Elastic Management of Cloud Applications using Adaptive Reinforcement Learning

Konstantinos Lolos, Ioannis Konstantinou, Verena Kantere and Nectarios Koziris
School of Electrical and Computer Engineering, National Technical University of Athens
Email: {klolos, ikons, nkoziris}@cslab.ece.ntua.gr, verena@dblab.ece.ntua.gr

***Abstract*— Modern large-scale computing deployments consist of complex applications running over machine clusters. An important issue in these is the offering of elasticity, i.e., the dynamic allocation of resources to applications to meet fluctuating workload demands. Threshold based approaches are typically employed, yet they are difficult to calibrate and optimize. Approaches based on reinforcement learning (RL) have been proposed, but they require a large number of states in order to model complex application behavior. Methods that adaptively partition the state space have been proposed, but their partitioning criteria and strategies are sub-optimal. In this work we present *MDP_DT*, a novel full-model based reinforcement learning algorithm for elastic resource management that employs adaptive state space partitioning. We propose two novel statistical criteria and three strategies and we experimentally prove that they correctly decide both where and when to partition, outperforming existing approaches. We experimentally evaluate *MDP_DT* in a real large scale cluster over variable not-encountered workloads and we show that it takes more informed decisions compared to static, model-free and threshold approaches, while requiring a minimal amount of training data. We experimentally show that this adaptation enabled *MDP_DT* to optimize the achieved profit while being 40% cheaper than calibrated RL and threshold approaches.**

I. Introduction

Modern large-scale computing environments, like large private clusters, cloud providers and data centers may have deployed tens of platforms, like NoSQL and traditional SQL database servers, web servers, etc on thousands of machines, and run on them hundreds of services and applications [25]. A vital issue in such environments is the allocation of resources to platforms and applications so that they are neither over-provisioned, nor under-provisioned, aiming to avoid both resource saturation and idling, and having as utmost goal fast execution of user workload while keeping the cost of operating the infrastructure as low as possible.

Managing the above trade-off and achieve a truly elastic behavior is quite challenging for multiple reasons. First, the number of system and application parameters that affect behavior (i.e., performance) is exceedingly large; therefore, the number of possible states of the system, which correspond to combinations of different values for all such parameters is exponentially large. Facebook for instance deal with this complexity with a proprietary highly sophisticated distributed system designed only for configuration management [21], whereas Google's Borg [25] manages hundreds of thousands of jobs deployed in tens of thousands of machines.

Second, the value range or the interesting values of such parameters may not be known; moreover, many of these parameters are continuous instead of discrete (e.g. cluster and load characteristics, live performance metrics, etc.), making it necessary to devise ahead techniques for their discretization in a way that the number of discrete values is kept small, but ranges of continuous values that lead to different system behavior correspond to different discrete values. Third, most often we do not know if and how a parameter, or a parameter value set affects the system behavior. Therefore, we do not know if changing the value of this parameter will make an impact, and furthermore, a desirable impact to the system behavior. Fourth, the time interval between two consecutive resource management decisions is usually at least in the order of minutes, reducing the collection rate of training data. Nevertheless, the resource management technique should be able to work with little such data. All four challenges are hard to address even for static workloads and applications, and become insurmountable for dynamic ones.

Since the issue of resource management in elastic environments is so vital and challenging, there are numerous efforts to address it in both the industry and research. Public cloud providers such as Amazon, Google, Microsoft and IBM offer autoscaling services [1]. These employ threshold-based rules to regulate infrastructural resources (e.g., if mean CPU usage is above 40% then add a new VM). However, such solutions do not address any of the four challenges discussed above. Some research approaches to this issue also explore threshold-based solutions [8], [10], [15], [20], [22]. More sophisticated approaches employ *Reinforcement Learning* algorithms such as *Markov Decision Processes* (MDP) and *Q-Learning*, algorithms which are natural solutions for decision making problems and offer optimality guarantees under conditions. These approaches suffer from important limitations that derive from the assumption of a priori knowledge of parameters and their role to system behavior, as well as from the curse of dimensionality as a result of their effort to create a full static model of the computing environment [2], [4], [5], [9], [16], [19], [23].

In this work we address all four challenges of elastic resource management in a large-scale computing environment by employing RL in a novel manner that starts from one global state that represents the environment, and gradually partitions this into finer-grained states adaptively to the workload and the behavior of the system; this results in a state space that has coarse states for combinations of parameter values (or value ranges) for which the system has unchanged behavior

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

and finer states for combinations for which the system has different behavior. Therefore, the proposed technique is able to zoom into regions of the state space in which the system changes behavior, and use this information to take decisions for elastic resource management. Specifically, the proposed technique works as follows: **Adoption of a full model.** Since decisions are taken in intervals in the order of minutes, it is realistic to maintain a full MDP model of the system. Information about the behavior of the system is acquired at a slow rate, limiting the size of the model and making possible expensive calculations for each decision. **Adaptive state space partitioning.** We create a novel decision tree-based algorithm, called *MDP_DT*[1] that dynamically partitions the state space when needed, as instructed by the behavior of the system. This allows the algorithm to work on a multi-dimensional continuous state space, but also to adjust the size of the state space based on the amount of information on the system behavior. The algorithm starts with one or a few states, and the MDP model is trained with a small amount of data. As more data on the behavior of the system are acquired, the number of states dynamically increases, and with it increases the accuracy of the model. **Splitting criteria and strategies.** The algorithm can take as input criteria for splitting the states, which aim to partition the existing behavior information, with respect to the measured parameter values, into subsets that represent the same behavior. We propose two novel criteria, the *Parameter test* and the *Q-value*. Also, the algorithm can adopt strategies that perform splitting of one or multiple states, employing small or big amounts of information on behavior. **Reuse of information on system behavior.** It is essential that we do not waste collected information. Therefore, if an old state is replaced with two new ones, the information used to train the old state is re-used to train the new ones. This way, even though new states are introduced, these are already trained and their values already represent all the experiences acquired since the start of the model's life. The contributions of this work are:

- The *MDP_DT* algorithm that performs adaptive state space partitioning for elastic resource management.
- Two splitting criteria, the *Parameter test* and the *Q-value test* that decide how the system behavior changes w.r.t. the measured parameter values, and split states.
- Three splitting strategies, *Chain Split*, *Reset Split* and *Two-phase Split*, which can be used in combination depending on the modeled computing environment.
- A thorough experimental study on simulation that provides insight on the behavior of the *MDP_DT* algorithm, the splitting criteria, employed statistical tests and splitting strategies, and allow the calibration of the algorithm for optimal performance.
- An extensive experimental study on a real large-scale elastic cluster based on the calibration resulted from the simulation study, which proves the effectiveness of *MDP_DT* in taking optimal resource management decisions fast, while taking into account tens of parameters, and its superiority in comparison with classical full model-based and model-free-based algorithms.

