TÉCNICO LISBOA

Self-tuning of cloud systems

Maria da Loura Casimiro

Relatório da disciplina de Introdução à Investigação em Engenharia Electrotécnica e de Computadores

Engenharia Electrotécnica e de Computadores

Orientadores: Doutor João Nuno Silva
Doutor Paolo Romano

Júri

Orientador: Doutor Paolo Romano
Vogal: Doutor Francisco António Chaves Saraiva de Melo

Janeiro de 2018

Abstract

Over the past years, there has been an increase in the use of Cloud Computing for delivery of services. These services range from remote storage systems to Virtual Machines (VMs), available for purchase and for online provisioning of user specific jobs. In response to such an increasing demand, modern cloud providers have widely increased the heterogeneity of platforms they offer (e.g., in terms of computational, storage and network capacities). Therefore, users are left with a huge variety of machines and services to choose from. This is a complex and time consuming optimization problem with relevant economical implications, since there are multiple sets of machines, of different types and numbers — configuration — that may yield similar performances with identical costs, making it difficult to distinguish the best one.

This motivated recent research aimed to automate the identification of the optimal cloud configuration to deploy user applications. Existing approaches employ a wide range of predictive techniques and share a common mechanism: an exploratory phase during which the target application is deployed and tested over a diverse set of configurations, which are typically established in a dynamic fashion, depending on the applications' workload characteristics.

State-of-the-art systems have been shown to be able to identify near-optimal configurations for the final application deployment (i.e., its steady state). Unfortunately, though, existing solutions aim solely at optimizing the efficiency of the final configuration. As such, they neglect the cost of the exploration phase, which can be quite expensive not only for short running jobs, but also for applications subject to frequent workload changes, which accordingly, require frequent reconfigurations.

We argue that, in order to tackle this problem, the optimization process should explicitly take into account the cost dynamics of the whole exploration phase. This is a non-trivial problem as the exploration cost is not only determined by the set of configurations to be tested (that is unknown a priori), but also by the order in which they are explored.

As a first step towards pursuing this goal, this report reviews the related work in this area, and presents the design of the proposed system.

Keywords

Cloud Computing; Exploration Cost; Deployment Cost; Near-Optimal Configuration; Virtual Machine;

Resumo

Nos últimos anos viu-se um crescimento da utilização de computação na núven para prestação de serviços. Estes serviços vão desde sistemas para armazenamento remoto até Máquinas Virtuais (VMs) disponíveis para compra e para correr trabalhos específicos dos utilizadores. Devido a esta crescente procura, os fornecedores aumentaram amplamente a heterogenidade das plataformas que disponibilizam (nomeadamente em termos de capacidades computacionais, de armazenamento e de rede). Assim sendo, os utilizadores ficam com um número enorme de opções entre as quais podem escolher. Este é um complexo e demorado problema de otimização com implicações comerciais relevantes, visto que há vários conjuntos de máquinas, de diferentes tipos e quantidades — configuração — que levam a desempenhos semelhantes e com custos idênticos, tornando difícil escolher a melhor.

Isto motivou os investigadores a desenvolverem sistemas direccionados para identificar a configuração ótima para correr trabalhos na núvem.

Abordagens recentes utilizam uma vasta variedade de técnicas de previsão e partilham uma característica: uma fase de exploração durante a qual a aplicação alvo é corrida e testada utilizando um diverso conjunto de configurações, que são criadas dinamicamente e que dependem das características de trabalho da aplicação.

Sistemas recentes conseguem encontrar a configuração próxima da ótima para correr a aplicação no seu estado estável. Infelizmente, no entanto, as soluções existentes focam-se somente na otimização da eficiência da configuração final. Assim sendo, negligenciam o custo da fase de exploração, que pode ser bastante dispendiosa não só para trabalhos de curta duração como também para aplicações sujeitas a constantes mudanças de cargas de trabalho que requerem reconfigurações frequentes para garantir eficiência ótima.

De modo a solucionar este problema, defendemos que o processo de otimização deve ter em conta, explicitamente, as dinâmicas de custo de toda a fase de exploração. Este problema é não trivial, visto que o custo de exploração é não só determinado pelo conjunto das configurações que são testadas (que é desconhecido a priori), como também pela ordem em que essas configurações são exploradas.

Como um primeiro passo para atingir este objetivo, este relatório faz uma análise ao trabalho recente

nesta área e apresenta um esboço do sistema proposto.

Palavras Chave

Computação na Nuvem; Custo de Exploração; Custo de Execução; Configuração ótima; Máquina Virtual

Contents

List of Figures

List of Tables

List of Acronyms

AWS	Amazon Web Services
EC2	Elastic Compute Cloud
GCE	Google Compute Engine
VM	Virtual Machine
vCPU	virtual CPU
QoS	Quality of Service
ML	Machine Learning
GP	Gaussian Process
BO	Bayesian Optimization
PI	Probability of Improvement
EI	Expected Improvement
RS	Recommender System
CF	Collaborative Filtering
OED	Optimal Experimental Design
LHS	Latin Hyper-Cube Sampling
TM	Transactional Memory
KPI	Key Performance Indicator
TSP	Travelling Salesman Problem

1

Introduction

Cloud computing allows users to enjoy the delivery of services over the internet, such as sending e-mails, editing documents, listening to music or watching TV. Over the past decade, the ever increasing availability of low-cost computers and storage devices and the constant drive towards innovation and improvement of networks, services and devices, has led to a growth in cloud computing, since the first services of the kind were launched by Amazon in 2006. This increasing popularity is related to its remarkable potential in bringing cost savings and ease of access to complex computing platforms to users, while enabling economy of scale for large scale cloud providers.

All the popularity and widespread use led to the emergence of a number of cloud computing providers, among which Amazon (with Amazon Web Services (AWS) Elastic Compute Cloud (EC2)), Google (with Google Compute Engine (GCE)) and Microsoft who are probably some of the main competitors.

Their dog eat dog competition has led to broadening the offer of different platforms and virtual machine types, which today encompass a plethora of different options, which vary widely in terms of computational, memory and network capacity. The choice of machines has a severe impact on the performance of jobs and if the right one is not made, the allocations are sub-optimal and the costs may be higher than necessary.

1.1 Motivation

Existing approaches target different objective functions (e.g., minimizing user [1–3] vs provider [1, 4, 5] costs) and employ a wide range of predictive techniques. Despite their differences, though, they share a key common mechanism: an exploratory phase during which the target application is deployed and tested over a diverse set of configurations in order to build a model that maps the possible system's configurations to the corresponding application's performance. Which and how many configurations will

have to be explored before a final recommendation for the system's configuration is outputted is typically established in a dynamic fashion, based on the shape of the performance function over which the model is being fitted and on the expected accuracy of the model learnt so far.

State-of-the-art systems have been shown to be able to identify near-optimal configurations for the final application deployment (i.e., its steady state). Unfortunately, existing solutions aim solely at optimizing the efficiency of the final configuration. As such, they neglect the cost of the exploration phase.

As a matter of fact, the cost of the exploration phase can be quite expensive not only for short running jobs, but also for long running applications that are subject to frequent workload changes. In this case, the system has to undergo frequent re-optimization phases in order to adapt to workload changes and pursue optimal efficiency.

Since existing systems are mainly focused on optimizing performance and minimizing deployment cost, usually at the expense of the exploration cost, a problem remains: what configuration provides users with a workload performance above some desirable threshold while maintaining both the deployment cost and the cost for finding that solution below some desirable limit?

1.2 Objectives

The goal of this work is to build a self-tuning system for cloud applications that aims to optimize the cost efficiency not only of the final system's configuration but also of the exploration phase performed as part of the tuning process.

