A Decision Tree Based Approach Towards Adaptive Modeling of Big Data Applications

Ioannis Giannakopoulos*, Dimitrios Tsoumakos§ and Nectarios Koziris*
* *Computing Systems Laboratory, School of ECE, NTUA, Athens, Greece, {ggian, nkoziris}@cslab.ece.ntua.gr*
§ *Department of Informatics, Ionian University, Corfu, Greece, dtsouma@ionio.gr*

Abstract—The advent of the Big Data era has given birth to a variety of new architectures aiming at applications with increased scalability, robustness and fault tolerance. At the same time these architectures have complicated application structure, leading to an exponential growth of their configuration space and increased difficulty in predicting their performance. In this work, we describe a novel, automated profiling methodology that makes no assumptions on application structure. Our approach utilizes oblique Decision Trees in order to recursively partition an application's configuration space in disjoint regions, choose a set of representative samples from each subregion according to a defined policy and return a model for the entire space as a composition of linear models over each subregion. An extensive evaluation over real-life applications and synthetic performance functions showcases that our scheme outperforms other state-of-the-art profiling methodologies. It particularly excels at reflecting abnormalities and discontinuities of the performance function, as well as identifying the parameters with the highest impact on the application's behavior.

I. INTRODUCTION

Performance modeling is a well-researched problem [1], [2], [3]. The identification of an application's behavior under different configurations is a key factor for it to be able to fulfill its objectives. As the application landscape evolves, mainly due to the emergence of the Big Data era, new architectures and design patterns have enabled an increasing number of applications to be deployed in a distributed manner and benefit from the merits of this approach: Scalability, robustness and fault-tolerance are some of the properties that render distributed platforms attractive and explain their wide adoption. By virtue of their design, distributed applications are commonly deployed to cloud infrastructures [4], in order to combine their inherent characteristics with the power of the cloud: Seemingly infinite compute and storage resources, dynamically allocated and purchased, enable a Big Data application, i.e., an application that processes Big Data, to scale in a cost effective way. However, the adoption of the distributed paradigm increases the complexity of the application architecture: Many assisting software modules that support coordination, cluster management, etc., are essential for the application to run properly. Yet, each module can be configured in numerous ways. As such, the application configuration space has vastly expanded. Hence, the problem of modeling an application performance, also called an *application profile*, has become particularly complicated.

Evidently, the automated estimation of an application profile would prove highly beneficial. The magnitude of the configuration space renders exhaustive approaches that explore a massive part of the configuration space impractical, since they entail both a prohibitive number of deployments and an enormous amount of computation. Several approaches target to model application performance in an analytical way [5], [6], [7], [8]. These approaches are effective for known applications with specific structure and they are based on simulation or emulation techniques [9], [10]. To overcome the rigidness of these schemes, other methods (e.g., [11], [12], [13]) take a "black-box" approach, in which the application receives a set of inputs, corresponding to the different factors of the configuration space and produces one or more outputs, corresponding to its performance. These approaches try to identify the relationship between the input and the output variables for a subset of the configuration space (utilizing sampling) and generalize the findings with Machine Learning techniques (modeling).

Such profiling approaches require multiple application deployments for distinct configurations in order to efficiently explore the configuration space and capture its behavior. However, the policy employed for selecting these configurations, is a decisive factor. This dimension of the profiling problem, though, is not addressed by current research works, as most of the suggested *application profiling* approaches assume a random configuration selection policy, implying that all input dimensions have the same impact on the application's behavior, and aim at minimizing the modeling error through increasing the number of examined configurations. Therefore, the employment of an adaptive configuration selection policy, that wisely picks the tested configurations with respect to investigating the most "representative" application configurations, would result in more accurate application profiles with fewer tested configurations, accelerating the profiling process and minimizing the respective cost.

In this work, we propose an adaptive approach to automatically generate an application profile for any Big Data application with unknown structure, given a specific number of deployments, particularly focusing on appropriately selecting the tested configurations in order to maximize

the modeling accuracy. Our work tackles the sampling and modeling steps in a unified way: First, we introduce an accuracy-driven sampling technique that favors regions of the configuration space which are not accurately approximated. Second, we decompose the configuration space in disjoint regions and utilize different models to approximate the application performance in each of them. The basis of our approach lies on the mechanics of Classification and Regression Trees [14] that use a recursive partitioning of the application's configuration space. Each partition is assigned with a number of configurations to be deployed, with the process being iterated until a pre-defined maximum number of sample configurations is reached. The number of configurations allocated at each region is adaptively decided according to the approximation error and the size of each partition. Finally, the entire space is approximated by linear models per partition, based on the deployed configurations. Intuitively, our approach attempts to "zoom-in" to regions where application performance is not accurately approximated, paying specific interest to all the abnormalities and discontinuities of the performance function. By utilizing oblique Decision Trees [15], we are able to capture patterns that are affected by multiple configuration parameters simultaneously. In this work we make the following contributions:

•We propose an adaptive, accuracy-driven profiling technique for Big Data applications that utilizes oblique Decision Trees. Our method decomposes the multi-dimensional input space into disjoint regions, naturally adapting to the complex performance application behavior in a fully automated manner. Our scheme utilizes three unique features relative to the standard Decision Tree algorithm: First, it proposes a novel expansion algorithm that constructs oblique Decision Trees by examining whether the obtained samples fit into a linear model. Second, it allows developers to provide a compromise between *exploring* the configuration space and *exploiting* the previously obtained knowledge. Third, it adaptively selects the most accurate modeling scheme, based on the achieved accuracy.

•We perform an extensive experimental evaluation over diverse, real-world applications and synthetic performance functions of various complexities. Our results showcase that our methodology is very effective, achieving modeling accuracies even $3\times$ higher that its competitors and, at the same time, being able to create models that reflect abnormalities and discontinuities of the performance function orders of magnitude more accurately. Furthermore, our sampling methodology proves to be particularly beneficial for linear classifiers, as linear models trained with samples chosen by our scheme present up to 38% lower modeling error.

II. BACKGROUND

A. Problem formulation

Application profiling can be formulated as a function approximation problem [16], [12]. The application is viewed

as a black-box that receives a number of inputs and produces a single (or more) output(s). The main idea behind constructing the performance model is to predict the relationship between the inputs and the output, without making any assumption regarding the application's architecture. Inputs reflect any parameters affecting application performance, such as the number and quality of different types of resources (e.g., cores/memory, number of nodes, etc.), application-level parameters (e.g., cache used by an RDBMS, HDFS block size, etc.), workload-specific parameters (e.g., throughput/type of requests) and dataset-specific parameters (e.g., size, dimensionality, distribution, etc.).