[1] Available at https://github.com/klolos/reinforcement_learning

TABLE I: Terminology

Term	Type	Semantics
collect_measurements	function	collects real system measurements
state	vector	stores state values with respect to values in collect_measurements
select_action	function	selects an action given the current state s
m, m'	vector	an ordered set of real parameter measurements
r	real	stores the value of a reward
get_reward	function	calculates the reward given a measurement m
a	integer	stores the value of an action
e	vector	stores the measurements m, m' for states s, s' respectively, the action a and the reward r from s to s'
experiences	2D vector	stores the experiences e for a pair of states s and s'
transitions	3D vector	stores the number of transitions from state s to s' taking action a
rewards	3D vector	stores the accumulated reward r for transitions from state s to s' taking action a
optimal_action	function	returns the optimal action a for a state s

In the rest of this paper: Section II describes the proposed *MDP_DT* algorithm as well as splitting criteria and strategies. Section III presents the experimental study on simulation. Section IV presents experiments on a real cloud environment and Section V summarizes related work. Section VI concludes the paper. A more detailed technical report of this paper can be found in [11].

II. The MDP_DT Algorithm

In this section we present our novel algorithm for elastic resource management based on adaptive state space partitioning and the use of decision trees. In [11] we outline the basic notations regarding Markov Decision Processes (MDPs) and we discuss preliminary knowledge. In a typical RL setting, the world is assumed to be in one of a finite number of *states*, and from each state a number of *actions* are available. Upon the execution of an action, a scalar reinforcement is received and the world transitions to a new state. An algorithm is *optimal* in the sense that it chooses actions that maximize some predefined long-term measure of the reinforcements. We select a *model-based* approach, i.e. the agent attempts to find the exact behavior of the world, in the form of the transition and reward functions, and then calculate the optimal policy using dynamic programming approaches [3], or using alternative algorithms such as *Prioritized Sweeping* [14] to decrease the required calculation in each step. A full-model approach requires a small training set in order to converge to an optimal policy, which makes it a preferable choice.

A. Overview

Goal. The computing environment consists of the *system* and the *workload*, and the *system resources*, which are elastic and are used to accommodate the workload execution. The parameters of the system resources and the parameters of the workload are used to model the environment. These can be multiple, behave in an interrelated or independent manner, play a significant or insignificant role in the performance of workload execution, and may be related to:

Parameters of system resources: cluster size, the amount of VMs, the amount of RAM per VM, the number of virtual CPUs (VCPUs) per VM, the storage capacity per VM, network characteristics, etc.

Parameters of the workload: CPU and network utilization, I/O reqs/sec, average job latency, etc.

Our goal is to accommodate the workload execution by adapting in a dynamic and online manner the system resources, so that it is executed efficiently and resources are not over-provisioned. We need to do this without knowing in advance the characteristics, role and interaction of the parameters of system resources and workload.

Motivating Example: A start-up company hosts services in a public IaaS cloud. The company employs a distributed data-store (for instance, a NoSQL database) to handle user-generated workload that consists, e.g., of a mix of read and write requests. The administrator wants to optimize a given business policy, typically of the form "maximize performance while minimizing the cost of operating the infrastructure" under variable and unpredictable loads. For example, if the company offers user-facing low-latency services, the performance can be measured based on throughput, and the cost can be the cost of renting the underlying IaaS services. In order to achieve this she wants to use an automated mechanism to perform one or both of the following: (i) scale the system, (e.g., change the cluster size, the RAM size, etc.) or (ii) reconfigure the system (e.g., increase cache size, change replication factor, etc). Such a mechanism may implement a rule-based technique, like one of the aforementioned frameworks [1], which monitors the value of representative performance parameters, such as CPU, RAM usage, incoming workload, etc. and perform specific scaling and reconfiguration actions. This has the following shortcomings. First, it is difficult to detect which, among the numerous, system parameters affect performance, as they are application and workload dependent. For example, in a specific write-heavy scenario the CPU usage may not be affected, as the bottleneck is mainly due to I/O operations and, thus, a CPU based rule will not work. Second, even if we can detect the parameters, it is difficult to determine which the respective actions should be. For example, even if we can conclude that for a write-heavy scenario we need a I/O rule, the appropriate action may not be obvious: for instance, increasing the RAM and cache size of existing servers to avoid I/O thrashing may be a better action than adding more servers. Third, it is difficult to detect the threshold values based on which actions need to be triggered. For example, a pair of thresholds on "high"/"low" CPU usage, a threshold on the number I/O ops or on memory usage are very application-specific and need a lot of fine tuning. For all three reasons, the translation of higher level business policies into a rule-based approach that automatically scales and reconfigures the system is difficult and error-prone.

Solution. We create a model of the computing environment by representing each selected parameter of system resources and workload with a distinct dimension. Therefore the environment is modelled by a multi-dimensional space in which all possible states of the environment can be represented, with variable detail. We create a novel MDP algorithm that starts with one or a few model states that cover the entire multi-dimensional state space. The algorithm gradually partitions the coarse state space into finer states depending on observed measurements of the modelled environment parameters. The algorithm employs a decision tree in order to perform this dynamic and adaptive partitioning. At each state the algorithm takes an action in order to make transition toward another state by optimizing a user defined reward function. Such actions may change the values of some of the parameters of the system resources, e.g. change of (i) the size of the cluster in terms of machines and (ii) the number of VMs. Nevertheless, even though the actions change only some parameters of the system resources, they may affect many more such parameters, as well as parameters of the workload. The type of actions allowed is given as an input to the algorithm.

Algorithm 1 MDP_DT Algorithm

```
1: m = collect_measurements()
2: while True do
3:     s = state(m); a = select_action(s); execute_action(a); sleep()
4:     m' = collect_measurements(); r = get_reward(m')
5:     e = (m, m', a, r); UpdateMDPModel(e); UpdateModelValues(s)
6:     ApplySplittingCriterion(s); m = m'
7:
8: procedure UPDATEMDPMODEL(e)
9:     m, m', a, r = e; s = state(m); s' = state(m')
10:    experiences(s, s').add(e); transitions(s, a, s')++;
11:    rewards(s, a, s') += r
12:
```

Motivating Example - continued: Employing our MDP approach the administrator of the startup company needs only to provide the following: a) a list of parameters she considers important to performance (even if some of them may turn out to not affect performance, at least for the observed states), b) a list of available scaling or reconfiguration actions, c) a high-level user-defined policy in the form of a reward function that encapsulates the maximization of performance and minimization of cost, and, optionally d) some initial knowledge in the form of a "training set" to speed up the learning process (in Section III we give a concrete example of parameters, actions and policies). Then, our algorithm adaptively detects both the set of parameters that affect the reward and the appropriate scaling and reconfiguration actions that maximize the reward.