This is a non-trivial problem as the exploration cost is not only determined by which configurations are explored, but also by the order in which they are explored. Consider an example system, where the configuration of a cloud based application is expressed via three parameters: i) the number of Virtual Machines (VMs); ii) the instance type of each VM; and iii) the size of an application level buffer, app_{buffer} (e.g., a database in-memory cache). Now, assume the following three configurations:

- c1, composed by 10 VMs, large instance type, $app_{buffer} = 10MB$;
- c2, composed by 10 VMs, large instance type, $app_{buffer} = 100MB$;
- c3, composed by 10 VMs, small instance type, $app_{buffer} = 10MB$;

Exploring the three configurations in the order c1, c2, c3 is cheaper than in the order c1, c3, c2. In fact, the latter requires allocating three clusters of different VM types (large-small-large). Conversely, using the former order, it is possible to test c1 and c2 using the same cluster of VMs, thus allocating only two clusters in total. Indeed, in practical settings, testing different application-level parameters (without altering the underlying platform configurations) is normally cheaper and faster than testing different platform configurations (in terms of different types and/or number of allocated VMs). Changes in application level parameters can, in fact, be tested without incurring the cost and latency of provisioning a (potentially completely) different set of VMs, which includes the corresponding bootstrapping and state transfer phases. Further, allocating a new cluster in order to test a configuration for a few seconds may be very inefficient, as the billing is typically per hour or, in the best case, per minute. More in general, the cost of

exploring a configuration c_{next} depends on the current configuration c_{cur} and can vary significantly with c_{cur}.

We argue that, in order to tackle this problem, the optimization process should explicitly take into account the cost dynamics of the exploration phase. Current state-of-the-art systems reason/plan the next exploration steps using a greedy strategy that only looks at the immediate expected "reward" from visiting a given configuration, say c. The optimization process should, instead, also account for the cost of visiting c, as well as the cost of visiting, after c, other configurations deemed also potentially interesting.

In order to cope with large search spaces (as it is the case for cloud systems), we plan to use Bayesian Optimization (BO) with Expected Improvement (EI) as acquisition function (c.f. Section 2.2.2), so as to find a set of candidate/interesting configurations, which are expected to yield the best improvements over the currently known optimum. The candidate set is determined on the basis of the model built so far and of its uncertainty regarding untested configurations. The cardinality of this size represents the look-ahead factor, i.e., the number of future exploration steps that are expected to be required before outputting a final recommendation.

We then cast the problem of determining the exploration strategy of the configurations in the candidate set to a variant of the well-known Travelling Salesman Problem (TSP). In the TSP, a salesman has to travel through a group of cities, passing through each city only once and returning to the initial one. The problem is finding the shortest route between all cities.

Considering each configuration as a city and the cost to change configurations as the cost of a route, our goal is to visit all the configurations, spending as little money as possible, so as to minimize the exploration cost, while feeding the predictive model with knowledge on the whole set of configurations in the candidate set.

This is a more relaxed variant of the TSP, since there is no need to return to the initial city. Also, an interesting and challenging aspect is that, whenever a configuration is visited, the model can be updated, leading to changes in the candidate set, i.e., to the cities to be visited by the salesman in the corresponding TSP.

1.3 Outline

This report is structured as follows: Chapter 2 analyses related work on systems for the self-tuning of cloud systems; Chapter 3 presents the work plan for this thesis; and finally Chapter 4 presents some observations and comments.

2

State-of-the-Art

This chapter begins by providing an overview of the alternatives made available by modern cloud providers in terms of both diverse platforms, as well as costs and pricing schemes. It emphasizes the need for current state-of-the art systems that analyze the large space of machines in order to provide users with the best configurations for the deployment of their applications.

Section 2.2 overviews some of the most relevant modelling and optimization methodologies that are used by recent systems for self-tuning of cloud applications. These systems are then reviewed in Section 2.3.

Finally, since there are similarities between our problem and the Travelling Salesman Problem (TSP), it is introduced. These similarities become evident when we think that each of the configurations that we wish to explore can be considered as a city and that, just like there are costs to travel from one city to another, there are also different costs to change from one configuration to another.

2.1 Pricing Models for Resource Provisioning in the Cloud

This section provides an overview of the large offer of Virtual Machines (VMs) types and pricing schemes made available by modern cloud providers.The following data considers, in particular, three of the main cloud providers: Google, Amazon and Microsoft, since the machines of these providers are the ones considered in the reviewed systems [1–6].

The existence of multiple providers allows for competition and, therefore, for the prices to come down. However, the huge amount of choice makes it difficult for users to know, first of all, which provider they should go with and then which instance type(s) and how many instances they should choose, in order to obtain a good enough performance for a reasonable price.

To choose a machine for deployment of an application, a user must first think about the higher level

characteristics of the job which are predominant and have more influence in the overall performance. Providers usually offer a choice of machines in 5 broad categories: general purpose, compute optimized, memory optimized, accelerated computing or storage optimized machines. Depending on the providers, some differences may be spotted.

Now that the user has chosen the broad category that best fits his job, he will have to analyze all families that exist in that category, e.g., instances of type 'F' or 'F1'. Each family has certain specific categories, such as the processors of the machines and the optimizations they provide, for example in terms of network bandwidth or support for enhanced networking.

Once the family with the characteristics that best fit the job has been discovered, the user is required to choose the size of the machine that he wants. The sizes vary between 1 to 96 virtual CPUs (vCPUs) and the bigger the machine, the more memory it has.

Usually, all these characteristics are fixed, however some providers, such as Google, allow users to customize their machines. Although there are some rules for the number of vCPUs of each processor and for the amount of memory that can be chosen, the user is still left with a huge amount of combinations that, if correctly chosen, provide configurations that give optimal performance.

Nonetheless, the trick is to be able to choose wisely, which given the broad choice that exists, may be a hard task even for an expert, let alone for a user that has just been introduced to the cloud environment.

Table 2.1 shows the number of VMs offered by each provider. Disregarding custom machine types, GCE doesn't have such a broad choice. However, when they are taken into account, there are more than 3000 possible machines one can choose from. This count was made based on the assumption that each vCPU must have the same memory. Both Amazon and Microsoft don't offer the option of customizing machines, yet have a large selection.

The values for the instance families and for the sizes show the lowest value existent for one case. For example in Amazon's case, there is at least one use case with 3 instance families and several use cases with more. For the sizes, the reasoning is the same: at least one family has 2 sizes available while others have more. All the machines add up to a total showed in the homonymous row.

	Amazon	**Google**	**Microsoft**
Use Cases	5	5	6
Instance Family	≥ 3	≥ 1	≥ 1
Sizes	≥ 2	≥ 1	≥ 3
Customized	—	3080	—
TOTAL	83	25 (3105)	133

Table 2.1: Amount of virtual machines of each cloud provider

To differentiate between machine prices, the first characteristic that pops-up is the period for which resources are acquired. A user can buy one of two types of resources: on-demand resources, for short running, unpredictable jobs or reserved resources, for long running and predictable jobs. While most machines are available for both types of resources, some can only be acquired for one of the two types.

Regarding on-demand resources, the user pays for what he uses, with no long-term commitment. Billing can be per hour or per second, however there is always a payment of one minute, even if only a

few seconds are used.

As for reserved resources, these imply a commitment of one or three years and offer some discounts when compared to on-demand prices.

