2017 IEEE 33rd International Conference on Data Engineering

Adaptive State Space Partitioning of Markov Decision Processes for Elastic Resource Management

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 there 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 configure and optimize. Approaches based on reinforcement learning 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 and model-free approaches, while requiring a minimal amount of training data.**

I. Introduction

Modern large-scale computing environments, like large private clusters, cloud providers and data centers may have deployed tenths of platforms, like NoSQL and database servers, web servers, etc. on thousands of machines, and run on them hundreds of services [1]. 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 is challenging. 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 configuration management system [2], whereas Google's Borg [1] manages 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.

Public cloud providers such as Amazon, Google, Microsoft and IBM offer autoscaling services [3]. 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 also explore threshold-based solutions [4], [5]. More sophisticated approaches employ *Reinforcement Learning* (RL) algorithms such as *Markov Decision Processes* (MDP) and *Q-Learning* which are natural solutions for decision making problems and offer optimality guarantees under conditions, yet they suffer from limitations that derive from the assumption of a priori parameter knowledge 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 [6]–[9].

In this work we employ 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 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. Our 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 state space size based on the amount of information on the system behavior. It starts with one or a few states and as more data are acquired, the number of states and the model accuracy dynamically increases. **Splitting criteria and strategies:** The algorithm

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

2375-026X/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDE.2017.72

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*. 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.** If an old state is replaced with two new ones, the old information is re-used to train the new ones.

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 accordingly.
- Three split strategies, *Chain, Reset and Two-phase Split*.
- An experimental study on a large-scale cluster which proves the effectiveness of *MDP_DT* in taking optimal resource management decisions fast, while taking into account tenths of parameters, and the superiority of the algorithm versus full model-based and model-free-based algorithms.

A detailed technical report can be found here [10].

II. The MDP_DT Algorithm

We outline the basic notations regarding Markov Decision Processes (MDPs) and we discuss preliminary knowledge in [10]. 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, or using alternative algorithms such as *Prioritized Sweeping* [11] to decrease the required calculation in each step. A full-model approach requires a smal training set in order to converge to an optimal policy, which makes it a preferable choice.

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 RAM or vCPUs or storage per VM, network characteristics, etc. **Workload parameters:** 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 the workload is executed efficiently and resources are not over-provisioned, without having prior knowledge of the characteristics, role and interaction of the parameters of both system resources and workload.

Motivating Example: A start-up company hosts services in a public IaaS cloud and 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/profit while minimizing the cost of operating the infrastructure" under variable and unpredictable loads. E.g., if the company offers user-facing low-latency services, the performance/profit can be measured based on throughput, and the cost can be the cost of renting the underlying IaaS services. 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 or the RAM size) or (ii) re-configure the system (e.g., increase cache size or change replication factor). A rule-based technique [3] that performs specific scaling and reconfiguration actions has the following shortcomings. First, it is difficult to detect which, among the numerous, parameters affect performance, as they are application and workload dependent. E.g., in a write-heavy scenario CPU usage may not be affected and a CPU based rule will not work. Second, even if we know the parameters, it is difficult to detect the respective actions. Even if we conclude that for a write-heavy scenario we need an 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 that will trigger actions. E.g, 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. Therefore, the translation of higher level business policies for the maximization of performance and minimization of cost into a rule-based approach that automatically scales and reconfigures the system is very 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 modeled 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 and gradually partitions the coarse state space into finer states depending on observed measurements by employing a decision tree 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 cluster size. Even though the actions change only some system parameters, the entire system behavior may be affected. The type of actions allowed is given as an input to the algorithm.

Motivating Example - continued: With 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 irrelevant), b) a list of scaling or reconfiguration actions, c) a high-level user-defined policy in the form of a reward function that encapsulates for instance 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. Then, *MDP_DT* algorithm adaptively detects both the set of parameters that affect the reward and the appropriate scaling and reconfiguration actions that maximize the reward.

Algorithm Description. 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

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:

maintained for all possible environment states. 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. 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', 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*, *UpdateMDPValues* and *ApplySplittingCriterion*, respectively [10]. 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 single update, value iteration or prioritized sweeping [11]. *ApplySplittingCriterion* considers splitting state s in two new states, based on a criterion. We propose two splitting criteria, the *parameter test* and the *Q-value test* (their pseudo-code is given in [10]). The proposed criteria 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.

The splitting criterion *parameter test* 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 choose to split the parameter with the lowest such probability, as long as it is lower than the error $max\_type\_I\_error$, else we abort. If the split proceeds, the splitting point is the average of the means of p_- and p_+.

The splitting criterion *Q-value test* 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 a $max\_type\_I\_error$. It performs a comparison of subsets of experiences w.r.t. the optimality of the taken action. It is a criterion *that we adapted from the Continuous U Tree algorithm [12], and* resembles splitting criteria used in algorithms for decision tree induction such as C4.5.

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

Once a split has been decided for a state s, procedure *Split* [10] creates two new states. It uses the obsolete state information to retrain the new states. The splitting criteria include a statistical test to determine whether the two groups of compared values are statistically different. For this, we employ four different statistical tests, namely *Student's t-test*, *Welch's test*, *Mann Whitney U test* and *Kolmogorov-Smirnov* test [10].