Description of the algorithm. The *MDP_DT* algorithm is presented in Alg. 1 and Table I summarizes its terminology. *MDP_DT* starts with a single tree node (the root of the decision tree), which corresponds to one state covering the entire state space of the model of the environment. A vector *state* is maintained for all possible states of the environment. Each element s in *state* corresponds to a list of Q-states, holding the number of transitions *transitions* and the sum of rewards *rewards* towards each state s' in the model, along with the total number of times the action has been taken. The current state s is represented by a set of measurements m that contain the names and current values for all the parameters of the environment. The state s' to which action a leads is represented by a respective set of measurements m'. Given the current state s (and corresponding measurements m), the algorithm selects an action a, the action is performed, and the algorithm collects the measurements m' for the new state s' of the environment, for which it calculates the reward r. This transition experience $e = (m, a, m', r)$ is used to update the MDP model, the model values and split the state s, using procedures *UpdateMDPModel*, *UpdateMD-*

Algorithm 2 Parameter Test

```
1: procedure SplitParameterTest(s)
2:     a = optimal_action(s)
3:     es = {e | e ∈ experiences(s, ∗), e.a = a}
4:     e+ = {e | e ∈ es, q_value(e) ≥ value(a)}
5:     e− = {e | e ∈ es, q_value(e) < value(a)}
6:     lowest_error = 1
7:     for p in parameters do
8:         p− = {e.m[p] | e ∈ e−}, p+ = {e.m[p] | e ∈ e+}
9:         error_prob = stat_test(p−, p+)
10:        if error_prob < lowest_error then
11:            lowest_error = error_prob
12:            best_p = p, best_p− = p−, best_p+ = p+
13:    if lowest_error ≤ max_type_I_error then
14:        mean− = mean(best_p−), mean+ = mean(best_p+)
15:        split_point = (mean− + mean+)/2
16:        split(s, best_p, split_point)
```

Algorithm 3 Q Value Test

```
1: procedure SplitQValueTest(s)
2:     a = optimal_action(s), es = {e | e ∈ experiences(s, ∗), e.a = a}
3:     N = length(es), lowest_error = 1
4:     for p in parameters do
5:         sort_by_param(es, p)
6:         for i in 1..N − 1 do
7:             if es[i].m[p] = es[i + 1].m[p] then
8:                 continue
9:             q− = {q_value(e) | e ∈ es[1..i]}
10:            q+ = {q_value(e) | e ∈ es[i + 1..N]}
11:            error_prob = stat_test(q−, q+)
12:            if error_prob < lowest_error then
13:                lowest_error = error_prob, best_p = p
14:                split_point = (e.m[p] + e′.m[p])/2
15:    if lowest_error ≤ max_type_I_error then
16:        split(s, best_p, split_point)
17:
18: function QValue(e)
19:    m, m′, a, r = e
20:    s′ = state(m′), a′ = optimal_action(s′)
21:    return r + γ · value(a′)
```

PValues and *ApplySplittingCriterion*, respectively. Procedure *UpdateMDPModel* saves the experience $e = (m, m', a, r)$ in the $experiences$ vector in the place corresponding to the pair of s, s', increases the number of transitions for the pair of s, s' and adds the new reward r to the accumulated reward for the pair of s, s'. Procedure *UpdateMDPValues* updates the Q-state values for state s by employing one of the classical update algorithms: single update, value iteration, and prioritized sweeping. Procedure *ApplySplittingCriterion* considers splitting state s in two new states, based on a criterion. We propose two splitting criteria.

B. Splitting Criteria

The proposed criteria, *parameter test* and the *Q-value test*, have two strengths. First, the Q-value derived from each experience is calculated using the current, most accurate values of the states instead of the values at the time the action was performed. Second, the partitioning of experiences is done by comparing them to the current value of state s instead of partitioning them to experiences that increased or decreased the Q-value at the time of their execution. These features allow reliable re-use of experiences collected early in the training processing, at which point the values of the states were not yet known, throughout the lifetime and adaptation of the model.

Parameter test: Procedure *SplitParameterTest* presented in Alg. 2 implements the splitting criterion *parameter test* which works as follows. From the experiences $e = (m, a, m', r)$ stored in the *experiences* vector for every pair of s, s', we isolate the experiences where the action a was the optimal action for state s (i.e., a led to the highest Q-value). For each of these experiences, we find the state s' in the current model that corresponds to m' using the decision tree, and calculate the value $q(m, a) = r + \gamma V(s')$. We then partition this subset of experiences to two lists e_- and e_+ by comparing $q(m, a)$ with the current value of the optimal action for state s. For each parameter p we divide the values of p for the measurements in e_- and e_+ in two lists p_- and p_+, and run a statistical test on p_- and p_+ to determine the probability that the two samples come from the same population. We want to split the parameter with the lowest such probability, as long as it is lower than the error $max\_type\_I\_error$, else the procedure aborts. If the split proceeds, the splitting point is the average of the means of p_- and p_+.

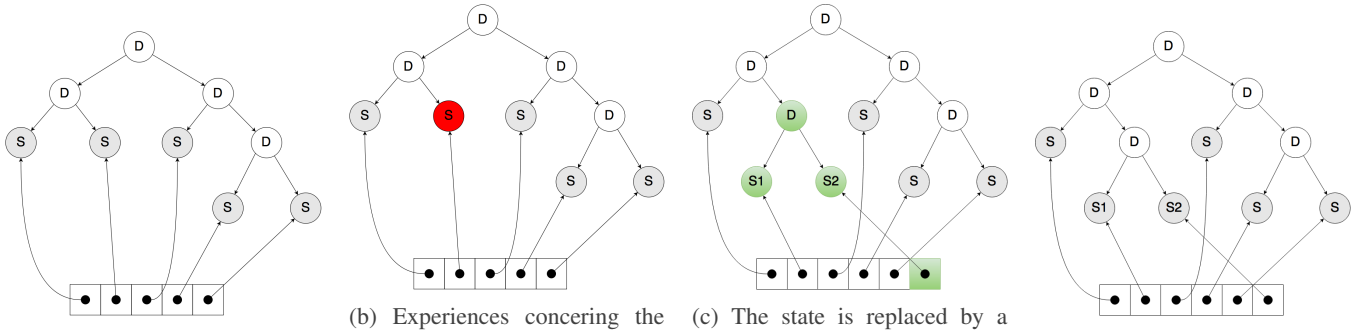
Algorithm 4 Splitting a State

```
1: procedure Split(s, param, point)
2:     transitions(∗, ∗, s) = 0, rewards(∗, ∗, s) = 0
3:     es = {e | e ∈ experiences(s, ∗) ∪ experiences(∗, s)}
4:     experiences(s, ∗) = experiences(∗, s) = []
5:     replace_with_decision_node(s, param, point)
6:     for e in es do
7:         UpdateMDPModel(e)
8:     UpdateModelValues(s)
```

Q-value test: Procedure *SplitQValueTest* presented in Alg.3 implements the splitting criterion *Q-value test* which works as follows. Again, from the experiences $e = (m, a, m', r)$ related to pairs of s, s', we isolate the experiences where the action a is the current optimal action for s. For each such experience, we find the state s' that corresponds to m' using the current decision tree and calculate $q(m, a) = r + \gamma V(s')$. For each parameter p of the system, we sort these experiences based on the value of p, and consider splitting in the midpoint between each two consecutive unequal values. For that purpose, we run a statistical test on the Q-values in the two resulting sets of experiences, and choose the splitting point that produces the lowest probability that represents the fact the two sets of values are statistically indifferent, as long as that probability is less than the error $max\_type\_I\_error$. This criterion performs a straightforward comparison of subsets of experiences with respect to the optimality of the taken action. It is a criterion *that we adapted from the Continuous U Tree algorithm [24], and* resembles splitting criteria used in traditional algorithms for decision tree induction such as C4.5 [18]. Additionally, we experimented with splitting in the midpoint between the two unequal consecutive measurements that are closest to the median. This considers a single splitting point per parameter, which splits the experiences approximately equally.