Depending on the providers, some more machine types are available. For instance, Amazon also has EC2 spot instances. These instances are spare compute capacity in the AWS cloud that users can acquire by specifying bids. A bid corresponds to the price a user is willing to pay for the instance. A machine is acquired when the bid specified by the user is higher than the current market price for that instance. These instances run either until the user stops them or until the spot price exceeds the price that was specified. Another thing that can happen is that the machines can be revoked when EC2 needs that compute capacity back. Users get a two minutes warning before being evicted. These machines also offer the option of specifying a duration of up to 6 hours, with hourly increments. In this case, they will run until that time has passed or until the user terminates them. These resources are known as revocable resources and although they are prone to evictions and uncertainty, the cost savings they offer make up for those disadvantages when good provisioning strategies are employed, like in the work of Shivaram et al. [1].

After the type of resource has been chosen, the availability zone, which corresponds to where the machines actually are, and the operating system that is chosen also influence the prices.

In Table 2.2 the ranges of prices for each provider and for the two payment options are shown. These values reflect the prices of the cheapest and most expensive machines, independently of their use case, family, size, operating system and availability zone. The cheapest machine has the cheapest operating system, is in the cheapest availability zone, belongs to the cheapest family and has the smallest size. The same reasoning, but on the other way round, applies for the most expensive machine.

By analyzing both tables it is clear that there is an enormous number of combinations of instances that can offer the same performance but that have different costs. Choosing between use cases, families, sizes and types of resources is a choice between hundreds of machines, whose difficulty is further increased by the fact that the performance of the job in each configuration is only known after trying it. Therefore, if a non-ideal configuration is chosen, this will only be known a posteriori, after purchasing the machines and trying the workload.

This motivates the need for systems, such as the one we propose in Chapter 3, that are able to predict how many instances and of which model should be acquired, given a specific workload and thresholds for performance and cost.

		Price by hour in $$		
Payment choice		Providers		
		Amazon	Google	Microsoft
On-Demand		0.0058 — 92.576	0.0076 — 12.362	0.011 — 94.49
Reserved	1 year	0.0030 — 79.475	0.03 — 6.65	0.007 — 75.26
	3 years	0.0020 — 66.866	0.021 — 4.68	0.005 — 64.29

Table 2.2: Range of prices of each cloud provider

2.2 Background on Modelling and Optimization Techniques

Current work on self-tuning of complex applications aims to find the best values for tuning specific parameters of those complex applications and state-of-the-art work on optimizing resource allocation in the cloud has as objective discovering the best configurations for deployment of applications in the cloud. These optimizations are application specific. Either of these lines of work requires the use of modelling and optimization techniques, that may be interesting and valuable to our work and ergo shall be briefly described in the next sub sections.

2.2.1 Gaussian Processes

A Gaussian Process (GP) [7, 8] is a stochastic process, that is, a set of random variables sampled over time, that follows a multivariate Gaussian distribution. This distribution is a generalization of the one-dimensional normal distribution to higher dimensions. While a Gaussian distribution is a distribution over a random variable, a GP is a distribution over a function that has mean function m, usually set to 0, and a covariance function K.

GPs are often used as a prior function for Bayesian Optimization (BO). After sampling a search space, GPs are used to fit the sampled points in order to create a model. Each sampled point is associated with a normal distribution and the group of all the distributions that are fitted to all the data points makes the GP, which, in turn, creates the model. With this model, predictions of the best next points to sample are then made with the help of the uncertainty estimations provided by the model.

Current systems, such as CherryPick [2] and iTuned [9], use this technique to estimate a model for predicting the best points to sample next.

2.2.2 Bayesian Optimization

Bayesian Optimization (BO) [10, 11] is a method for finding the optimum value for expensive cost functions. It is especially efficient in situations where the closed-form expression for the function being evaluated is unknown, but wherein samples can be extracted at certain points. This technique not only converges to the optimum with a small number of explorations but also tries to minimize this number.

By setting a prior function, such as GPs described in 2.2.1, that will serve to model the sampled data, BO gathers knowledge about the problem, which is helpful when deciding where to sample next. This decision is made based on an acquisition function. This function can be the Probability of Improvement (PI), which considers the points that have a higher probability, even if it is just a tiny bit higher, of being better than the current one. PI has the drawback of discarding points that, although may have a smaller probability, have higher uncertainty and can thus offer better improvements. Another acquisition function is the EI, which takes into account not only the PI, but also the magnitude of the improvement. This translates a trade-off between exploration (sampling from areas with high uncertainty) and exploitation (sampling from areas with high probability of improvement). However, both acquisition functions are greedy and thus consider only one step look-ahead, which can lead to sub-optimal choices.

Usually, the exploration ends and the optimum is considered to have been found when the improve-

ment that can be achieved by sampling new points is smaller than some given threshold, like it is done in CherryPick [2].

For the purpose of using BO in fixed budget settings (with respect to number of explorations), so as to minimize some overall cost, some proposals have been made [12]. They attempt to leverage those greedy strategies, combining them with look-ahead heuristics so as to move towards a long-term reward.

2.2.3 Recommender Systems and Collaborative Filtering

Recommender System (RS) are systems designed to provide users with recommendations on items they may like [13, 14]. In the past years, the use of such systems has increased, namely being used by applications such as Amazon [15], Netflix, Twitter or Spotify. In order to know which items to recommend, the system uses an algorithm, for example content-based filtering or Collaborative Filtering (CF) [16]. In this report, CF will be briefly explained since it is a technique used by several state-of-the-art systems in the area of self-tuning for the cloud, like ProteusTM [17] and Quasar [4].

CF works by finding a set of users that are similar in terms of their past ratings to the user whom we want to recommend something to. After these users are found, their ratings for the items are analyzed and the ones that have high ratings and that haven't been rated by the target user, are recommended to him.

The similarity between users can be computed in several ways. The most common two are Euclidean Distance Score and the Pearson Correlation Coefficient. The former corresponds to the length of the line segment that connects two points. As an example, if we consider the ratings of two film series, for instance Star Wars and Lord of the Rings, then each point would correspond to the ratings that a certain user had given to both those movies. The closer two of those points are, the more similar those users are likely to be. In order to be able to make comparisons, the distances must be normalized. The Pearson Correlation Coefficient takes the common ranked items between two users into account and considers the correlation between those data sets. The correlation can vary between $\big[-1, 1\big]$, where 1 is total linear correlation, 0 is no correlation and -1 is total negative correlation.

2.2.4 Random Forests

A Random Forest is a machine learning method capable of doing both regression (predicting the value of an input based on previous learnt values) and classification (deciding to which category an input belongs) [18]. It is a collection of decision trees, trained over different subsets of the training sets, and whose outputs are reconciled through some form of voting or averaging. The forest then outputs the classification with most votes. An advantage of random forests over decision trees is that they tend to be less subject to overfitting issues.

Each tree is grown in the following way: consider a forest with N trees and an initial and general training set and take N random samples of the training set, with replacement — one sample for each tree; for each set of samples, each having M variables, choose randomly $k \ll M$ of those variables; split the node into the best split of these k variables. The trees are grown to the largest extent possible, in order to provide more accuracy, and the value k is held constant throughout the whole execution.

2.2.5 Optimal Experimental Design

Another relevant methodology for self-tuning systems, e.g., Ernest [1], is Optimal Experimental Design (OED) [19, 20]. OED establishes a methodology to determine which experimental runs should be performed to estimate a statistical model in an optimal way. Optimality is defined with respect to some statistical criterion and to a given statistical model. An optimal experiment design requires a smaller number of experimental runs to estimate the parameters with the same precision as a non-optimal design.

To measure optimality and to evaluate designs one needs to look at the covariance matrix of the estimator, since minimizing the variance means maximizing the information and thus building a better model. There are several optimality criterions, a popular one, for instance, is D-optimality, which seeks to minimize the determinant of the covariance matrix.

2.3 Self-tuning of Complex Systems

This sections reviews several state-of-the-art techniques for self-tuning of complex systems, with a focus on self-tuning of complex parameters of applications (Section 2.3.1) and optimizing resource allocation in the cloud (Section 2.3.2).