Assume that an application comprises n inputs and one output. We assume that the ith input, $1 \leq i \leq n$, receives values from a predefined finite set of values, denoted as d_i. The Cartesian product of all $d_i, 1 \leq i \leq n$ is the *Deployment Space* of the application $D = d_1 \times d_2 \times \cdots \times d_n$. Similarly, the application's output reflects values that correspond to a performance metric, indicative of its ability to fulfill its objectives. The set of the application's output will be referred to as the application's *Performance Space*. Based on the definitions of D and P, we define the performance model m of an application as a function $m : D \to P$. The estimation of the performance model entails the estimation of the performance value $b_i \in P$ for each $a_i \in D$. However, $|D|$ increases exponentially with n (as $|D| = \prod_{i=1}^{n} |d_i|$), thus the identification of all performance values becomes prohibitive, both in terms of time and cost. A common approach to tackle this challenge is the extraction of a subset $D_s \subseteq D$ ($|D_s| \ll |D|$) and the estimation of the performance points P_s for each $a_i \in D_s$. Using D_s and P_s, model m can be approximated, creating an approximate model m'. While $m' \to m$, the approximation is more accurate.

Note that this formulation is valid only under the assumption that distinct deployments are *reproducible*, i.e., in case where a given Deployment Space point is redeployed, the measured outcome is identical or at least very similar. For many reasons, such as the interference [17], network glitches, etc., such an assumption can be violated when the application is deployed to cloud environments, because of the introduced unpredictability that distorts the application's behavior. The treatment of this dimension of the problem is outside the scope of our work. This works tackles the complexity introduced by the excessive dimensionality of the Deployment Space. The presented methodology can be, thus, directly applied to fairly predictable environments with reduced interference, such as private cloud installations that, according to [18], remain extremely popular, since they will host half of the user generated workloads for 2017, maintaining the trend from the previous years.

B. Decision Trees

Classification and Regression Trees (CART) [14], or Decision Trees (DT), are a very popular classification and

regression approach. They are formed as tree structures containing intermediate (*test*) and *leaf* nodes. Each test node represents a boundary of the data space and each leaf node represents a class, if the DT is used for classification, or a linear model, if used for regression. The boundaries of the DT divide the original space into a set of disjoint regions. The construction of a DT is based on recursively partitioning the space of the data so as to create disjoint groups of points that maximize their intra-group homogeneity and minimize their inter-group homogeneity. The homogeneity metric of a group differs among the existing algorithms: *GINI impurity* has been used by the CART algorithm [14], *Information Gain* has been used by ID3 and C4.5 [19] for classification, whereas the Variance Reduction [14] is commonly used for regression. These heuristics are applied to each leaf to decide which dimension should partitioned and at which value. The termination condition of the DT construction varies between different algorithms as the tree height is either pre-defined or dynamically decided, i.e., the tree grows until new leaves marginally benefit its accuracy.

Each boundary of a DT is parallel to one axis of the data, since it involves a single dimension of the data space, i.e., the boundary line is expressed by a rule of the form $x_i = c$, where c is a constant value. Generalizing this rule into a multivariate line, we obtain the *oblique* DTs [15] that consist of lines of the form: $\sum_{i=1}^{n} c_i x_i + \gamma = 0$. The multivariate boundaries boost the expressiveness of a DT, since non axis-parallel patterns can be recognized and expressed. In this paper, we utilize oblique DTs in two ways: First, they are employed to create an approximate model of the performance function. Second, their construction algorithm is modified to adaptively sample the Deployment Space of the application, focusing more on regions where the application presents a complex behavior and ignoring regions where it tends to be easily predictable.

III. PROFILING METHODOLOGY

A. Method overview

The main idea of the suggested algorithm is to partition the Deployment Space by grouping samples that better fit different linear models, infer knowledge about the performance function through sampling the Deployment Space, deploying the selected configurations and, when a predefined number of deployments is reached, model the performance function utilizing different linear models for each partition. Specifically, at each step, the Deployment Space is partitioned by grouping the already obtained samples according to their ability to create a linear model that accurately approximates the application performance. Estimates are then created regarding the intra-group homogeneity, which corresponds to the prediction accuracy of the performance function for the specified region. Therefore, poorly approximated regions need to be further sampled in order to be better approximated, whereas accurate regions need not to

be further explored. Intuitively, the suggested algorithm is an attempt to adaptively "zoom-in" to areas of the Deployment Space where the behavior of the performance function is more *obscure*, in the sense that it is unpredictable and hard to model. This enables the number of allowed application deployments to be dynamically distributed inside the Deployment Space, leading to more accurate predictions as more samples are collected for performance areas that are harder to approximate. This smart distribution of the deployed samples requires the closer examination of regions of the Deployment Space independently, something that is inherently conducted by DTs. Their properties, such as their scalability to multiple dimensions, robustness and their innate divide-and-conquer functionality render them a perfect fit for the application profiling problem and facilitate the decomposition of the Deployment Space in an intuitive and efficient manner. In Algorithm 1, we provide the pseudocode of the suggested methodology.

Algorithm 1 DT-based Adaptive Profiling Algorithm

1: **procedure** DTADAPTIVE(D, B, b)
2:    $tree \leftarrow$ TREEINIT($\emptyset$), $samples \leftarrow \emptyset$
3:    **while** $|samples| \leq B$ **do**
4:      $tree \leftarrow$ PARTITION($tree$, $samples$)
5:      $s \leftarrow$ SAMPLE(D, $tree$, $samples$, b)
6:      $d \leftarrow$ DEPLOY(s)
7:      $samples \leftarrow samples \cup d$
8:    $model \leftarrow$ CREATEMODEL($samples$)
9:    **return** $model$

Our Algorithm takes three input parameters: (a) the application's Deployment Space D, (b) the maximum number of samples B and (c) the number of samples selected at each algorithm iteration b. The *tree* variable represents a DT, while the *samples* set contains the obtained samples. While the number of obtained samples is less than B, the following steps are executed: First, the leaves of the tree are examined and tested whether they can be replaced by subtrees that further partition their regions (Line 4). The leaves of the expanded tree are, then, sampled with the *SAMPLE* function (Line 5), the chosen samples are deployed (Line 6) according to the sample's deployment configuration and performance metrics are obtained. Note that, $s \subseteq D$ whereas $d \subseteq D \times P$, i.e., s contains the sampled configurations and $samples$ contains the deployment configurations along with their performance value. Finally, when B samples have been chosen, the final model is created (Line 8).