C. Performing Splits

Once a split has been decided by the *parameter test* or the *Q-value test* for a state s, the splitting is performed by procedure *Split* shown in Alg. 4. Figure 1 showcases this procedure. For state s, all transition and reward information is removed from the $transitions$ and $rewards$ vectors, respectively. Also, the experiences e that involve s as the starting or the ending state are accumulated and stored in temporary storage es.



(a) The decision tree before splitting a state

(b) Experiences concering the state to be split are transferred to temporary storage, respective transition and reward information is removed.

(c) The state is replaced by a new decision node and two children corresponding to the two new states.

(d) Experiences in temporary storage are used to train the new states, and calculate the new transitions and rewards.

Fig. 1: Splitting a state s into two states s_1, s_2: s_1 replaces s in the $state$ vector and the s_2 is appended at the end.

These experiences are removed from the $experiences$ vector, because state s will be substituted by two new states s_1, s_2. This is performed in two steps: First, the node in the decision tree that corresponds to state s is replaced with a new decision node, and two children nodes, (leafs), that correspond to the two new states that result from splitting s. Second, one of the new states takes the place of the split state s in the $state$ vector and the second new state takes a new position appended at the end of the $state$ vector. The leaf nodes are linked to the positions in the $state$ vector of the corresponding states. The $reward$, $transition$ and $experiences$ vectors are updated and extended accordingly, with new elements for combinations of s_1, s_2 with all other states. The obsolete experiences es are used to retrain the new states: For each $e = (m, a, m', r)$, the new states s and s' are found using the updated decision tree, and the respective positions in the $reward$ and $transition$ vectors are updated.

D. Statistical Tests

The splitting criteria include a statistical test to determine whether the two groups of compared values are statistically different from each other. For this, we employ four different statistical tests. The statistic formulas can be found in [11].

Student's t-test: The equal variance *t*-test, widely known as *Student's t-test*, estimates the probability that the two compared samples have a different mean, under the assumption that they share the same variance.

Welch's test: The unequal variance *t*-test, also known as *Welch's test*, is an alternative to the Student's *t*-test that also tests whether the population means are different, but without assuming that they share the same variance.

Mann Whitney U test: This test is also an alternative to the *t*-test that does not require the assumption that the two populations follow a normal distribution, and is used on both discrete and continuous data. It calculates a U statistic, with known distribution under the null hypothesis (for sizes above 20 we assume a normal distribution).

Kolmogorov-Smirnov test: The two sample Kolmogorov Smirnov test can be used to test whether two underlying one-dimensional probability distributions differ, without assuming normality for the two distributions.

E. Splitting Strategy

By default, the *MDP_DT* algorithm attempts to split the starting state of each experience after this has been acquired, *and* depending only on this. However, the effectiveness of the algorithm may be better if the splitting is performed after the acquisition of more than one experiences *and/or* independently of these specific experiences. We investigate this with three basic *splitting strategies*:

- **Chain Split**: The *Chain Split* strategy aims at accelerating the division of the state space into finer states, by accelerating the growth of the decision tree. It attempts to split every node of the tree, regardless of whether it was involved in the current experience. The rationale is that the change in the value of the current state may affect also the value of other states, and, therefore, it should trigger the splitting not only of the current state, but also of others.
- **Reset Split**: The *Reset Split* strategy aims at correcting splitting mistakes, by resetting the decision tree periodically, and by taking more accurate decisions after each reset, considering all accumulated experiences.
- **Two-phase Split:** The *Two-phase Split* strategy splits periodically based on accumulated experiences. Therefore, in this case the *MDP_DT* algorithm is versioned so that it has two phases, a *Data Gathering* phase that collects data but does not perform any splits, and a *Processing Phase* that the tree nodes are tested one by one to check if a split is needed, and if so, perform the splits.

III. Simulation Results

In this section we show experimental results on a simulation of the elastic computing environment. The agent makes elasticity decisions that resize a cluster running a database under a varying incoming load. The load consists of read and write requests, and the cluster capacity (i.e., the maximum achievable query throughput) depends on its size as well as the percentage of the incoming requests that are reads. Specifically:

- The cluster size can vary between 1 and 20 VMs.
- The available actions are: the increase cluster size by one, decrease the cluster size by one, or do nothing.
- The incoming load is a sinusoidal function of time: $load(t) = 50 + 50sin\left(\frac{2\pi t}{250}\right)$.

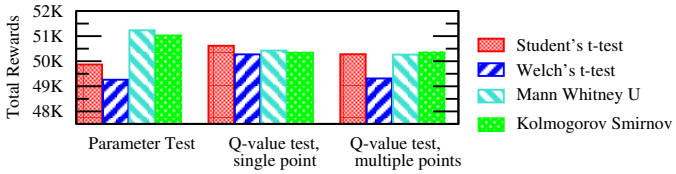


Fig. 2: Performance comparison of all the splitting criteria using their optimal settings (400 runs)

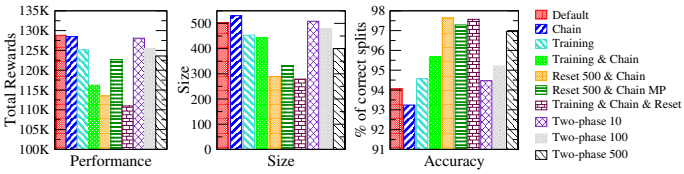


Fig. 3: Strategies performance, size and accuracy, (200 runs)

- The percentage of incoming read requests is a sinusoidal function of time with a different period: $r(t) = 0.75 + 0.25sin\left(\frac{2\pi t}{340}\right)$.
- If $vms(t)$ is the number of VMs currently, the cluster capacity is: $capacity(t) = 10 \cdot vms(t) \cdot r(t)$.
- The reward for each action depends on the state of the cluster after executing the action and is given by $R_t = min(capacity(t+1), load(t+1)) - 3 \cdot vms(t+1)$.

The reward function encourages the agent to increase the size of the cluster to the point where it can fully serve the incoming load, but punishes it for going further than that. In order to test the algorithm's ability to partition the state space in a meaningful manner, apart from the three relevant parameters (size of the cluster, incoming load and percentage of reads) the dimensions of the model include seven additional parameters, the values of which vary in a random manner. Four of them follow a uniform distribution within $[0, 1]$, while the rest take integer values within $[0, 9]$ with equal probability. To be successful the algorithm needs to partition the state space using the three relevant parameters and ignore the rest.