2.3.1 Self-tuning of Complex Applications

Another class of systems that is worth discussing is that of systems for self-tuning of complex applications. These systems search for the best values for tuning parameters that are specific to those applications.

The first of these systems that will be analyzed is iTuned [9]. The goal of this system is to automatically find the best settings for database parameters. iTuned [9] is composed of a planner, that determines the next best experiments to run and an executor that conducts the experiments.

iTuned's [9] planner chooses the experiments based on a technique called Adaptive Sampling. This technique is very similar to the Bayesian Optimization (BO) technique described in section 2.2.2. In order to get the first samples, Latin Hyper-Cube Sampling (LHS) [21, 22] is used. This sampling technique selects m samples from each of m sub-domains of a parameter. These sub-domains are generated by partitioning the domain of a parameter into m equal sub-domains.

To pick the next experiment, first a Gaussian Process (GP) (c.f. section 2.2.1) model is built on top of the initial samples. Then the Expected Improvement (EI) is computed for the next possible points to experiment. The one that has the highest EI is the next experiment. This process ends either when the user is satisfied with the improvements or when the improvement offered by more exploration is below a certain threshold. Because the GP model may be flawed, a cross-validation technique, like the one used by Ernest [1], is used to check how trustworthy the model is. Only when the cross-validation verification guarantees that the model is sufficiently good, is the EI threshold used to stop the search.

The executor runs experiments either in resources specified by the user for that effect or in under-utilized resources of the database so as not to harm the production workload. To determine whether

the resources are underutilized or not, the following policy is employed *"if the CPU, memory, and disk utilization of the resource for its home use is below* 10% *(threshold t1) for the past* 10 *minutes (threshold t2), then the resource can be used for experiments"*. iTuned's [9] efficiency can be improved by adding some features such as eliminating parameters that affect less the performance and worrying more about the critical ones and using several resources in parallel for running experiments.

Once again, and as it happens for the other systems that were analyzed in the previous section, exploration cost is not taken into account.

Another system for self-tuning of applications, that aims to tune Transactional Memory (TM) implementations for specific workloads and that, just like iTuned [9] doesn't consider exploration cost, is ProteusTM [17]. This system is composed of two components: PolyTM, which is a TM library that dynamically adjusts itself and changes TM implementations according to current requirements and RecTM, that is in charge of finding the optimal TM configuration for a specific workload.

PolyTM has a large variety of TM implementations. For transactions to change the TM implementation they are using, and because having concurrent transactions executing different TM implementations is usually not safe, PolyTM safeguards against this problem by defining a policy which prevents two different transactions from executing two different TM implementations in parallel.

RecTM has three elements: a recommender, that rates unexplored target configurations according to similar explored configurations; a controller, that chooses the configurations to be tried and notifies PolyTM to make changes accordingly; and a monitor, that controls the quality of the explored configurations and that informs the controller of workload changes to initiate new optimization phases.

The recommender uses Collaborative Filtering (CF), described in 2.2.3. In order to rate configurations a Key Performance Indicator (KPI) is used and because there is no information as to the maximum value this indicator can reach for a given workload, it needs to be normalized. To solve this problem the authors of [17] developed a new technique called rating distillation. This technique maps KPI values to a known scale so that CF can use them to rate configurations.

The controller uses BO with EI as acquisition function to select the next point to sample. The samples are modelled using CF and the search stops when the EI is lower than some threshold, the performance achieved in the previous exploration was lower than some bound and the EI of the best experiments to choose decreased in the previous 2 experiments. This way there is a balance between exploration and exploitation.

The monitor gathers KPIs from the current implementations in order to provider some feedback to the controller. By monitoring the KPIs it also detects workload changes. If the performance indicator for the current workload drops suddenly and significantly, the monitor notifies the controller of the need for a change in the TM implementation.

The methodologies used by systems like iTuned [9] and ProteusTM [17] can be applied for tuning the configuration of any system (including cloud based systems). However, they do not take into account specific issues/challenges of cloud systems, which are targeted by systems like CherryPick [2], PARIS [3] or Quasar [4].

2.3.2 Optimizing Resource Allocation in the Cloud

Several systems for optimizing resource allocation try to solve the same problem: ”In which instances should a given workload be deployed in order to maximize either performance or resource utilization?”. This problem can be addressed from the perspective of the users, directing the aim to finding the best configurations for deployment of those users' applications, i.e. the cheapest ones and that offer the best performance with respect to some metric, or from the perspective of the cloud providers, trying to achieve an efficient utilization of available resources..

Systems focusing on the providers tend to disregard exploration and deployment costs, while focusing on finding which resources are needed for each workload and how the several workloads that are running on the same machine may interfere with each other. An example of such a system is Quasar [4], that tries to maximize the utility of cloud facilities by determining the resources needed for each workload. When a workload is received, data is collected according to four different categories: scale-up (number of resources per server), scale-out (number of servers), heterogeneity (types of servers) and interference.

When the profiling results are ready to be used, CF techniques, as described in 2.2.3, are used for classification purposes. The system compares the profiling results with the available labels so as to decide how to fully characterize the workload. This characterization is then used to map the workload to the available machines, following a policy of allocating the least amount possible of resources for a given job. This policy allows the search space to be reduced, since the biggest resources are examined first. If smaller resources were analyzed at the beginning, there would be an enormous number of combinations of those resources that would satisfy the constraints. By starting off with the biggest resources, fewer combinations exist, because fewer resources are needed for the deployment.

As the system uses CF techniques, the more jobs it analyzes, the more it learns and the better it can get over time. Moreover, it tries to maximize resource utilization. However, not only the allocations that it suggest may not be close to optimal when it comes to allocation costs, but also there is no differentiation between reserved and on-demand resources.

As a result, researchers suggested HCloud [5], a system for determining which jobs should be allocated to on-demand resources versus reserved resources. In such hybrid systems, as two types of resources are used, knowing how many of each should be acquired is a problem. Another problem is deciding when to map jobs to the on-demand resources versus the reserved resources.

To address the former problem, the authors use Quasar's [4] estimations for resource needs and measure a job's sensitivity to interference, by analyzing its quality, in the same way that is done in Tarcil [23].

As for the latter problem, a policy was defined to map jobs between the resources. This policy was built on top of three principles:

- Reserved resources are utilized before on-demand resources;
- Applications that can be deployed on on-demand resources ought not delay the scheduling on interference sensitive jobs;

- The utilization limits of the reserved instances should be adjusted dynamically so as to reduce the queuing.

Therefore, the dynamic policy that was built devises two limits. To begin with, a soft limit, below which all jobs are mapped to reserved resources. When this limit is reached, the system starts to differentiate between interference prone and insensitive jobs. To make this distinction, the target quality the jobs need and the quality of previously obtained on-demand resources are calculated. If the quality of the on-demand instances is higher than the one needed by jobs, then they are mapped on on-demand resources. Otherwise, they are deployed in the reserved resources. The second limit that is defined by the policy is a hard limit that defines when jobs need to be queued before reserved resources become available. The soft limit is adjusted based on the queuing ratio.

Although this systems already takes into account both types of resources, it is still geared towards maximizing efficiency and utilization of resources, which is beneficial for providers, while still neglecting allocation and deployment costs for users.

Systems like Proteus [6] whose goal is to exploit the availability of transient excess idle resources made available by providers and that can be revoked when needed, are directed towards users.

Proteus considers as target application domain the, so called, parameter server framework. This is a popular approach for distributed machine learning jobs, which subdivides available resources in workers and servers. The workers share the current parameter values and training data is divided between them so that they can execute applications concurrently and adjust model parameter values. The servers keep the current values and the workers interact with them to fetch and update these values.

