X. Feng, S. Song, et al. „PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications“. In: *IEEE Transactions on Parallel and Distributed Systems* 21.5 (2010), pp. 658–671 (cit. on p. 13).

[44]Tobias Glasmachers, Tom Schaul, Sun Yi, Daan Wierstra, and Jürgen Schmidhuber. „Exponential Natural Evolution Strategies“. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. GECCO '10. Portland, Oregon, USA: ACM, 2010, pp. 393–400 (cit. on p. 49).

[45]Bhavishya Goel, Sally A. McKee, and Magnus Själander. „Chapter two - Techniques to Measure, Model, and Manage Power“. In: ed. by Ali Hurson and Atif Memon. Vol. 87. Advances in Computers. Elsevier, 2012, pp. 7–54 (cit. on p. 12).

[46]Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016 (cit. on pp. 4, 11).

[47]Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. „Multi-Resource Packing for Cluster Schedulers“. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 455–466 (cit. on p. 11).

[48]R. E. Grant, S. L. Olivier, J. H. Laros, R. Brightwell, and A. K. Porterfield. „Metrics for Evaluating Energy Saving Techniques for Resilient HPC Systems“. In: *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 2014, pp. 790–797 (cit. on p. 12).

[49]Jian Guo, Akihiro Nomura, Ryan Barton, Haoyu Zhang, and Satoshi Matsuoka. „Machine Learning Predictions for Underestimation of Job Runtime on HPC System“. In: *Supercomputing Frontiers*. Ed. by Rio Yokota and Weigang Wu. Cham: Springer International Publishing, 2018, pp. 179–198 (cit. on pp. 9, 59, 68).

[50]Daniel Hackenberg, Thomas Ilsche, Robert Schöne, et al. „Power measurement techniques on standard compute nodes: A quantitative comparison“. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2013, pp. 194–204 (cit. on p. 14).

[51]N. Hansen. „The CMA evolution strategy: a comparing review“. In: *Towards a new evolutionary computation. Advances on estimation of distribution algorithms*. Ed. by J.A. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea. Springer, 2006, pp. 75–102 (cit. on p. 49).

[52]James W. Hardin and Joseph Hilbe. *Generalized Linear Models and Extensions*. College Station, Texas: Stata Press, 2001, p. 245 (cit. on p. 10).

[53]Michio Hatanaka. *Time-Series-Based Econometrics: Unit Roots and Co-integrations*. Oxford University Press, 1996 (cit. on p. 93).

[54]Benjamin Hindman, Andy Konwinski, Matei Zaharia, et al. „Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center“. In: NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308 (cit. on p. 22).

[55]E. Hopper and B. C. H. Turton. „A Review of the Application of Meta-Heuristic Algorithms to 2D Strip Packing Problems“. In: vol. 16. 4. Dec. 2001, pp. 257–300 (cit. on p. 7).

[56]E. S. H. Hou, N. Ansari, and Hong Ren. „A genetic algorithm for multiprocessor scheduling“. In: *IEEE Transactions on Parallel and Distributed Systems* 5.2 (Feb. 1994), pp. 113–120 (cit. on p. 7).

[58]Intel. „Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide“. In: 3 (2009) (cit. on p. 13).

[59]Mateusz Jarus, Ariel Oleksiak, Tomasz Piontek, and Jan Węglarz. „Runtime power usage estimation of HPC servers for various classes of real-life applications“. In: *Future Generation Computer Systems* 36 (July 2014), pp. 299–310 (cit. on p. 13).

[60]Morris A. Jette, Andy B. Yoo, and Mark Grondona. „SLURM: Simple Linux Utility for Resource Management“. In: *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60 (cit. on pp. 2, 9).

[61]Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. „Reinforcement Learning: A Survey“. In: *J. Artif. Int. Res.* 4.1 (May 1996), pp. 237–285 (cit. on p. 11).

[62]S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. „Optimization by simulated annealing“. In: vol. 220. 4598. 1983, pp. 671–680 (cit. on p. 49).

[63]Dalibor Klusáček, Luděk Matyska, and Hana Rudová. „Alea – Grid Scheduling Simulation Environment“. In: *Parallel Processing and Applied Mathematics*. Ed. by Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1029–1038 (cit. on p. 22).