B. Space Partitioning

Space partitioning occurs through the expansion of the DT, i.e., the replacement of the DT's leaves with test nodes with two new children/leaves. The replacement of a leaf entails the identification of a split line that maximizes the homogeneity between the two newly leaves. Recall that, the utilization of *oblique* DTs allows the test node to be represented by a multivariate line and enhances their adaptability to different performance functions. Nevertheless, calculating

an optimal multivariate split line is NP-complete [15]. Since we want to exploit the expressiveness of the oblique partitions without introducing a prohibitive computation cost for their estimation, we express the problem as an optimization problem [15] and utilize Simulated Annealing (SA) [20] to identify a near-optimal line. In Algorithm 2, we present the PARTITION function. For each leaf of the tree (Line 3), SA is executed to identify the best multivariate split line (Line 5), considering solely the samples whose Deployment Space-related dimensions ($d.in$) lie inside the specified leaf (Line 4). The samples of the specified leaf are then partitioned in two disjoint sets according to their position (Lines 7-11), in which the symbol $\preceq$ indicates that a sample s is located below $line$. Finally, a new test node is generated (Line 12), replacing the original leaf node of the tree and the new tree is returned.

Algorithm 2 Partitioning Algorithm

1: **procedure** PARTITION($tree$, $samples$)
2:    $newTree \leftarrow tree$
3:    **for** $l \in$ leaves($tree$) **do**
4:       $v \leftarrow \{d | d \in samples, d.in \in l\}$
5:       $line \leftarrow SA(v)$
6:       $L_1 \leftarrow \emptyset, L_2 \leftarrow \emptyset$
7:       **for** $s \in v$ **do**
8:          **if** $s \preceq line$ **then**
9:             $L_1 \leftarrow L_1 \cup s$
10:         **else**
11:            $L_2 \leftarrow L_2 \cup s$
12:       $testNode \leftarrow \{line, L1, L2\}$
13:       $newTree \leftarrow$ replace(l, $testNode$)
14:    **return** $newTree$

The cornerstone of SA's effectiveness is the score function that quantifies the efficacy of each candidate solution. The methodologies utilized by various DT construction algorithms make no assumption regarding the nature of the data. Although this enhances their adaptability to different problem spaces, their utilization in this work resulted in poorly partitioned Deployment Spaces. The data that our work attempts to model belong to a performance function. Intuitively, if all the performance function points were available, one would anticipate that a closer observation of a specific region of the Deployment Space would present an approximately linear behavior, as "neighboring" configurations are anticipated to produce similar performance. When focusing on a single neighborhood, this "similar performance" could be summarized by a linear hyperplane. A split line is considered to be "good" when the generated leaves are best summarized by linear regression models or, equivalently, if the samples located in the two leaves can produce linear models with low modeling error. Given a Deployment Space D, a line l and two sets L_1, L_2 that contain D's samples after partitioning it with l (as in lines 6-10 of Algorithm 2), we estimate two linear regression models for L_1 and L_2 and estimate their residuals using the coefficient of determination R^2. l's score is: $Score(l) = -\frac{|L_1|R_{L_1}^2 + |L_2|R_{L_2}^2}{|L_1|+|L_2|}$.

Note that, $0 \leq R^2 \leq 1$ and a value of 1 represents perfect fit to the linear model. $Score(l)$ is minimized when both L_1 and L_2 generate highly accurate linear models or, equivalently, if the two sets can be accurately represented by two linear hyperplanes. We weight the importance of each set according to the number of samples they contain as an inaccurate model with more samples has greater impact to the accuracy than an inaccurate model with fewer samples. The negative sign is employed in order to remain aligned with the literature, in which SA seeks for minimum points. Finally, one SA property not discussed this far is the ability to achieve a customizable compromise between the modeling accuracy and the required computation. Specifically, prior to SA's execution, the user defines a maximum number of iterations to be executed for the identification of the best split line. When one wants to maximize the quality of the partitioning, many iterations are conducted. On the contrary, the number of iterations is set to lower values in order to allow for a rapid split line estimation.

C. Adaptive Sampling

Algorithm 3 Sampling algorithm

1: **procedure** SAMPLE(D, $tree$, $samples$, b)
2:    $errors, sizes \leftarrow \emptyset, maxError, maxSize \leftarrow 0$
3:    **for** $l \in$ leaves($tree$) **do**
4:       $points \leftarrow \{d | d \in samples, d.in \in l\}$
5:       $m \leftarrow$ regression($points$)
6:       $errors[l] \leftarrow$ crossValidation(m,$points$)
7:       $sizes[l] \leftarrow |\{e | e \in D \cap l\}|$
8:       **if** $maxError \leq errors[l]$ **then**
9:          $maxError \leftarrow errors[l]$
10:       **if** $maxSize \leq sizes[l]$ **then**
11:          $maxSize \leftarrow sizes[l]$
12:    $scores, newSamples \leftarrow \emptyset, sumScores \leftarrow 0$
13:    **for** $l \in$ leaves($tree$) **do**
14:       $scores[l] \leftarrow w_{error} \cdot \frac{errors[l]}{maxError} + w_{size} \cdot \frac{sizes[l]}{maxSize}$
15:       $sumScores \leftarrow sumScores + scores[l]$
16:    **for** $l \in$ leaves($tree$) **do**
17:       $leafNumDeps \leftarrow \lceil \frac{scores[l]}{sumScores} \cdot b \rceil$
18:       $s \leftarrow$ RANDOMSELECT($\{d | d \in D \cap l\}$, $leafNumDeps$)
19:       $newSamples \leftarrow newSamples \cup s$
20:    **return** $newSamples$

After the tree expansion, the SAMPLE function is executed (Algorithm 3). The algorithm iterates over the leaves of the tree (Line 3). For the samples of each leaf, a new linear regression model is calculated (Line 5) and its residuals are estimated using Cross Validation [21]: The higher the residuals, the worse the fit of the points to the linear model. The size of the specified leaf is then estimated. After storing both the error and the size of each leaf into a map, the maximum leaf error and size are calculated (Lines 8-11). Subsequently, a score is estimated for each leaf (Lines 13-15). The score of each leaf is set to be proportional to its scaled size and error. This normalization is conducted so as to guarantee that the impact of the two factors is equivalent. Two coefficients w_{error} and w_{size} are used to assign different weights to each measure. These scores are accumulated and used to proportionally distribute b to each

leaf (Lines 16-19). In that loop, the number of deployments of the specified leaf is calculated and new samples from the subregion of the Deployment Space are randomly drawn with the *RANDOMSELECT* function, in a uniform manner. Finally, the new samples set is returned.