All experiments include a training phase and an evaluation phase. During the training phase, the selected action at each step is a random action with probability e, or the optimal action with probability $1 - e$ (e-greedy strategy). During the evaluation phase only optimal actions are selected, as proposed by the algorithm. The metric according to which different options are compared is the sum of rewards that the agent managed to accumulate during the evaluation phase.

A. Splitting Criteria

In this experiment we compare the performance of the splitting criteria using a $max\_type\_I\_error$ value of 0.002 for which all the splitting criteria behaved optimally (more details can be found in [11]).In Fig. 2 the two tests that do not assume a normal distribution of the values, the *Kolmogorov-Smirnov test* and the *Mann Whitney U test*, perform better in the *Parameter test* criterion, while the two tests that assume normal distributions perform better employed in the *Q-value test*. The reason is that some of the parameters are discrete; this means that their distribution differs significantly from a normal distribution, resulting in lower performance of the tests if they are applied on the values of these parameters. Oppositely, these two tests perform better employed in the *Q-value test* criterion, since the Q-values are generally not discrete.

Among all options, the *Mann Whitney U test* achieved the best performance, employed in the *Parameter test*. Finally, between the two available options for the *Q-value test*, namely considering a single or multiple splitting points, the consideration of a single splitting point achieves generally better results, while it produces smaller trees.

TABLE II: Splitting Strategies

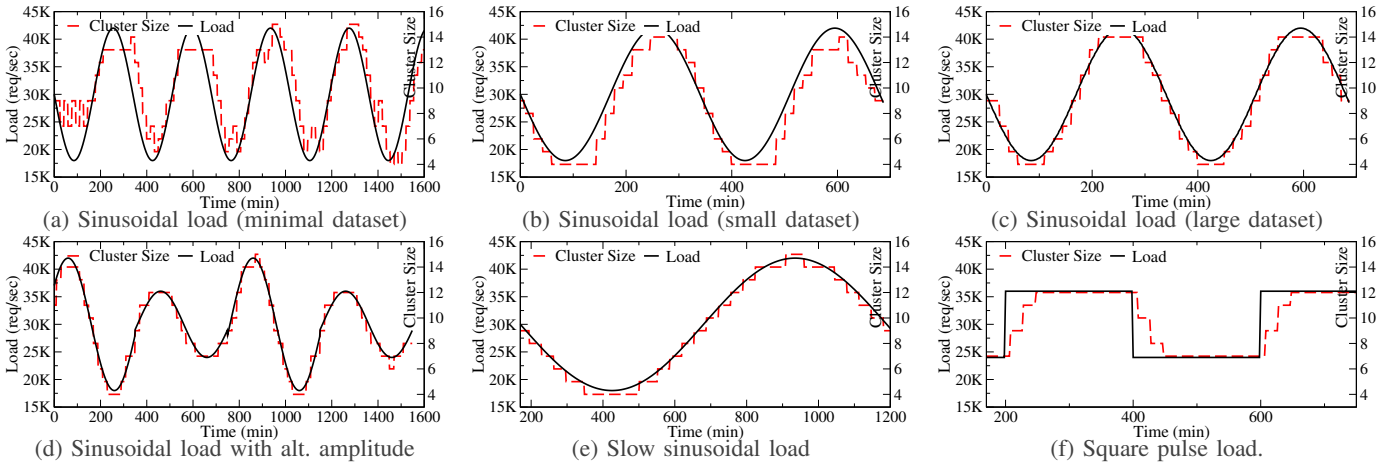
Name	Description
Default	Attempt to split the starting state for each new experience.
Chain	Perform a *Chain Split* with every experience.
Training	Allow splitting to begin after a training of 2500 steps and then start splitting with every new experience.
Training & Chain	Allow splitting to begin after a training of 5000 steps, perform one chain split at that time; then continue splitting with every experience.
Reset 500 & Chain	At each step perform a *Chain Split* and every 500 steps reset the decision tree.
Reset 500 & Chain MP	As above, but using the multiple points *Q-value test* criterion, attempting to split each state at multiple points per parameter.
Training & Chain & Reset	After a training of 5000 steps, perform a *Chain Split* and then do the same by resetting the tree every 500 steps.
Two-phase 10	After a *Data Gathering* phase of 10 steps, run a *Processing* phase.
Two-phase 100	After a *Data Gathering* phase of 100 steps, run a *Processing* phase.
Two-phase 500	After a *Data Gathering* phase of 500 steps, run a *Processing* phase.

B. Splitting Strategy

We compare various different splitting strategies by mixing the three basic strategies defined in Section II-E, and in Table II we present the created variety of strategies.

In Figure 3 we present the performance in terms of total rewards, the size of the produced decision tree and the accuracy for every different strategy. Even though *Chain Split* adopts a much more aggressive (and computationally intensive) strategy in attempting to grow the decision tree, it has a performance similar to the *Default* strategy. Also, even though *Chain Split* performs 30 additional splits on average, the split quality decreases. The relatively low amount of additional splits reveals that the default strategy already depletes most of the opportunities to create new states. Waiting for more data to be available in order to start splitting did not perform well. Despite offering a slight increase in the accuracy of the splits, it causes a 10% reduction in their number and in the case of strategy (iv) a drop in performance. Periodically resetting the tree to rebuild it provides the most accurate splits on the final tree, since the splits are performed with the maximum amount of data. However, the resulting tree size is significantly smaller.

The results for strategy (vii) show the impact of having a long training before starting splitting, then doing a chain split, and resetting the decision tree every 500 steps thereafter. During the long training the optimal action in each region of the state space is repeated only a few times due to the e-greedy strategy. Less data is available in order to perform splits, which results in a smaller tree, with consequences in performance. Finally, using a *Data Gathering* and a *Processing* phase periodically instead of regularly splitting performed better the smaller that period was. If *Processing* is performed every 10 steps it nearly reached the performance of the default strategy (though having a significantly larger running time), but for periods larger than that it falls behind. Overall, the results of this experiment show that the default splitting method is



(a) Sinusoidal load (minimal dataset)
(b) Sinusoidal load (small dataset)
(c) Sinusoidal load (large dataset)
(d) Sinusoidal load with alt. amplitude
(e) Slow sinusoidal load
(f) Square pulse load.

Fig. 4: *MDP_DT* under different training data (Figs 4a, 4b and 4c) and workload types (Figs 4d, 4e and 4f).

efficient and effective.

IV. Experimental Results

In this section we evaluate our findings on a real experimental setup. In subsection IV-B we evaluate the performance of *MDP_DT* using different training data sizes and workloads, in subsection IV-C we compare *MDP_DT* with model free, static partitioning and threshold algorithms and finally in IV-D we showcase *MDP_DT*'s adaptation ability to benefit from multiple parameters and perform correct decisions.