[64]Michael Kuchnik, Jun Woo Park, Chuck Cranor, et al. *This is why ML-driven cluster scheduling remains widely impractical*. Tech. rep. CMU-PDL-19-103. Carnegie Mellon University, Parallel Data Laboratory, 2019 (cit. on pp. 9, 59, 62).

[65]Jérôme Lelong, Valentin Reis, and Denis Trystram. „Tuning EASY-Backfilling Queues“. In: *21st Workshop on Job Scheduling Strategies for Parallel Processing*. 31st IEEE International Parallel & Distributed Processing Symposium. Orlando, United States, May 2017 (cit. on pp. 4, 9, 19, 25, 34, 39, 54, 72).

[66]Zewen Li, Wenjie Yang, Shouheng Peng, and Fan Liu. *A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects*. 2020. arXiv: `2004.02806 [cs.CV]` (cit. on p. 11).

[67]Uri Lublin and Dror G. Feitelson. „The Workload on Parallel Supercomputers: Modeling the Characteristics of Rigid Jobs“. In: *J. Parallel Distrib. Comput.* 63.11 (Nov. 2003), pp. 1105–1122 (cit. on p. 60).

[68]Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. „Resource Management with Deep Reinforcement Learning“. In: *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. HotNets '16. Atlanta, GA, USA: ACM, 2016, pp. 50–56 (cit. on pp. 4, 11).

[69]Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. „Learning Scheduling Algorithms for Data Processing Clusters“. In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Beijing, China: Association for Computing Machinery, 2019, pp. 270–288 (cit. on pp. 4, 11).

[70]Ahuva W. Mu'alem and Dror G. Feitelson. „Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling“. In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (2001), pp. 529–543 (cit. on pp. 16, 25, 39).

[71]Stefano Nembrini, Inke R König, and Marvin N Wright. „The revival of the Gini importance?“ In: *Bioinformatics* 34.21 (May 2018), pp. 3711–3718. eprint: `https://academic.oup.com/bioinformatics/article-pdf/34/21/3711/26146978/bty373.pdf` (cit. on p. 69).

[72]Y. Ngoko, D. Trystram, V. Reis, and C. Cérin. „An Automatic Tuning System for Solving NP-Hard Problems in Clouds“. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 1443–1452 (cit. on p. 22).

[73]Kenneth O'brien, Ilia Pietri, Ravi Reddy, Alexey Lastovetsky, and Rizos Sakellariou. „A Survey of Power and Energy Predictive Models in HPC Systems and Applications“. In: *ACM Comput. Surv.* 50.3 (June 2017) (cit. on pp. 12, 13).

[74]*Parallel Workloads Archive: Logs*. `http://www.cs.huji.ac.il/labs/parallel/workload/logs.html`. 2018 (cit. on pp. 11, 15, 23, 29, 39, 42).

[75]Johnatan E Pecero, Denis Trystram, and Albert Y Zomaya. „A new genetic algorithm for scheduling for large communication delays“. In: *European Conference on Parallel Processing*. Springer. 2009, pp. 241–252 (cit. on p. 7).

[76]Heyang Qin, Syed Zawad, Yanqi Zhou, et al. „Swift Machine Learning Model Serving Scheduling: A Region Based Reinforcement Learning Approach“. In: SC '19. Denver, Colorado: Association for Computing Machinery, 2019 (cit. on p. 11).

[77]Gonzalo P Rodrigo, P-O Östberg, Erik Elmroth, et al. „Towards understanding HPC users and systems: a NERSC case study“. In: *Journal of Parallel and Distributed Computing* 111 (2018), pp. 206–221 (cit. on p. 25).

[78]L. Sant'ana, D. Carastan-Santos, D. Cordeiro, and R. De Camargo. „Real-Time Scheduling Policy Selection from Queue and Machine States". In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019, pp. 381–390 (cit. on pp. 4, 10).

[79]Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. „The EASY — LoadLeveler API project". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 41–47 (cit. on pp. 4, 10, 16, 17).

[80]Warren Smith, Ian T. Foster, and Valerie E. Taylor. „Predicting Application Run Times Using Historical Information". In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. IPPS/SPDP '98. Berlin, Heidelberg: Springer-Verlag, 1998, pp. 122–142 (cit. on p. 8).

[81]Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and P Sadayappan. „Characterization of backfilling strategies for parallel job scheduling". In: *Parallel Processing Workshops, 2002. Proceedings. International Conference on*. IEEE. 2002, pp. 514–519 (cit. on p. 16).