The consideration of two factors (error and size) for deciding the number of deployments spent at each leaf targets the trade-off of *exploring* the Deployment Space versus *exploiting* the obtained knowledge, i.e., focus on the abnormalities of the space and allocate more points to further examine them. This is a well-known trade-off in many fields of study [22]. In our approach, one can favor either direction by adjusting the weights of leaf error and size, respectively. Note that this scheme enables the consideration of other parameters as well, such as the deployment cost through the extension of the score function (Line 14). This way, more "expensive" deployment configurations, e.g., ones that entail multiple VMs with many cores, would be avoided in order to regulate the profiling cost.

D. Modeling

After B samples are returned by the profiling algorithm, they are utilized by the *CREATEMODEL* (Algorithm 1, Line 8) function to train a new DT. The choice of training a new DT instead of expanding the one used during the sampling phase is made to maximize the accuracy of the final model. When the first test nodes of the former DT were created, only a short portion of the samples were available to the profiling algorithm and, hence, the original DT may have initially created inaccurate partitions. Moreover, in cases where the number of obtained samples is comparable to the dimensionality of the Deployment Space, the number of constructed leaves is extremely low and the tree degenerates into a linear model that covers sizeable regions of the Deployment Space with reduced accuracy. To overcome this limitation, along with the final DT, a set of Machine Learning classifiers are also trained, keeping the one that achieves the lowest Cross Validation error. However, when the DT is trained with enough samples, it outperforms all the other classifiers. This is the main reason for choosing the DT as a base model for our scheme: The ability to provide higher expressiveness by composing multiple linear models in areas of higher unpredictability, make them a perfect choice for modeling a performance function. Note that, the linearity of this model does not compromise its expressiveness: Non-linear performance functions can also be accurately approximated by a piecewise linear model, through further partitioning the Deployment Space.

Finally, we provide an example of execution of Algorithm 1 for the Wordcount operator. The operator is executed for a 100G synthetic dataset, on a Hadoop cluster of varying sizes, i.e., the Deployment Space is 1-d and represents the number of nodes of the Hadoop cluster (4–256 nodes) and the Performance Space represents the execution time. We execute

our methodology for a budget of $B = 14$ configurations and $b = 7$ for each iteration. In Figures 1 (a) and (b) we depict the actual and approximated performance functions for the first and second algorithm iterations, respectively.
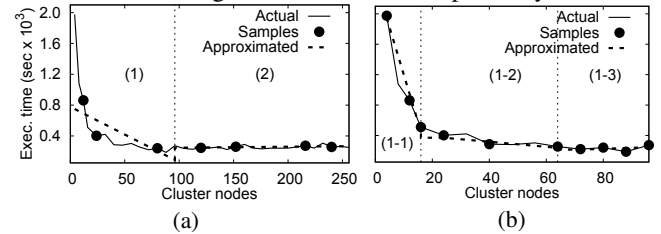

Figure 1. Algorithm execution for Wordcount

During the first iteration, 7 samples are randomly picked and the first partitioning of the Deployment Space takes place (for a cluster of 96 nodes) generating the areas (1) and (2) of Figure 1 (a). Note that, the performance function in area (2) presents linear behavior and, hence, is extremely accurately approximated with only 4 tested configurations. On the contrary, the performance function is strongly non-linear in area (1) and, hence, poorly approximated. The scores of the two Deployment Space areas are, then, calculated, and in the following iteration all the 7 available configurations are assigned to area (1), because the error of area (2) is practically zero (and $w_{size} = 0$). In Figure (b), only area (1) is depicted, as the rest of the space remains intact. The new samples enforce space partitioning which, makes the model accurately approximate the actual performance function using a piecewise linear function. The final model approximates the actual performance function extremely accurately, examining only a mere 5% of the Deployment Space. This example is indicative of our methodology's power: Our divide-and-conquer approach allows us to sample and model each region separately, assigning different level of detail to different regions of the Deployment Space. Moreover, the piecewise linear function is extremely effective for the performance functions of typical Big Data applications and operators, since the behavior observed in this example (i.e., the "Actual" line) is commonly encountered, especially when the Deployment Space consists of resource-related dimensions. This enables us to provide extremely accurate approximations with a minimal number of deployments. Finally, even when addressing performance functions of high complexity and strongly non-linear behavior, the utilization of more detailed partitions (as in the area (1) of Figure 1 (a)) guarantees that the actual line can be accurately approximated through the deployment of more samples.

IV. EXPERIMENTAL EVALUATION

A. Methodology and data

To evaluate the accuracy of our profiling algorithm, we test it over various real and synthetic performance functions. The *Mean Squared Error* (MSE) metric is utilized for the comparison, estimated over the *entire* Deployment Space, i.e., we exhaustively deploy all possible combinations

of each application's configurations, so as to ensure that the generated model successfully approximates the original function for the entire space. We have deployed four different popular real-world Big Data operators and applications, summarized in Table I. We opt for applications with diverse characteristics with Deployment Spaces of varying dimensionality (3 – 7 dimensions). The first three operators are implemented in Spark (k-means, Bayes) and Hadoop (Wordcount) and they are deployed to a YARN cluster. In all cases, the performance metric corresponds to the execution time. MongoDB is deployed as a sharded cluster and it is queried using YCSB [23]. The sharded deployment of MongoDB consists of three components: (a) A configuration server that holds the cluster metadata, (b) a set of nodes that store the data (MongoD) and (c) a set of loadbalancers that act as endpoints to the clients (MongoS). Each application was deployed in a private Openstack cluster with 8 nodes aggregating 200 cores and 600GB of RAM. Due to space constraints, in Appendix B of the Extended Version of this work [24], a thorough application analysis is provided.