The architecture of Proteus comprises two main component: AgileML, a reconfigurable parameter server system, and BidBrain, which is responsible for managing resources.

AgileML takes care of promoting and demoting nodes according to their reliability, by working in three stages. Initially, a list of the reliable and transient nodes is received. Transient nodes are those that can be revoked, namely spot instances in the case of Amazon Web Services (AWS) (c.f. Section 2.1), while reliable resources are all those upon which the user has full control, i.e., he has the resources until he shuts them down.

In stage 1, workers are located in every machine, but only reliable machines become servers. In this way, if transient machines are revoked, there are no losses because all the information is kept in the reliable machines. However, when the ratio of workers to servers becomes too high, there is a network bottleneck, because the servers cannot deal with all the requests from the workers. To prevent this, the system moves to stage 2.

In the second stage, some transient machines become active servers and reliable machines become back-up servers. Now the workers interact with the active servers, that push all the information in bulk to the back-up servers. Once again, recovery from revocations and node losses is made possible because all the information is stored in the back-up servers.

In stage 3, all workers are removed from back-up servers. This stage is necessary because running workers in the same machines as back-up servers was found to cause straggler effects.

BidBrain, Proteus' other component, has information about current and historical market prices and

is responsible for deciding when new resources should be acquired. Its primary objective is to minimize cost per unit work. To estimate this cost, BidBrain considers the probability of eviction, which is translated into free computing time, since when a resource is revoked, the price paid in the beginning of the billing hour is refunded. BidBrain considers also the amount of expected useful compute time, that depends on overheads of evictions and additions. BidBrain makes resource allocation decisions periodically (every two minutes) and a few minutes before the end of each billing hour. Each time a decision is pending, a set of instances is analyzed and if an instance lowers the overall expected cost of the set, it is acquired.

This work is a good effort towards aiding users in their choices, however there are some drawbacks. It strives to minimize costs of a specific framework (parameter server), also exploiting spot instance market dynamics. However, it does not tackle the problem of identifying the most efficient platform and application level configurations to be used. For instance, Proteus [6] does not optimize the choice of the irrevocable reliable machines to be used as servers.

Another class of user-oriented systems, but that are focused on irrevocable resources, are systems like Ernest [1], which take advantage of the fact that multiple jobs have similar structures in terms of computation and communication, hence allowing performance models to be built based on the behavior of those jobs on small samples of data. In order to instantiate the model, the Optimal Experimental Design (OED) technique described in 2.2.5 was used to decide which experiments to run, so as to achieve near optimal results, i.e. to build an accurate model, as fast as possible.

To determine how accurate the model is, and because there is not much data available, a cross-validation technique is used. Thus, to evaluate the model, if it has m training data points, they are divided into two subsets: a validations set, with 1 data point; and a training set, with $m - 1$ data points. After the division, the training set is used to train the model and the validation set tests it. This process is repeated m times and the results are averaged to produce a final estimation.

The models used by Ernest [1] to characterize applications' performance are specific for analytic jobs and capture only the size of the job (i.e., amount of data to be processed) and the number of machines used. This limits the applicability of Ernest's [1] approach both in terms of application domains and of configuration parameters that it can optimize. Also, the optimality criterion used by Ernest [1] assumes that exploring any configuration has the same cost, which is clearly not the case in cloud settings.

CherryPick [2] leverages Bayesian Optimization (BO), described in 2.2.2, to build performance prediction models that are used to predict high quality configurations, i.e., configurations that minimize cloud usage cost and guarantee application performance. At first sight this may seem like an easy task, however, not only is it difficult to find the right balance between resource prices and the running time of the machines, because sometimes by getting one more machine the time that is saved may offer more gains than what it costs to get that machine and vice-versa, but also there is a restricted amount of information, due to the limited runs of cloud configurations that are afforded to search for the model.

The objective function that is optimized by the system, minimizes the deployment cost of a given configuration, expressed as a function of the time the machines are up and how much they cost per hour, with a time constraint to guarantee application performance. GPs, c.f. Section 2.2.1, are used as a prior function to model the data points and EI is used as acquisition function.

The system begins by sampling three starting points using a quasi-random sequence, which will give an estimate of the cost function that is to be minimized. Then, a confidence interval is computed with BO and the next best points to experiment, according to their EI, are selected. Finally, if the improvement that can be achieved with further exploration is less than some defined threshold and if a sufficient number of configurations have been explored, the best configuration has been found. If not, this procedure is repeated until those conditions are met.

Although CherryPick [2] minimizes the deployment cost for users and doesn't require a model for each instance type, it has some issues. First, the cost of exploring all configurations until the best is found is not taken into account directly by the model. Also, the search space considered in CherryPick's [2] evaluation is relatively small, consisting of only 66 configurations. Even so, CherryPick [2] requires exploring 6-9 configurations, corresponding to approximately 10% of the whole search space. Considering the data reported in Table 2.1, and the possibility of including in the optimization process additional application level configuration parameters, it is reasonable to expect that systems like CherryPick [2] would require exploring a significantly larger number of configurations.

A different approach to the same problem is PARIS [3], which also provides performance estimates with minimal data collection but, instead of BO, uses the random forests technique described in 2.2.4.

The work flow of this system will now be introduced. Initially, the user provides a representative task of the workload, the desired performance metric and a set of candidate VM types. PARIS [3] then outputs predictions for the costs and performances of the instances provided by the user. To make these predictions, PARIS [3] needs to know the resource requirements of the workload and how the different VM types affect workloads such as the current one. However, trying this workload on all machines is too expensive. So PARIS [3] has the modelling task divided in two phases.

In an offline phase, which only runs one time, extensive benchmarking of various workloads with each VM type is done. Detailed system performance metrics, fitting broad categories such as CPU utilization, network utilization, disk utilization, memory utilization and system-level features are collected. Each time there are new instance types, the benchmark only has to be rerun on those new instances. Once all this data has been collected, a series of decision trees are trained for each workload and a forest is built.

The online phase runs the representative task provided by the user on 2 pre-defined reference VMs and collects performance metrics and resource usage information. When the forests have been trained according to the user specified performance metrics, the information collected during the profiling phase and a VM configuration are fed to the forest, which outputs the mean and 90^{th} percentile performances. This process is repeated to all candidate VM types. To obtain the cost of choosing those instances, it is assumed that the cost is a function of the performance metric and of the cost per hour of that instance, which is assumed to be known.

Thus, PARIS [3] produces a performance-cost trade-off map that aids users when choosing VM instances. The main drawbacks of PARIS' [3] methodology is that its accuracy is strongly affected by the correct choice of reference configurations and by the representativeness of the data in the training set. Once again, the only cost that is considered is the deployment cost, while in terms of exploration nothing is taken into account directly in the model.

Table 2.3 classifies the systems that were analyzed according to the following dimensions:

- whether they are user-centric or provider-centric
- their optimization goal
- techniques they use to reach the optimization goal
- whether they take advantage of spot markets
- whether their model takes into consideration the deployment cost of the configuration
- whether their model takes into consideration the cost of exploring the configurations.

	Orientation	**Optimization Goal**	**Optimization/ Modelling Technique(s)**	**Spot Market (Historical Data)**	**Considers Deployment Cost**	**Considers Exploration Cost**
Proteus [6]	User	Minimize cost $\times$ job done at steady state	time-series based prediction	Yes	Yes	No
Ernest [1]	User	Minimize Estimation Error	Optimal Experimental Design	No	No	No
CherryPick [2]	User	Minimize execution cost subject to a performance constraint (time)	Bayesian Optimization	No	Yes	No
PARIS [3]	User	Minimize generic function of performance metric and VM cost	Random Forest Models	No	Yes	No
Quasar [4]	Provider	Maximize resource utilization subject to QoS constaints	Classification Techniques (Collaborative Filtering)	No	No	No
HCloud [5]	Provider	Determine which jobs should be mapped to reserved versus on-demand resources	Uses Quasar's results (classification techniques are implicit)	No	No	No

Table 2.3: Comparison between the state-of-the-art system implementations

Overall, although in the literature a number of useful ideas and powerful modelling techniques have been proposed, none of the existing systems attempts to identify the optimal balance between the cost of exploration and the gains achievable during the exploitation phase.