[82]Srividya Srinivasan, Rajkumar Kettimuthu, Vijay Subramani, and Ponnuswamy Sadayappan. „Selective Reservation Strategies for Backfill Job Scheduling". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 55–71 (cit. on p. 16).

[83]Victoria C. Stodden, Friedrich Leisch, and Roger D. Peng. *Implementing Reproducible Research*. Ed. by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. CRC Press, 2014, p. 448 (cit. on pp. 40, 71).

[84]AR Surve, AR Khomane, and S Cheke. „Energy awareness in hpc: a survey". In: *International Journal of Computer Science and Mobile Computing* 2.3 (2013), pp. 46–51 (cit. on p. 12).

[85]Mohammed Tanash, Brandon Dunn, Daniel Andresen, et al. „Improving HPC system performance by predicting job resources via supervised machine learning". In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. 2019, pp. 1–8 (cit. on p. 9).

[86]Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. „Analyzing and adjusting user runtime estimates to improve job scheduling on the Blue Gene/P". In: *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. Apr. 2010, pp. 1–11 (cit. on p. 71).

[87]Wei Tang, Zhiling Lan, Narayan Desai, and Daniel Buettner. „Fault-aware, utility-based job scheduling on BlueGene/P systems". In: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE. 2009, pp. 1–10 (cit. on p. 25).

[88]Josep Torrellas. „Architectures for Extreme-Scale Computing". In: *Computer* 42 (Dec. 2009), pp. 28–35 (cit. on p. 12).

[89]Josep Torrellas. „Extreme-scale computer architecture". In: *National Science Review* 3.1 (Jan. 2016), pp. 19–23. eprint: `https://academic.oup.com/nsr/article-pdf/3/1/19/31565756/nwv085.pdf` (cit. on p. 12).

[90]Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. „Backfilling Using System-Generated Predictions Rather Than User Runtime Estimates“. In: vol. 18. 6. Piscataway, NJ, USA: IEEE Press, June 2007, pp. 789–803 (cit. on pp. 8, 23, 39, 59, 80).

[91]Ozan Tuncer, Emre Ates, Yijia Zhang, et al. „Diagnosing Performance Variations in HPC Applications Using Machine Learning“. In: *ISC*. 2017 (cit. on p. 10).

[92]Y. Ukidave, X. Li, and D. Kaeli. „Mystic: Predictive Scheduling for GPU Based Cloud Servers Using Machine Learning“. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 353–362 (cit. on p. 10).

[93]Pengfei Wei, Zhenzhou Lu, and Jingwen Song. „Variable importance analysis: A comprehensive review“. In: *Reliability Engineering and System Safety* 142.C (2015), pp. 399–432 (cit. on p. 69).

[94]M. Witkowski, A. Oleksiak, T. Piontek, and J. Wundefinedglarz. „Practical Power Consumption Estimation for Real Life HPC Applications“. In: 29.1 (Jan. 2013), pp. 208–217 (cit. on p. 13).

[95]Michael R. Wyatt, Stephen Herbein, Todd Gamblin, et al. „PRIONN: Predicting Runtime and IO Using Neural Networks“. In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018 (cit. on p. 9).

[96]Fatos Xhafa and Ajith Abraham. „Computational models and heuristic methods for Grid scheduling problems“. In: *Future Generation Computer Systems* 26.4 (2010), pp. 608–621 (cit. on p. 7).

[97]Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. *RLScheduler: An Automated HPC Batch Job Scheduler Using Reinforcement Learning*. 2020. arXiv: `1910.08925 [cs.DC]` (cit. on p. 11).

[98]Huazhe Zhang and H Hoffman. „A quantitative evaluation of the RAPL power control system“. In: *Feedback Computing* (2015) (cit. on p. 13).

[99]Dmitry Zotkin and Peter J. Keleher. „Job-Length Estimation and Performance in Backfilling Schedulers“. In: *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*. HPDC '99. Washington, DC, USA: IEEE Computer Society, 1999 (cit. on p. 9).

[100]Dmitry Zotkin and Peter J Keleher. „Job-length estimation and performance in backfilling schedulers“. In: *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*. IEEE. 1999, pp. 236–243 (cit. on p. 20).

Webseiten

[@57]*How much energy do data centers use*. 2020. URL: `https://davidmytton.blog/how-much-energy-do-data-centers-use/` (visited on Feb. 10, 2020) (cit. on p. 12).

List of Figures

List of Tables