Table I
APPLICATIONS UNDER PROFILING

Application (perf. metric)	Dimensions	Values
Spark k-means (execution time)	YARN nodes	2–20
	# cores per node	2–8
	memory per node	2–8 GB
	# of tuples	200–1000 ($\times 10^3$)
	# of dimensions	{1,2,3,5,10}
	data skewness	5 levels
	k	{2,5,8,10,20}
Spark Bayes (execution time)	YARN nodes	4–20
	# cores per node	2–8
	memory per node	2–8 GB
	# of documents	0.5–2.5 ($\times 10^6$)
	# of classes	50–200 classes
Hadoop Wordcount (execution time)	YARN nodes	2–20
	# cores per node	2–8
	memory per node	2–8 GB
	dataset size	5–50 GB
MongoDB (throughput)	# of MongoD	2–10
	# of MongoS	2–10
	request rate	5–75 ($\times 10^3$) req/s

We have also generated a set of synthetic performance functions, listed in Table II. The listed functions are chosen with the intention of testing our profiling algorithm against multiple scenarios of varying complexity and dimensionality. To quantify each function's complexity, we measure how accurately can each function be approximated by a linear hyperplane. Linear performance functions can be approximated with only a handful of samples, hence we regard them as the least complex case. For each of the listed functions, we calculate a linear model that best represents the respective data points and test its accuracy using the coefficient of determination R^2, whose value indicates the linearity of the respective function (1 indicating total linearity and 0 the opposite). Based on R^2 values for each case, we generate three complexity classes: Functions of *LOW* complexity (when R^2 is higher than 0.95), functions of *AVERAGE*

complexity (when R^2 is between 0.5 and 0.7) and functions of *HIGH* complexity when R^2 is close to 0. The values of R^2 depicted in Table II refer to two-dimensional Deployment Spaces.

Table II
SYNTHETIC PERFORMANCE FUNCTIONS

| Complexity	Name	Function	R^2
LOW	LIN	$f_1(\mathbf{x}) = a_1 x_1 + \cdots + a_n x_n$	1.00
	POLY	$f_2(\mathbf{x}) = a_1 x_1^2 + \cdots + a_n x_n^2$	0.95
AVG	EXP	$f_3(\mathbf{x}) = e^{f_1(\mathbf{x})}$	0.65
	EXPABS	$f_4(\mathbf{x}) = e^{|f_1(\mathbf{x})|}$	0.62
	EXPSQ	$f_5(\mathbf{x}) = e^{-f_1(\mathbf{x})^2}$	0.54
HIGH	GAUSS	$f_6(\mathbf{x}) = e^{-f_2(\mathbf{x})}$	0.00
	WAVE	$f_7(\mathbf{x}) = \cos(f_1(\mathbf{x})) \cdot f_3(\mathbf{x})$	0.00
	HAT	$f_8(\mathbf{x}) = f_2(\mathbf{x}) \cdot f_6(\mathbf{x})$	0.00

B. Profiling algorithms comparison

First, we compare our profiling methodology against other end-to-end profiling schemes. Our approach is referred to as DT-based Adaptive methodology (*DTA*). Active Learning [25] (*ACTL*) is a Machine Learning field that specializes on exploring a performance space by obtaining samples assuming that finding the class or the output value of the sample is computationally hard. We implemented *Uncertainty Sampling* that prioritizes the points of the Deployment Space with the highest uncertainty, i.e., points for which a Machine Learning model cannot predict their class or continuous value with high confidence. PANIC [12] is an adaptive approach that favors points belonging into steep areas of the performance function, utilizing the assumption that the abnormalities of the performance function characterize it best. Furthermore, since most profiling approaches use a randomized sampling algorithm [11], [16], [13] to sample the Deployment Space and different Machine Learning models to approximate the performance, we implement a profiling scheme where we draw random samples (*UNI*) from the performance functions and approximate them using the models offered by WEKA [26], keeping the most accurate in each case. In all but a few cases, the Random Committee [27] algorithm prevailed, constructed using Multi-Layer Perceptron as a base classifier. For each of the aforementioned methodologies, we execute the experiments 20 times and present the median of the results.

1) Sampling rate: We first compare the four methods against a varying Sampling Rate, i.e., the portion of the Deployment Space utilized for approximating the performance function ($SR = \frac{|D_s|}{|D|} \times 100\%$). SR varies from 3% up to 20% for the tested applications. In Figure 2 we provide the accuracy of each approach measured in terms of MSE.

Figure 2 showcases that DTA outperforms all the competitors for increasing SR, something indicative of its ability to distribute the available number of deployments accordingly so as to maximize the modeling accuracy. In more detail, all algorithms benefit from an increase in SR since the error metrics rapidly degrade. Both in k-means and Bayes, when the SR is around 3% UNI and DTA construct models of the

highest accuracy. As mentioned in Section III-D, for such low SR the linearity of the DT would fail to accurately represent the relationship between the input and the output dimensions, thus a Random Committee classifier based on Multi-Layer Perceptrons is utilized for the approximation. The same type of classifier also achieves the highest accuracy for the rest of the profiling algorithms (ACTL, PANIC) that also present higher errors due to the less accurate sampling policy at low SR. As SR increases, the DT obtains more samples and creates more leaves, which contributes in the creation of more linear models that capture a shorter region of the Deployment Space and, thus, producing higher accuracy. Specifically, for $SR \geq 3\%$, DT created a more accurate prediction than other classifiers and was preferred.