2.4 Travelling Salesman Problem (TSP)

As mentioned earlier, the TSP might be useful in helping us solve the problem of choosing which configurations to experiment in order to find the optimal one.

The TSP, a classic algorithm problem in the field of computer science, is the following: *"a salesman is required to visit once and only once each of n different cities, starting from a base city, and returning to this city. What path minimizes the total distance travelled by the salesman?"*. This problem is NP-hard, which means it cannot be solved in polynomial time (unless P = NP).

Several considerations can be made: the TSP can be modelled as a weighted graph, in which the vertices are the cities and the edges are the paths; it can also be thought of as asymmetric, in which case the distances of the same path in opposite directions are different, or as symmetric, the distances of a path are the same in both directions.

Throughout the years multiple solutions for the problem have been proposed [24]. Exact algorithms that find the exact solution for the TSP have high complexities and are, therefore, impractical for large search spaces. For instance, the brute force technique, which is an exact algorithm that consists in trying all possible combinations of paths, is already unfeasible for only 20 cities. Dynamic programming algorithms also compute exact solutions [25]. Branch-and-Bound algorithms [26], also in the group of exact algorithms, enumerate candidate solutions by means of a state space search.

Heuristics techniques, on the other hand, are able to find a solution $2 - 3\%$ away from the optimal solution, with provable guarantees of the distance of the solution that is obtained to the optimal one, and consider millions of cities.

There are several classes of heuristics algorithms. One of these classes is iterative improvement, such as the k-opt heuristics or Lin-Kernighan heuristics [27], which approach a solution by progressive approximation. These heuristics use the k^{th} iteration to find the value for the k^{th+1} and can find optimal solutions for problems with 10000 cities.

Another class of heuristics is constructive heuristics, which start with an empty subset that is updated in each iteration with the best possible element at that time. This is a greedy strategy.

Randomized improvement heuristics use ant colony algorithms, which leverage the way ants behave to intelligently compute solutions [28]. A set of cooperating agents, ants, deposit pheromones in the edges of the graph while they are building the solution. In this way, by interpreting each others pheromones they can communicate and build a near-optimal solution.

Our problem consists of exploring the multiple configurations in the least costly way, probing the ones that are similar and that do not require extra machine allocations as groups and then moving to other groups of configurations. The shift between these groups of configurations ought to be in an order which is the least expensive.

Mapping our problem to the TSP requires some adjustments, since TSP is a very specific problem with very specific constraints, while ours is more relaxed. For instance, the requirement of returning to the initial city is not necessary in our case. We first define that each configuration corresponds to a city that has to be visited and the way to change between configurations as the path. These paths have costs that may be different considering the direction of the path that is followed, therefore, in our case, the problem is asymmetric.

2.5 Summary

The increasing use of cloud services and the competition between providers led to an increment in the number of machines available for deployment of jobs and applications by the users. The search space of current machines available was shown in Section 2.1, namely in Table 2.1. With such a broad selection, between machines that may have similar characteristics non distinguishable by the inexperienced user, the need for systems that automated this choice began to arise.

Furthermore, since the performance of an application on a given configuration is not known a priori, finding the optimal configuration requires testing and experimenting several configurations, which is both time consuming and expensive.

Hereupon, researchers developed the current self-tuning systems for optimizing the configuration of complex (cloud) systems that were analyzed in Section 2.3. Although these systems find a near-optimal configuration for deployment of user specific workloads, and some of them, such as CherryPick [2], even consider the deployment cost of the configurations in the model, none is concerned with the exploration cost, which can be quite expensive.

3

Work Proposal

This dissertation proposes to fill the gaps identified in Chapter 2, as a result of the analysis of the literature on self-tuning systems for cloud-based applications.

Specifically, this dissertation aims to address a key limitation of state-of-the-art approaches: focusing only on maximizing the quality of the final/steady state configuration, without keeping into account the cost dynamics associated with the exploration of applications' and platforms' configurations in the cloud.

With this work we intend to tackle this problem by developing a technique aimed to strike an optimal balance between the cost associated with the exploration phase and the quality/cost of the final configuration identified by the self-tuning process.

The remainder of this chapter is devoted to illustrating the key ideas and research directions that we plan to pursue to fulfill this goal.

3.1 Optimization Criterion

Generally speaking, the proposed system will consider a user-centric optimization criterion that will seek to minimize the cost of both the exploration phase and of the exploitation phase, i.e., the period of time during which the application will be deployed in the finally recommended configuration. Additionally, the optimization criterion could take into account additional user-defined constraints on, e.g., the fulfillment of minimum Quality of Service (QoS) levels during the exploitation phase and/or the duration/cost of the exploration phases.

Intuitively, the optimization criterion should ensure that the cost of exploring a configuration should not be larger than the expected gain stemming from using that configuration (instead of the currently known optimum) during the exploitation phase.

Clearly, this formulation is based on the assumption that it is possible to estimate the duration of

the exploitation phase. As already discussed, the duration of the exploitation phase is actually application/domain dependent. In the simplest case, it could simply be a fixed, a priori known period of time during which, say, the system is expected to provide some service. Alternatively, the duration of the exploitation phase could coincide with the time between two subsequent optimizations of the same system, triggered, e.g., by workload changes. An even more complex case occurs when the efficiency of the recommended configuration affects the duration of the exploitation phase. This is the case, for instance, of jobs that need to process a fixed amount of data and whose processing rate depends on the overall quality of the system's configuration: the faster is the configuration used at steady state, the shorter is the job duration and, hence, the duration of the exploitation phase.

Our plan is to consider first the simpler scenario in which the duration of the the exploitation phase is assumed to be fixed and a priori known.

Next, and time allowing, we will also investigate the more complex case in which the duration of the exploitation phase is not only unknown but also of the configuration identified via the exploration phase. In fact, we argue that, in order to tackle this more complex case, one could employ the same model that is used to predict the quality of unexplored configurations in order to forecast also the expected duration of the exploitation phase (and accordingly instantiate the optimization problem).

3.2 Predicting QoS of Unexplored Configurations

As already discussed in Chapter 2, existing systems have explored a wide range of modelling techniques to predict the QoS (e.g., response time or throughput) of cloud-based applications in untested configurations. These include, e.g., solutions based on Collaborative Filtering (CF) [4], generic black-box regressors like random forests [3] and Gaussian Processs (GPs) [2], and custom models inspired by the structure of the target application [1].

The methodology that we intend to propose is, indeed, orthogonal to the choice of the specific modelling approach used to predict the quality of unexplored configurations. We argue, indeed, that the proposed approach could be used in combination with different modelling techniques.

Yet, in order to evaluate our proposal, we plan to use a modelling approach based on Gaussian Processs (GPs), analogously to what is done, e.g., in systems like CherryPick [2]. GPs have the key advantage of being (relatively) fast to train and query, while naturally providing a means to estimate the modelling uncertainty (c.f. Section 2.2.1). The last aspect is particularly useful when one wants to compute the Expected Improvement (EI) associated with untested configurations.

3.3 Reasoning on the Cost of the Exploration Phase

As already discussed in Chapter 1, the cost of exploring a set S of candidate configurations, in cloud environments, depends both on the current configuration and on the order with which the configurations in S are actually explored.