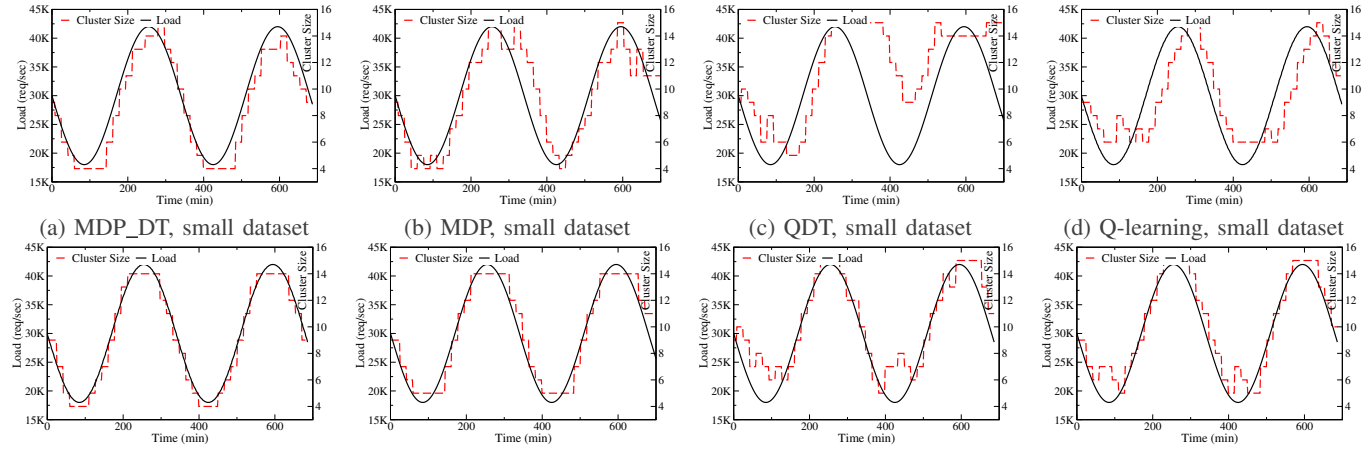
A. System and Algorithm Setup

In order to test our proposal in a real cloud environment, we use an *HBase 1.1.2* NoSQL distributed database cluster running on top of *Hadoop 2.5.2*. We generate a mix of different read and write intensive workloads of varying amplitude by utilizing the *YCSB* [7] benchmark, while *Ganglia* [12] is used for the collection of the NoSQL cluster metrics. The cluster runs on a private *OpenStack* IaaS cloud setup. The coordination of the cluster is performed by a modified version of [23]. In any case, we allow for 5 different actions, which include adding or removing 1 or 2 VMs from the cluster, or doing nothing. Decisions are taken every 15 minutes, as it is found that this time is necessary for the system to reach a steady state after every reconfiguration. This fact also confirms our full-model choice, since this time is adequate to calculate the updated model after every decision. Every workload runs for approximately 10 hours during which around 20M queries are being sent. The cluster size used in our experiments ranges between 4 and 15 VMs. Each VM in the HBase cluster has 1GB of RAM, 10GB of storage space and 1 virtual CPU, while the master node has 4GB of RAM, 10GB of storage and 4 virtual CPU's. For the training of the decision tree based models we use a set of 12 parameters including: The cluster size, the amount of RAM per VM, the percentage of free RAM, the number of virtual CPU's per VM, the CPU utilization, the storage capacity per VM, the number of I/O requests per second, the CPU time spent waiting for I/O operations, a linear prediction of the next incoming load, the percentage of read requests in the queries, the average latency of the queries and the network utilization.

For the Decision Tree based algorithms we split the state space using the Mann Whitney U test with the Parameter Test splitting criterion over a default splitting strategy, following our findings in sections III-A and III-B. The model-based algorithms update their optimal policies utilizing the *Prioritized Sweeping Algorithm* [14]. For the static partitioning algorithms we select two dimensions that were found to be the most relevant for the cluster performance (i.e., the cluster size and the linear load prediction) divided in 12 and 8 equal partitions respectively, resulting in 96 states. We note that this setup is optimal for the static schemes, as they require a small number of relevant states to behave correctly. The reward function is $R_t = served\_requests\_per\_sec(t+1) - 800 \cdot vms(t+1)$ and encourages the agent to increase the cluster size to serve the load, but punishes it for going further.

B. MDP_DT Behavior

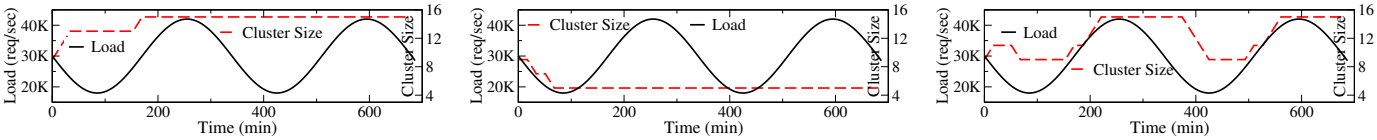
In this section we test *MDP_DT* using different workloads and training set sizes. In Figure 4 we present our findings. Every experiment runs for a total of 700-1200 minutes (X axis). The solid line represents the workload in terms of Reqs/sec (left Y axis) whereas the dotted line represents the cluster size (right Y axis). Every step in the dotted line represents an *MDP_DT* action of adding or removing VMs. We initialize the *MDP_DT* decision tree with 6 states and let it partition the state space on its own from that point on. The training load is a sinusoidal load of varying amplitude. First, we run *MDP_DT* with a minimal dataset of 500 experiences (Figure 4a). When trained with this dataset, only 17 splits are performed during the training (4 using the size of the cluster and 13 using the incoming load), increasing the total number of states to 22. During this run 12 additional splits are performed (4 using cluster size, 7 using incoming load and 1 using latency), allowing *MDP_DT* to continuously adapt and follow the incoming load (Fig. 4a). When provided with bigger datasets of 1000 and 20000 experiences, the performance improves and very closely converges to the incoming load,



(a) MDP_DT, small dataset (b) MDP, small dataset (c) QDT, small dataset (d) Q-learning, small dataset

(e) MDP_DT, large dataset (f) MDP, large dataset (g) QDT, large dataset (h) Q-learning, large dataset

Fig. 5: Comparison of the behavior of model-based vs model-free and decision-tree based vs static algorithms.



(a) REMOVE at 30%, ADD at 60% (b) REMOVE at 70%, ADD at 90% (c) REMOVE at 50%, ADD at 70%

Fig. 6: Threshold-based approach behavior for different ADD and REMOVE thresholds on the aggregated %CPU metric

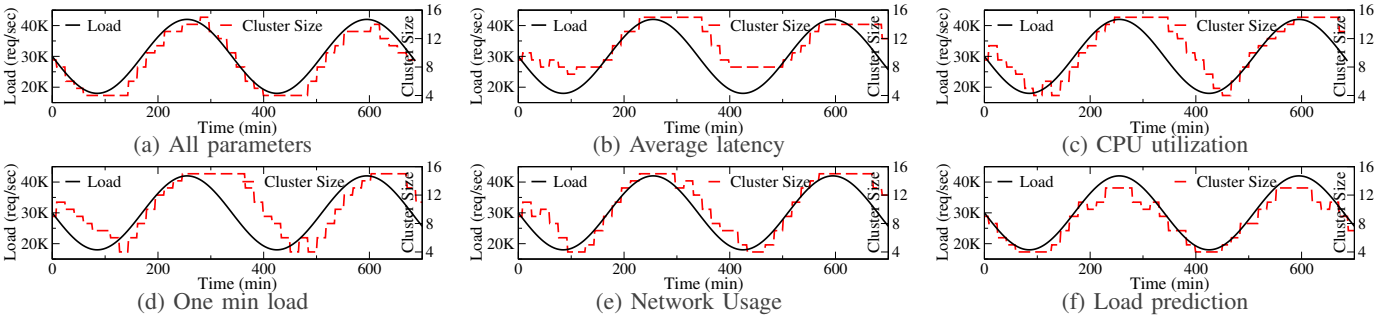
ending up with 54 and 576 states respectively (Fig. 4b and 4c). Finally, in Fig. 4d, 4e and 4f we observe how *MDP_DT* adapts to arbitrary workloads that has not previously encountered.