Figure 2. Accuracy vs sampling rate (MSE)
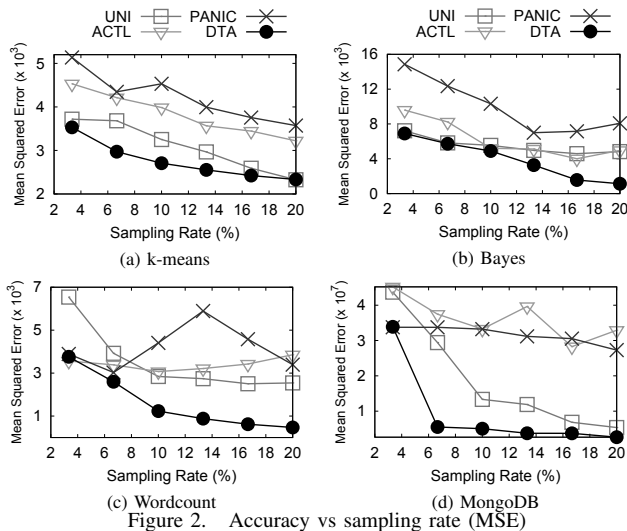
In the rest of the cases, DTA outperforms its competitors for the Wordcount application and, interestingly, this is intensified for increasing SR. Specifically, DTA manages to present $3\times$ less modeling error than UNI when $SR = 20\%$. Finally, for the MongoDB case, DTA outperforms the competitors increasingly with SR. In almost all cases, DTA outperforms its competitors and creates models even 3 times more accurate (for Bayes when $SR = 20\%$) from the best competitor. As an endnote, the oscillations in PANIC's and ACTL's behavior are explained by the aggressive exploitation policy they implement. PANIC does not explore the Deployment Space and only follows the steep regions, whereas ACTL retains a similar policy only following the regions of uncertainty, hence the final models may become overfitted in some regions and fail to capture most patterns of the performance function. Our work identifies the necessity of both exploiting the regions of uncertainty but also for exploring the entire space. This trade-off is only addressed by DTA and explains its dominance for difficult to approximate applications.

2) Performance function complexity and dimensionality: We now compare the accuracy of the profiling algorithms against synthetic profiling functions with varying complex-

ity and dimensionality. We create synthetic performance functions using the ones presented in Table II for 2 and 10 dimensions, employing 2 different SR (0.5% and 2%). Specifically, we assume that there exist 2 dimensions that highly impact the function's output and, in the 10-d case, the remainder dimensions are of much lower significance, i.e., their coefficients tend to zero. This simulates the real-world use case where one wants to profile an application with many dimensions, but only a handful of them are, indeed, significant to the output performance. The results are depicted in Table III. Since the output of each dataset is in different scale, we normalize all the results, dividing the error of each methodology with the error produced by UNI. All methodologies approximated LIN, POLY, EXP and EXPSQ with minimal error and hence are not provided. The lowest errors for each case are demonstrated in bold.

Table III
ACCURACY VS FUNCTION COMPLEXITY

	n	ACTL		PANIC		DTA	
		0.5%	2%	0.5%	2%	0.5%	2%
EXPABS	2	1.99	5.29	1.29	2.63	**0.58**	**0.52**
	10	0.40	0.31	0.33	0.28	**0.13**	**0.11**
GAUSS	2	1.55	5.91	1.32	5.63	**0.68**	**0.52**
	10	**0.42**	0.39	0.45	0.42	**0.42**	**0.36**
WAVE	2	0.98	2.34	1.01	2.51	**0.71**	**0.79**
	10	0.30	0.28	0.31	0.28	**0.18**	**0.15**
HAT	2	1.17	6.49	1.25	4.85	**0.60**	**0.42**
	10	0.52	0.43	0.49	0.45	**0.32**	**0.29**

Table III showcases that *all* the synthetic functions were more accurately approximated by DTA than the rest of the profiling schemes. Specifically, when the dimensionality of the space is low, DTA achieves considerably lower modeling errors than the UNI case, whereas ACTL and PANIC produce much worse results, compared to UNI, something intensified with increasing SR. This is ascribed to the fact that ACTL and PANIC only exploit the space abnormalities whereas UNI focuses on space exploration. In cases where the space consists of 2 dimensions with high impact, UNI is, understandably, better than the exploitation-centric ACTL and PANIC approaches, DTA standing in the middle and producing the best results. Interestingly, though, when the dimensionality of the space increases, UNI becomes insufficient, as in these cases all the other approaches exhibited a considerably more accurate behavior. The spaces now consist of 10 dimensions, the 8 of which are of low importance: All the approaches that focus on the abnormalities of the space (ACTL and PANIC) benefit from their exploitation-based functionality since they ignore deployment space regions with uninteresting behavior. DTA, on the other hand, achieves the same results, but for another reason: The construction of the DT during the sampling and modeling phases, is constantly ignoring the unimportant dimensions, executing a form of *Dimensionality Reduction* to the deployment space and only focusing on the dimensions of higher interest. This functionality renders our approach suitable both for cases where there exist several non-interesting dimensions that need to be ignored

and cases where there exist equally important dimensions that should be evaluated. This discussion highlights that DTA manages to elegantly compromise the contradicting aspects of exploration and exploitation. Furthermore, the recursive Deployment Space partitioning manages not only to distribute the deployment budget appropriately, but also ignore the dimensions that present no interest.

C. Parameter Impact Analysis

1) Per-iteration number of deployments: We now evaluate b's impact, i.e., the number of deployments spent in each algorithm iteration, in DTA's performance. In Figure 3, we model k-means and set b as a portion of B (the total number of allowed deployments) for three different SR. The horizontal axis represents the ratio B/b. The Figure on the left depicts the MSE while the right one represents the respective execution time of DTA.
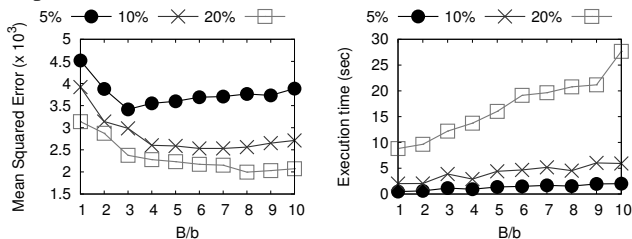

Figure 3. Accuracy vs B/b

When $B/b = 1$, the algorithm degenerates into UNI, since the tree is constructed in one step. At this point, the algorithm presents the highest error and the lowest execution time, since the tree is only constructed once and the most erroneous leaves are not prioritized. When the ratio increases, the algorithm produces more accurate results and the error decrease intensifies for increasing SR. For example, when $SR = 20\%$ the error decreases more than 35% for increasing B/b. However, when $SR = 5\%$ and $SR = 10\%$ and for low b values (e.g., $B/b = 10$) an interesting pattern appears: MSE starts to increase, neutralizing the effect of lower b. This occurs in cases where b is extremely small, compared to the dimensionality of the Deployment Space. In such cases, the per-iteration budget becomes too small in order to be efficiently distributed among the DT leaves, and hence, many iterations need to take place before the tree is further expanded and analyzed. This leads to a suboptimal distribution of the b samples and, finally, a suboptimal configuration selection policy.