As already hinted, these cost dynamics can be conveniently captured using a graph-oriented model, where we associate each configuration with a different node of the graph and encode the cost of testing

configuration c1 from configuration c2 via a directed[1] edge between the corresponding graph's nodes. It appears reasonable, at this stage, to assume that the costs associated with each edge in this graph can be estimated a priori, based on the type of system's re-configuration required. For instance, the economical cost of system's re-configurations entailing acquiring additional VM instances can be easily estimated based on the unit cost of each VM instance and on the expected duration of the time window during which that configuration will have to be monitored, in order to measure its quality — another parameter that, for simplicity, we shall assume to be a priori known and homogeneous across the configurations' space.

This graph-based formulation of the problem opens the possibility to employ techniques developed in the literature on the TSP in order to plan the most efficient way in which to explore the set of candidate configurations. It should be noted that the problem at hand corresponds, indeed, to a relaxed variant of TSP. Unlike in the original TSP formulation, in fact, it simply suffices to visit/test all the cities/configurations, and it is not needed to return to the initial configuration/city.

Another crucial factor affecting the cost of the exploration phase is predicting how many configurations shall be explored, i.e., the cardinality of the candidate set S. We plan to tackle this problem by using the (GP-based) model that predicts the expected quality of individual untested configurations to compute the aggregate EI stemming from visiting multiple configurations.

We plan to address this problem by building on recent results in the area of parallel Bayesian Optimization (BO) [29–32], which have developed theoretical frameworks to evaluate the, so called, multi-points expected improvement, i.e., EI for a set of q points or q-EI, and to predict the points that maximize the q-EI. In particular, the MOE-qEI toolkit appears to be a valuable resource, as it contains a high quality implementation of a recent technique by Wang et al. [29] that solves the problem of identifying the most promising (i.e., Bayes-optimal [10]) next q configurations that should be explored, given a GP-based model trained over a set of (already explored) configurations.

More in detail, we propose to leverage this technique to identify the configurations in candidate sets of increasing size, noted q, and compute the corresponding q-EI. This information can then be used to identify, via the aforementioned variant of the TSP, the most cost efficient order of exploration of the q candidate configurations.

Overall, this approach allows to predict both the exploration cost (via TSP) and the exploitation cost (via their q-EI) of candidate sets having size q=1, 2, and, hence, the quality of these candidate solutions. One can finally determine the optimum size q^* of a candidate set in various way, ranging from sweeping all values of q up to a some static upper bound (corresponding to the maximum look-ahead factor) to more sophisticated search techniques based, e.g., on stochastic gradient descent [33–35].

3.4 Putting all Pieces Together

The pseudo-code of Algorithm 1 describes, in a concise way, the various steps of the proposed technique.

[1] As, generally speaking, the cost of exploring a configuration c' starting from a configuration c differs from the cost of exploring c starting from c'.

Our system will begin by sampling some initial configurations from the search space. We will consider both simple random strategies for this initial sampling, as well as popular alternatives like Latin Hyper-Cube Sampling (LHS) [21, 22].

Then, a GP model will be fit to the sampled data. This GP model will be used to estimate the function mapping the application's configuration parameters (including number, type of VM instances, as well as middleware/framework specific parameters) to their corresponding QoS.

Next, we evaluate the solution quality of candidate sets, C_q, having increasing size q = 1, $\cdots$, Max_look-ahead. For each value of q, first the configurations in each candidate set C_q are obtained using the technique by Wang et al [29], along with its q-EI. Next, the sampling order that minimizes the exploration cost of C_q is obtained using the previously mentioned TSP-variant. Based on this information, it is possible to predict both the exploration and the exploitation costs of the candidate sets, and accordingly pick the best one, C_q*.

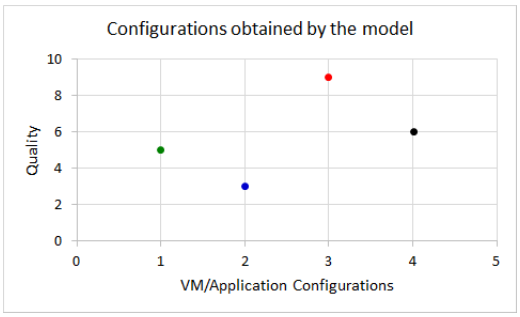
Once the optimal exploration strategy for the next q* configurations is established, we determine whether it is pointless to proceed with the exploration, by checking if the expected cost saving, after exploring the configurations in C_q* is below some tunable threshold.

If it appears that further exploration would be worthy, the configurations in C_q* are visited according to their (previously computed) minimum cost order. The GP model, however, can be updated as soon as a (or some) new configuration is sampled and, based on the new knowledge incorporated by the model, a different candidate set can be computed. This means that, although the system optimizes its exploration strategy looking ahead q* steps, it actually revises its exploration strategy after every single exploration step.

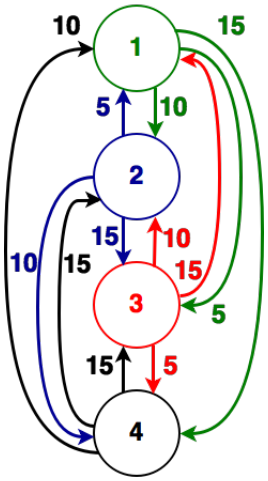
Algorithm 1: Pseudo-code of the various steps for the system

```
1  Sample initial configurations;
2  Fit a GP model to the samples;
3  for q = 1, ···, max_look-ahead do
4  |   Obtain C_q via MOE-qEI;
5  |   Compute EI by visiting C_q;
6  |   Find minCostTour(C_q);
7  |   if cost(C_q) < cost(pred_optimum) then
8  |   |   pred_optimum = C_q
9  if |cost(pred_optimum) − known_optimum| < threshold then
10 |   end;
11 Sample first configuration in minCostTour(C_q);
12 pred_optimum = known_optimum; //prepare for next cycle
13 Return to step 2;
```

In Figure 3.1 a concrete example of how the order in which the configurations are explored influences the overall cost of exploration is shown. Considering that the VM/application configurations in the graph of Figure 3.1 (a) (configurations 1, 2, 3 and 4) are the ones with highest quality in that iteration of the exploration phase, that the costs to change from one configuration to another are the ones shown in Figure 3.1 (b) and that the system is in configuration 2, it is clear to see that the overall cost of exploring will be lower if the configurations are explored in the order $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$.



(a) Example of an output from the model



(b) Example of exploration with TSP

Figure 3.1: Exploration of configurations

3.5 Work Plan

For the work proposed previously in this Chapter, the following calendar structure, depicted in Figure 3.2, is planned. Our system will be compared with similar systems, such as CherryPick [2] and PARIS [3], which already account for the exploitation cost, considering the following metrics: gains in terms of QoS vs. exploitation cost and savings achieved due to the optimal exploration phase. We will use workloads consisting of different Machine Learning (ML) models implemented using the tensorflow library [36], which require the tuning of a large number of hyper-parameters, to test the system.