C. Using Different Algorithms

In this section we test the system's behavior compared to other RL and threshold based schemes. RL and threshold schemes are presented in Figures 5 and 6 respectively whereas Table III summarizes results regarding every solution's aggregated cost and profit. RL approaches consist of model-free (i.e., *Q-learning*) and static partitioning schemes. The combinations of these schemes lead to four different algorithms, namely the model-based adaptively partitioned *MDP_DT*, the model-based statically partitioned *MDP*, and the respective model-free versions (*Q_DT* and *Q-learning*). Regarding RL (Figure 5), we notice that *MDP_DT* follows the applied workload very closely (Figures 5a and 5e), with a better result when a large training set is utilized (Figure 5e). Indeed the algorithm's decisions over time depicted in the red dotted line seem to perfectly adapt to the observed workload, since both lines almost overlap. The full-model based *MDP* algorithm also performs well in this setting (Figure 5f), managing to follow the incoming load reasonably well even when trained with the small dataset (Figure 5b). At the same time, it manages to perform accurately when trained with more data. This is not a surprise since this problem has a reasonably simple state space, and a partitioning using only the size of the cluster and the incoming load is quite sufficient to capture the behavior in this experiment: in this ideal setting the state space is very accurately defined and also a lot of training data is available. Yet, in sudden load spikes observed in the max and min load values it takes more time to respond compared to the *MDP_DT* case.

The *Q-learning* based algorithms though both require a large amount of data to follow the incoming load effectively (Figures 5c, 5d, 5g and 5h). In this experiment the decision tree based *Q-learning* algorithm (*QDT*) achieves the weakest performance with the small dataset (Figure 5c). With this few data this is not totally unexpected, since at the start of the training that model uses the first data it acquires to perform splits, but then discards it after the splits have been performed, leaving it with very little available information to make decisions. If more training data is provided though, it catches up to the traditional *Q-learning* model (Figure 5g). However, they both are noticeably less stable (they cannot follow the observed load in an adequate manner) compared to the full model approaches.

In the threshold-based scheme (Figure 6) we monitor the aggregated cluster %CPU usage and we add two different threshold rules, namely ADD and REMOVE, that trigger a cluster expansion or contraction when the monitored metric exceeds or drops the predefined threshold values respectively. The %CPU metric was selected as it was found to be the most representative of the cluster's performance. We have set an upper and lower limit on the total cluster size to avoid mis-calibration issues. In Figure 6a we notice that the cluster constantly increases its size to the maximum possible, without being able to follow the applied load: this is due to a combination of a low ADD threshold that is constantly triggered even in an underloaded cluster and a low REMOVE threshold that is never crossed since the cluster's CPU usage is constantly above this number. In Figure 6b the system has the exact opposite behavior and the cluster size drops to the minimum possible without adapting to the applied load: in that case, the REMOVE threshold is constantly triggered



(a) All parameters (b) Average latency (c) CPU utilization

(d) One min load (e) Network Usage (f) Load prediction

Fig. 7: System behavior when allowing splits with only the cluster size plus one additional parameter

shrinking the cluster to its minimum size. On the other hand, a very high ADD threshold is never met, as even in overloaded cases the %CPU never crosses a maximum number of 80%-85% since HBase does not consume the entire CPU, leaving room for other OS-related tasks. Nevertheless, this boundary is system and application specific. However, a small cluster size cannot accommodate the applied load (Table III). After a lot of fine-tuning, in Figure 6c we managed to find the most appropriate ADD and REMOVE parameters that adequately follow the applied load. In any case, we observe that even after a very tedious and error-prone calibration task that requires system specific knowledge, *MDP_DT* (Figure 5e) managed to outperform the threshold based approach by adaptively identifying both the relevant parameters and the appropriate scaling action timing.

In Table III we measure and compare the achieved profit (second column) and cost in terms of cloud infrastructure renting (third column) of every algorithm. The profit is calculated as a percentage of the total queries actually served to the total of 20M received queries (assuming a small charging fee for answering every query, a typical business model followed in cloud hosting services, this is proportional to the actual financial profit). The infrastructure cost is calculated by summing the amount of acquired hardware resources (i.e., cluster size) in every time unit, which is proportional to the areas under the dashed cluster-size lines in Figures 5 and 6. We report the percentage of every algorithm's cost compared to the *MDP_DT*'s cost. We notice that *MDP_DT* (first two lines) is constantly cheaper with values ranging from 10%-40% for RL based schemes and 40%- 65% for threshold schemes while being able to acquire almost all the available profit, losing only 2% of the applied queries. The most expensive RL scheme is *QDT small* due to its constant resource over-provision (Figure 5d) and the next cheaper to *MDP_DT* scheme is *Q-learning small* caused by a resource under-provisioning that results in 7% profit loss (Figure 5d). Regarding the threshold-based schemes, the 30-60 scheme is 60% more expensive, as it constantly utilizes the maximum resources (Figure 6a), whereas even in the calibrated 50-70 scheme the extra cost is more than 40% (Figure 6c). Finally, the 70-90 scheme is 40% cheaper than *MDP_DT*, nevertheless this comes at a price of losing 26% of the available profit due to resource under-provisioning. In any case, *MDP_DT* is the only algorithm that combines a low cost and a high profit.

TABLE III: Profit & Cost of RL & Threshold Algorithms

Algorithm	**% Achieved Profit**	**% Cost vs *MDP_DT***
MDP_DT, small	98	100
MDP_DT large	98	106
MDP small	97.5	112.2
MDP large	98	114.2
QDT small	97.5	138
QDT large	98	118
Q-learning small	93	108
Q-learning large	99	116
Threshold 30 60	99	163.7
Threshold 70 90	74	60.5
Threshold 50 70	99	140

D. Restricting the Splitting Parameters

In order to test the algorithm's ability to partition the state space using different parameters, as well as to test the reliability of some of the parameters in predicting the incoming load, we experiment with restricting the parameters with which the algorithm is allowed to partition the state space. For that purpose, we experiment with training the algorithm from a small dataset of 1500 experiences, but restricting the parameters with which the algorithm is allowed to partition the state space to only the size of the cluster plus one additional parameter each time. The parameters used are the CPU util., the one minute averaged system load, the incoming load prediction, the network usage and the average latency.