The performance gain presented by increasing B/b is ascribed to two factors: (a) the final DT has more accurate cuts and, hence, well-placed models and (b) the samples are properly picked during the profiling. To isolate the impact of each factor, we repeat the same experiment for all applications and train different ML classifiers instead of a DT. This way, we isolate the impact of sampling and only examine its importance. We identified that the classifiers that consist of linear models, i.e., regression classifiers such

as OLS (Ordinary Least Squares) [28], or an Ensemble of Classifiers created with Bagging [29] (BAG) that utilize linear models as base improve their accuracy with increasing B/b. In Table IV we provide our findings for two classifiers (OLS and BAG), expressing the percentage decrease of MSE of each case, compared to the $B/b = 1$ case.

Table IV
MSE DECREASE % FOR LINEAR CLASSIFIERS

Application	Classifier	B/b			
		2	5	8	10
k-means	OLS	8%	10%	12%	12%
	BAG	3%	8%	8%	9%
Bayes	OLS	23%	27%	28%	29%
	BAG	23%	22%	21%	23%
Wordcount	OLS	23%	17%	19%	23%
	BAG	21%	28%	41%	38%
MongoDB	OLS	19%	13%	12%	7%
	BAG	6%	12%	11%	2%

The table demonstrates that for all cases, increasing B/b benefits the linear models, something that showcases that our sampling algorithm itself achieves better focus on the interesting Deployment Space regions. Nevertheless, we notice that an increasing B/b ratio does not always lead to linear error reduction. When $B/b = 8$ and $B/b = 10$, one can notice that the error either degrades marginally (for k-means and Bayes) or slightly increases (for MongoDB) when compared to $B/b = 5$. This is, again, ascribed to the extremely low per-iteration number of deployments b that almost equals the dimensionality of the space. In summary, our findings demonstrate that even utilizing solely the sampling part of our methodology can be particularly useful in cases where linear models, or a composition of them, are employed.

2) Oblique boundaries: We now evaluate the impact of flat versus oblique DTs in the profiling accuracy and execution time. In Figure 4 (a), we model k-means for varying SR. The oblique tree introduces a slight accuracy gain (of about 10% for low SR) compared to the flat tree and demands about twice the time for algorithm execution. The accuracy gains fade out when the SR increases. This is due to the fact that when a higher SR is employed, more leaves are created and the profiling algorithm functions in a more fine-grained manner. At this point, the structure of the leaf nodes is not important. However, when the algorithm must work with fewer leaves, i.e., lower SR, the importance of the leaf shape becomes crucial, hence the performance boost of the oblique cuts. Nevertheless, a point not stressed in the results so far is that oblique DTs, apart from a minor performance boost, offer the ability to accurately approximate points of the performance function with patterns spanning to multiple dimensions of the Deployment Space. When measuring the accuracy of the model in the entire space, as in Figure 4 (a), this effect can be overlooked, yet, it is evident when focusing on the region containing the pattern. To showcase this, we conduct the following experiment: Assume EXPABS of Table II, minimized when $a_1x_1 + \cdots + a_nx_n = 0$. We again compare the flat and oblique approaches for a two-

dimensional Deployment Space but now use two different test sets: (i) points from the entire space (as before) depicted in Figure 4 (b) and (ii) points close to the aforementioned line (less than $\epsilon = 10^{-3}$) in Figure 4 (c).
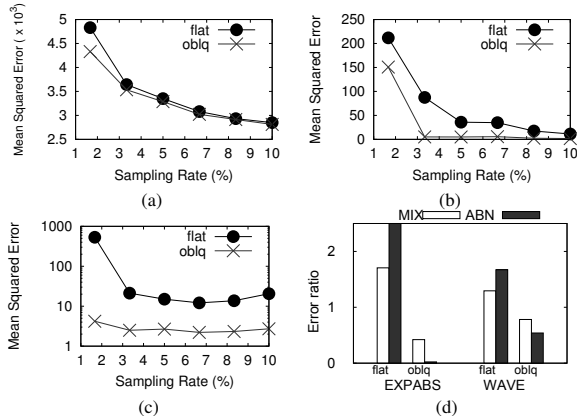

Figure 4. Accuracy vs flat and oblique cuts

While the difference in (b) is considerable for this case (a gain of 30% to 90%), when testing against points close to the abnormality we observe that the oblique version produces errors that are orders of magnitude lower than those of the flat case. To further evaluate the impact of the test set into the measured accuracy for different performance functions, we repeat the previous experiment for EXPABS and WAVE (that also contains similar complex patterns) for a $SR = 2\%$. We measure the accuracy of the model when the test points are picked from (a) the entire Deployment Space (ALL), (b) close to the "abnormal" region (ABN) and (c) both (MIX). For MIX, half of the test points are far from the abnormality and the other half is located in a distance less than ϵ. Figure 4 (d) showcases the respective results. The produced errors for MIX and ABN are divided with the error of ALL for each execution. Our results demonstrate that when more points are picked around the abnormality region, the flat DT produce higher modeling errors and oblique DT achieve much lower errors. A similar to the flat DT behavior is also verified for the rest of the profiling algorithms as well (UNI, ACTL and PANIC): None of them was able to approximate the abnormal regions of the synthetic functions with satisfying accuracy, rendering DTA the only profiling methodology with this feature.

V. RELATED WORK

Performance modeling is a vividly researched area. The distinct approaches used to model the behavior of a given application can be assigned in three categories: (a) simulation-based, (b) emulation-based and (c) approaches involving the benchmarking of the application and take a "black-box" view. In the first case, the approaches are based on known models of the cloud platforms [30] and enhance them with known performance models of the applications under profiling. CDOSim [5] is an approach that targets to model the Cloud Deployment Options (CDOs) and simulate the cost

and performance of an application. CloudAnalyst [6] is a similar work that simulates large distributed applications and studies their performance for different cloud configurations. Finally, WebProphet [7] is a work that specializes in web applications. These works assume that performance models regarding both the infrastructure and the application are known, as opposed to our approach that makes no assumptions neither for the application nor for the infrastructure.