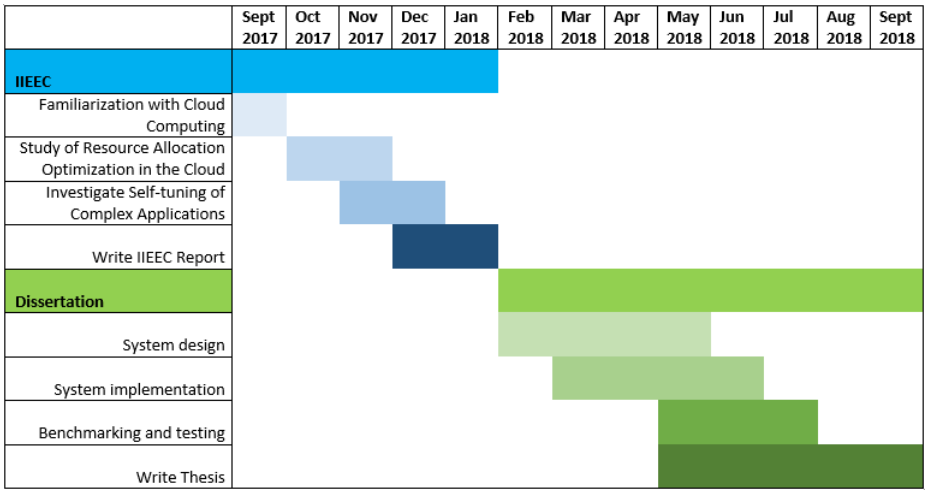


Figure 3.2: Calendar structure for the proposed work

4

Conclusions

When deploying complex applications in the cloud, one is faced with the problem of selecting the best set of machines, and the best configurations of both these machines and of the application parameters, to optimally run the application. Finding the best configuration is a complex problem, since cloud providers offer a wide variety of machines at different prices and, thus, discovering the near-optimal configuration requires searching a large space.

This problem is augmented by the fact that it is not known a priori how the application will perform on a given configuration. To acquire this knowledge, the deployment of the application in that configuration is required. This task corresponds to the exploration phase of current state-of-the-art systems, that aim to find the optimal configuration for deployment of users' jobs. All of them neglect the cost of exploring the configurations, which is, in fact, non-negligible and can be quite expensive, not only in short running jobs, but also in applications that are subject to frequent workload changes and require constant optimizations.

Our goal is to devise a system that tackles both the problems of exploration and exploitation, minimizing its costs while still providing acceptable QoS. We propose to achieve this by leveraging innovative techniques in the field of parallel Bayesian Optimization (BO) allied with a relaxed variant of the TSP to ensure an efficient order of exploration.

References

[1] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 363–378. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930635

[2] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

[3] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz, "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 452–465. [Online]. Available: http://doi.acm.org/10.1145/3127479.3131614

[4] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541941

[5] ——, "Hcloud: Resource-efficient provisioning in shared cloud systems," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 473–488. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872365

[6] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 589–604. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064182

[7] C. K. Williams and C. E. Rasmussen, "Gaussian processes for regression," in *Advances in neural information processing systems*, 1996, pp. 514–520.

[8] C. E. Rasmussen, M. Kuss *et al.*, “Gaussian processes in reinforcement learning.” in *NIPS*, vol. 4, 2003, p. 1.

[9] S. Duan, V. Thummala, and S. Babu, “Tuning database configuration parameters with ituned,” *PVLDB*, vol. 2, no. 1, pp. 1246–1257, 2009. [Online]. Available: http://www.vldb.org/pvldb/2/vldb09-193.pdf

[10] E. Brochu, V. M. Cora, and N. de Freitas, “A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning,” *CoRR*, vol. abs/1012.2599, 2010. [Online]. Available: http://arxiv.org/abs/1012.2599

[11] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, “Boa: The bayesian optimization algorithm,” in *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, ser. GECCO’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 525–532. [Online]. Available: http://dl.acm.org/citation.cfm?id=2933923.2933973

[12] R. Lam, K. Willcox, and D. H. Wolpert, “Bayesian optimization with a finite budget: An approximate dynamic programming approach,” in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 883–891. [Online]. Available: http://papers.nips.cc/paper/6188-bayesian-optimization-with-a-finite-budget-an-approximate-dynamic-programming-approach.pdf

[13] M. D. Ekstrand, J. T. Riedl, J. A. Konstan *et al.*, “Collaborative filtering recommender systems,” *Foundations and Trends® in Human–Computer Interaction*, vol. 4, no. 2, pp. 81–173, 2011.

[14] F. Ricci, L. Rokach, and B. Shapira, *Introduction to Recommender Systems Handbook*. Boston, MA: Springer US, 2011, pp. 1–35. [Online]. Available: https://doi.org/10.1007/978-0-387-85820-3_1

[15] G. Linden, B. Smith, and J. York, “Amazon.com recommendations: item-to-item collaborative filtering,” *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, Jan 2003.

[16] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen, *Collaborative Filtering Recommender Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 291–324. [Online]. Available: https://doi.org/10.1007/978-3-540-72079-9_9

[17] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano, “Proteustm: Abstraction meets performance in transactional memory,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’16. New York, NY, USA: ACM, 2016, pp. 757–771. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872385

[18] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[19] S. M. Stigler, “Optimal experimental design for polynomial regression,” *Journal of the American Statistical Association*, vol. 66, no. 334, pp. 311–318, 1971. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/01621459.1971.10482260

[20] S. Silvey, *Optimal design: an introduction to the theory for parameter estimation*. Springer Science & Business Media, 2013, vol. 1.

[21] M. Stein, “Large sample properties of simulations using latin hypercube sampling,” *Technometrics*, vol. 29, no. 2, pp. 143–151, 1987. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/00401706.1987.10488205

[22] M. D. McKay, R. J. Beckman, and W. J. Conover, “A comparison of three methods for selecting values of input variables in the analysis of output from a computer code,” *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979. [Online]. Available: http://www.jstor.org/stable/1268522

[23] C. Delimitrou, D. Sanchez, and C. Kozyrakis, “Tarcil: Reconciling scheduling speed and quality in large shared clusters,” in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC ’15. New York, NY, USA: ACM, 2015, pp. 97–110. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806779

[24] M. Bellmore and G. L. Nemhauser, “The traveling salesman problem: A survey,” *Operations Research*, vol. 16, no. 3, pp. 538–558, 1968. [Online]. Available: https://doi.org/10.1287/opre.16.3.538

[25] R. Bellman, “Dynamic programming treatment of the travelling salesman problem,” *J. ACM*, vol. 9, no. 1, pp. 61–63, Jan. 1962. [Online]. Available: http://doi.acm.org/10.1145/321105.321111

[26] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel, “An algorithm for the traveling salesman problem,” *Oper. Res.*, vol. 11, no. 6, pp. 972–989, Dec. 1963. [Online]. Available: http://dx.doi.org/10.1287/opre.11.6.972

[27] K. Helsgaun, “General k-opt submoves for the lin–kernighan tsp heuristic,” *Mathematical Programming Computation*, vol. 1, no. 2, pp. 119–163, Oct 2009. [Online]. Available: https://doi.org/10.1007/s12532-009-0004-6

[28] M. Dorigo and L. M. Gambardella, “Ant colony system: a cooperative learning approach to the traveling salesman problem,” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 53–66, Apr 1997.

[29] J. Wang, S. C. Clark, E. Liu, and P. I. Frazier, “Parallel Bayesian Global Optimization of Expensive Functions,” *ArXiv e-prints*, Feb. 2016.

[30] D. Ginsbourger, R. Le Riche, and L. Carraro, “A multi-points criterion for deterministic parallel global optimization based on kriging,” 03 2008.

[31] C. Chevalier and D. Ginsbourger, *Fast Computation of the Multi-Points Expected Improvement with Applications in Batch Selection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 59–69. [Online]. Available: https://doi.org/10.1007/978-3-642-44973-4_7

[32] S. Marmin, C. Chevalier, and D. Ginsbourger, "Differentiating the multipoint Expected Improvement for optimal batch design," *ArXiv e-prints*, Mar. 2015.

[33] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[34] J. Kiefer and J. Wolfowitz, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952. [Online]. Available: http://www.jstor.org/stable/2236690

[35] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization Methods for Large-Scale Machine Learning," *ArXiv e-prints*, Jun. 2016.

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467