In Figure 7 we present our findings. For all the parameters, the system seems to be able to find a correlation between the given parameter and the rewards obtained, and starts following the incoming load. Of course, the performance is significantly worse compared to the default case where all the available information is provided (Figure 7a), and thus the training of the model is noticeably slower: In Figures 7b, 7c, 7d, 7e and 7f decisions (dotted lines) do not follow workload as smooth as in Figure 7a. The fact that these correlations exist and can be detected even from a small dataset of only 1500 points, reveals the fact that it is possible, using techniques like the ones described in this work, to exploit these correlations in order to implement policies in systems with complicated behavior.

V. Related Work

The proposed approach is related to RL and adaptive resource management. We compare with methods that adaptively partition the state space and manage cloud system resources.

Adaptive State Space Partitioning: In [6], the authors propose a modification of *Q-learning* that uses a decision tree to generalize over the input. The agent goal is to control a character in a 2D video game, where the state is a bit string

representing the pixels of the on-screen game representation. While this approach succeeded at reducing a large state space to a manageable number of states, its applications are limited since it requires that the system parameters can only take two values, 0 and 1. In [17], a *Q-learning* algorithm that uses a decision tree to dynamically partition the state space is proposed. The algorithm discards training information each time a state is split and it is not based on a full model, which results in it requiring a larger set of experiences to train. In [13], a full model based algorithm using a decision tree to partition the state space is proposed. It is called *U Tree*, and it is able to work strictly on discrete state spaces. An extension is proposed in [24], called *Continuous U Tree*. The splitting criterion was a combination of the Kolmogorov-Smirnov and the *Q-value* tests on multiple points, which was outperformed by the Mann Whitney U test with the Parameter test criteria.

Adaptive Resource Management: Elastic resource scaling is typically employed in a cloud setting to regulate resource size and type according to observed workload. Amazon's autoscaling service [1] employs simple threshold based rules, whereas Google, RackSpace, etc., offer similar services. CloudScale [20] employs thresholds to meet user defined SLAs, while it focuses on accurate predictions. Lim et al [10] set thresholds to aggregated CPU usage and response time to regulate the HDFS cluster size. ElasTraS [8], SCADS [22] and AGILE [15] employ rule based approaches for scaling and reconfigurations. Although rule-based approaches are easy to implement and model, they require specific lower level knowledge of the correct parameters and the respective threshold values, limiting their broader applicability.

On the contrary, systems that employ RL or similar approaches allow the user to set higher level policies, like, minimize cost and maximize query throughput [19], [23]. In [19] a model-based RL algorithm to automatically select optimal configuration settings for clusters of VMs is proposed. However, static partitioning limits its applicability to 10 dimensions. In [5], RL finds optimal configuration settings for online web systems. High dimensionality is handled with parameter grouping according to their similarity, yet partitioning is static and uniform. An RL approach that splits the system parameters into two groups, namely application and cluster configuration parameters, is adopted by the authors of [4]. *Simplex-based space reduction* is implemented to further narrow down the state space, but such techniques can easily be trapped in local minima and offer no guarantees of convergence to an optimal configuration. In TIRAMOLA [9], [23], the MDP model uses a fixed state model both in terms of parameter size and grain. In [2] Q-learning is employed, and its execution is parallelized and used to regulate a VM cluster size. PerfEnforce [16] offers three scaling strategies to guarantee SLAs for analytical cloud workloads and one of them is based on RL [23]. They only employ RL and do not perform any algorithmic improvements.

VI. Conclusions

In this paper we presented *MDP_DT*, an RL algorithm that adaptively partitions the state space utilizing novel statistical criteria and strategies to perform accurate splits without losing already collected experiences. We calibrated the algorithm's parameters utilizing a simulation environment and we experimentally evaluated *MDP_DT*'s performance in a real cluster deployment where we elastically scaled a shared-nothing NoSQL database cluster. *MDP_DT* was able to identify and create only the relevant partitions among tens of parameters, enabling it to take accurate and fast decisions during the NoSQL scaling process over complex not-encountered workloads with minimal initial knowledge, compared to model-free, static and threshold algorithms. This adaptation allowed *MDP_DT* to optimize the achieved profit while being 40% cheaper than calibrated RL and threshold schemes.

Acknowledgment

This paper is supported by European Union's Horizon 2020 RIA programme under GA No 690588, project SELIS.

References

[1] AWS | Auto Scaling. http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html.
[2] E. Barrett, E. Howley, and J. Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.
[3] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ., 1957.
[4] X. Bu, J. Rao, and C. Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE TPDS*, 24(4):681–690, 2013.
[5] X. Bu, J. Rao, and C.-Z. Xu. A Reinforcement Learning Approach to Online Web Systems Auto-configuration. In *ICDCS*, pages 2–11, 2009.
[6] D. Chapman and L. P. Kaelbling. Input Generalization in Delayed Reinforcement Learning: An Algorithm and Performance Comparisons. In *IJCAI*, volume 91, pages 726–731, 1991.
[7] B. F. Cooper et al. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.
[8] S. Das et al. ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud. *TODS*, 38(1):5, 2013.
[9] E. Kassela et al. Automated workload-aware elasticity of nosql clusters in the cloud. In *BigData*, pages 195–200. IEEE, 2014.
[10] H. C. Lim, S. Babu, and J. S. Chase. Automated Control for Elastic Storage. In *ICAC*, pages 1–10. ACM, 2010.
[11] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Elastic Resource Management with Adaptive State Space Partitioning of Markov Decision Processes. *CoRR*, abs/1702.02978, 2017.
[12] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.
[13] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, 1996.
[14] A. W. Moore and C. G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, 1993.
[15] H. Nguyen et al. Agile: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *ICAC*, pages 69–82, 2013.
[16] J. Ortiz, B. Lee, and M. Balazinska. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *SIGMOD*, 2016.
[17] L. D. Pyeatt et al. Decision Tree Function Approximation in Reinforcement Learning. In *Symposium on adaptive systems: evolutionary computation and probabilistic graphical models*, volume 1, 2001.
[18] J. R. Quinlan. C4. 5: Programming for machine learning. *Morgan Kauffmann*, 1993.
[19] J. Rao et al. VCONF: a Reinforcement Learning Approach to Virtual Machines Auto-configuration. In *ICAC*, pages 137–146. ACM, 2009.
[20] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *SoCC*, page 5. ACM, 2011.
[21] C. Tang et al. Holistic Configuration Management at Facebook. In *SOSP*, pages 328–343. ACM, 2015.
[22] B. Trushkowsky et al. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *FAST*, pages 163–176, 2011.
[23] D. Tsoumakos et al. Automated, Elastic Resource Provisioning for NoSQL Clusters Using Tiramola. In *CCGRID*, pages 34–41, 2013.
[24] W. T. Uther and M. M. Veloso. Tree Based Discretization for Continuous State Space Reinforcement Learning. In *Aaai*, pages 769–774, 1998.
[25] A. Verma et al. Large-scale Cluster Management at Google with Borg. In *EuroSys*, page 18. ACM, 2015.