The idea of emulation-based approaches is to deploy the application and capture performance traces for various scenarios, which are then "replayed" to the infrastructure in order to to predict the performance. CloudProphet [31] is an approach used for migrating an application into the cloud. It collects traces from the application running locally and replays them into the cloud, predicting the performance it should achieve over the cloud infrastructure. //Trace [9] is an approach specializing in predicting the I/O behavior of a parallel application, identifying the causality between I/O patterns among different nodes. In [10] a similar approach is presented, in which a set of benchmark applications are executed in a cloud infrastructure, measuring microarchitecture-independent characteristics and evaluating the relationship between a target and the benchmarked application. According to this relationship, a performance prediction is extracted. Finally, [8] specializes to I/O-bound BigData applications, generating a model of the virtualized storage through microbenchmarking and generalizing it to predict the application performance. Although emulation approaches can be extremely efficient for capturing specific aspects of an application's behavior, they cannot be generally applied if the application structure is unknown.

The final category of profiling methodologies view the application as a black-box that receives a number of inputs and produces a number of outputs, corresponding to performance metrics. The application is deployed for some representative deployment configurations and performance metrics are obtained, utilized by machine learning techniques to map the configuration space to the application performance. In [11], [16] a generic methodology is proposed used to infer the application performance of based on representative deployments of the configuration space. The approach tackles the problem of generalizing the performance for the entire deployment space, but does not tackle the problem of picking the most appropriate samples from the deployment space, as the suggested approach. PANIC [12] is a similar work, that addresses the problem of picking representative points during sampling. This approach favors the points that belong to the most steep regions of the Deployment Space, based on the idea that these regions characterize most appropriately the entire performance function. However it is too focused on the abnormalities of the Deployment Space and the proposed approach outperforms it. Similarly, the problem of picking representative samples of the Deployment Samples is also addressed by Active Learning [25].

This theoretical model introduces the term of uncertainty for a classifier that, simply put, expresses its confidence to label a specific sample of the Deployment Space. Active Learning favors the regions of the Deployment Space that present the highest uncertainty and, as PANIC, fail to accurately approximate the performance function for the entire space, as also indicated by our experimental evaluation. Finally, in [13] two more generic black-box approaches are provided, utilizing different machine learning models for the approximation. Neither of these works, though, address the problem of picking the appropriate samples, since they are more focused on the modeling problem.

VI. Conclusions

In this work, we revisited the problem of performance modeling of Big Data applications. Their configuration space can grow exponentially large, making the required number of deployments for good accuracy prohibitively large. We proposed a methodology that utilizes oblique Decision Trees to recursively partition and sample the Deployment Space, assuming a maximum number of deployments. Our approach manages to adaptively focus on areas where the model fails to accurately approximate application performance, achieving superior accuracy under a small number of deployments. We demonstrated that our method better approximates both real-life and synthetic performance functions.

References

[1] N. H. Kapadia, J. A. Fortes, and C. E. Brodley, "Predictive application-performance modeling in a computational grid environment," in *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*. IEEE, 1999.

[2] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005.

[3] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, "Predictive performance and scalability modeling of a large-scale application," in *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM, 2001.

[4] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing - The business perspective," *Decision support systems*, vol. 51, no. 1, 2011.

[5] F. Fittkau, S. Frey, and W. Hasselbring, "CDOSim: Simulating cloud deployment options for software migration support," in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*. IEEE, 2012.

[6] B. Wickremasinghe, R. N. Calheiros, and R. Buyya, "Cloudanalyst: A cloudsim-based visual modeller for analysing cloud computing environments and applications," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 2010.

[7] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang, "WebProphet: Automating Performance Prediction for Web Services." in *NSDI*, vol. 10, 2010.

[8] I. Mytilinis, D. Tsoumakos, V. Kantere, A. Nanos, and N. Koziris, "I/O Performance Modeling for Big Data Applications over Cloud Infrastructures," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015.

[9] M. P. Mesnier, M. Wachs, R. R. Simbasivan, J. Lopez, J. Hendricks, G. R. Ganger, and D. R. O'hallaron, "//Trace: parallel trace replay with approximate causal events." USENIX, 2007.

[10] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006.

[11] M. Gonçalves, M. Cunha, N. C. Mendonca, and A. Sampaio, "Performance Inference: A Novel Approach for Planning the Capacity of IaaS Cloud Applications," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015.

[12] I. Giannakopoulos, D. Tsoumakos, N. Papailiou, and N. Koziris, "PANIC: Modeling Application Performance over Virtualized Resources," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015.

[13] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta, "Modeling virtualized applications using machine learning techniques," in *ACM SIGPLAN Notices*, vol. 47, no. 7. ACM, 2012.

[14] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[15] D. Heath, S. Kasif, and S. Salzberg, "Induction of oblique decision trees," in *IJCAI*, 1993.

[16] M. Cunha, N. Mendonça, and A. Sampaio, "Cloud Crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds," *Concurrency and Computation: Practice and Experience*, 2016.

[17] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, 2014.

[18] "RightScale 2017 State of the Cloud Report," https://www.rightscale.com/lp/2017-state-of-the-cloud-report.

[19] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, 1986.

[20] P. J. Van Laarhoven and E. H. Aarts, "Simulated annealing," in *Simulated Annealing: Theory and Applications*. Springer, 1987.

[21] S. Geisser, *Predictive inference*. CRC press, 1993, vol. 55.

[22] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 1998.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010.

[24] I. Giannakopoulos, D. Tsoumakos, and N. Koziris, "A Decision Tree Based Approach Towards Adaptive Profiling of Distributed Applications (Extended Version)," *arXiv preprint arXiv:1704.02855*, 2017.

[25] B. Settles, "Active learning literature survey," *University of Wisconsin, Madison*, vol. 52, no. 55-66, 2010.

[26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, 2009.

[27] L. Rokach, "Ensemble-based classifiers," *Artificial Intelligence Review*, vol. 33, no. 1-2, 2010.

[28] C. Dismuke and R. Lindrooth, "Ordinary least squares," *Methods and Designs for Outcomes Research*, vol. 93, 2006.

[29] J. R. Quinlan, "Bagging, boosting, and c4. 5," in *AAAI/IAAI, Vol. 1*, 1996.

[30] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, 2011.

[31] A. Li, X. Zong, S. Kandula, X. Yang, and M. Zhang, "CloudProphet: towards application performance prediction in cloud," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011.