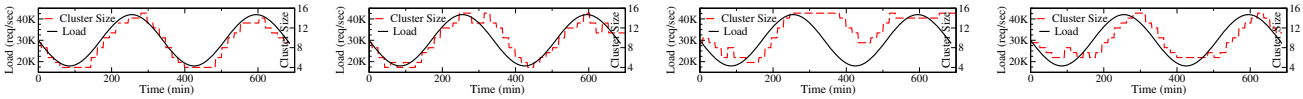
By default, the *MDP_DT* algorithm attempts to split the starting state of each experience after this has been acquired, *and* depending only on this. However, its effectiveness may be better if the splitting is performed after the acquisition of more than one experiences *and/or* independently of these specific experiences. We investigate this with three *splitting strategies*: **Chain Split** aims at accelerating the division of the state space into finer states. It tries to split every tree node, regardless of whether it was involved in the current experience. **Reset Split** aims at correcting splitting mistakes, by resetting the decision tree periodically, and by taking more accurate decisions after each reset, by taking into account all accumulated experiences. **Two-phase Split** splits the existing decision tree periodically. In this case the *MDP_DT* 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.

In [10] we found that the optimal results are achieved with the Parameter Test splitting criterion using the Man Whitney U statistical test and employing the default splitting strategy.

III. Experimental Results

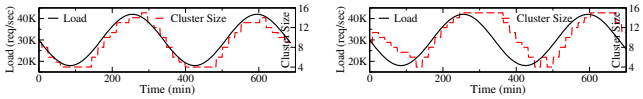
We employ our RL techniques in order to dynamically scale a real distributed database cluster deployed in a cloud environment under varying workloads. We evaluate the performance of *MDP_DT* compared with model free and static partitioning algorithms and we showcase *MDP_DT*'s ability to benefit from multiple parameters and perform correct decisions.

System and Algorithm Setup: We use an *HBase 1.1.2* NoSQL distributed database cluster over a private *Openstack* cloud setup. We generate a mix of different read and write intensive workloads of varying amplitude by utilizing the



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

Fig. 1: Comparison of the behavior of model-based vs model-free and decision-tree based vs static algorithms.



(a) All parameters (b) One min load

Fig. 2: Behavior when restricting splitting parameters

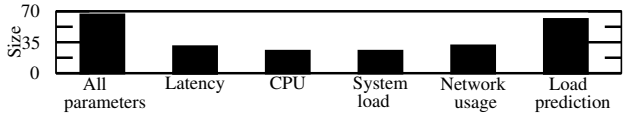


Fig. 3: Decision tree size when restricting splitting parameters

YCSB benchmark. Cluster coordination is performed by a modified version of [7]. We allow for 5 different actions, which include adding or removing 1 or 2 VMs, 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 also confirms our full-model choice, since this time is adequate to calculate the updated model after every decision. Each VM has 1GB of RAM, 10GB of storage space and 1 vCPU. Decision tree based models are trained with a set of 12 parameters including: Cluster size, VM metrics (free RAM, number of vCPU's, CPU utilization and storage capacity), I/O reqs per second, I/O wait CPU, a linear prediction of the next incoming load, percentage of read queries, average query latency and network utilization. The model-based algorithms update their optimal policies utilizing *Prioritized Sweeping* [11]. For the static partitioning algorithms we select two dimensions that were found to be the most relevant for the cluster performance (the cluster size and the linear load prediction) divided in 12 and 8 equal partitions respectively, resulting in 96 states. This setup is optimal for the static schemes, as they require a small number of relevant states. The reward function is $R_t = min(capacity(t+1), load(t+1)) - 3 \cdot vms(t+1)$ and encourages the agent to increase the cluster size so that it can fully serve the load, but punishes it for going further.

Using Different Algorithms: We compare model-free (i.e., *Q-learning*) and static partitioning schemes. These combinations lead to four different algorithms, namely the model-based adaptively and statically partitioned *MDP_DT* and *MDP* respectively, and the respective model-free versions (*Q_DT* and *Q-learning*). In Fig. 1 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 action of adding or removing VMs. A small training set challenges the algorithm's adaptation.

MDP_DT follows the applied workload very closely (Fig. 1a). The full-model based *MDP* algorithm also follows the incoming load reasonably well even when trained with the small dataset (Fig. 1b). Since this problem has a reasonably simple state space a partitioning using only the size of the cluster and the incoming load is quite sufficient to capture its behavior. Yet, in sudden load spikes in the max and min load values it takes more time to respond compared to *MDP_DT*.

The *Q-learning* based algorithms though both fail to follow the incoming load effectively (Figs 1c, 1d). The decision tree based *Q-learning* algorithm (*QDT*) achieves the weakest performance (Fig. 1c). 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. However, they both are noticeably less stable (they cannot follow the observed load in an adequate manner) compared to the full model approaches.

Restricting the Splitting Parameters: In order to test the algorithm's ability to partition the state space using different parameters 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. We use CPU utilization, the one minute averaged reported system load, the prediction of the incoming load, the network usage and the average latency.

Fig 2 present our findings when using all parameters compared to using only the one minute load. 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. In Fig. 2b decisions do not follow workload as smooth as in Fig. 2a. In Fig. 3 the decision tree size for each case reflects this fact, where the model has 20 to 30 states compared to the 66 of the default case. However, proving 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 use these correlations to implement policies in systems with complicated and not very well understood behavior.

ACKNOWLEDGMENT

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

REFERENCES

[1] A. Verma *et al.*, "Large-scale Cluster Management at Google with Borg," in *EuroSys*, 2015.

[2] C. Tang *et al.*, "Holistic Configuration Management at Facebook," in *SOSP*, 2015.

[3] "AWS | Auto Scaling," http://docs.aws.amazon.com/autoscaling/latest/userguide/as-scale-based-on-demand.html.

[4] S. Das *et al.*, "ElasTraS: An Elastic, Scalable, and Self-managing Transactional Database for the Cloud," *TODS*, 2013.

[5] H. Nguyen *et al.*, "Agile: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service," in *ICAC*, 2013.

[6] X. Bu *et al.*, "Coordinated Self-configuration of Virtual Machines and Appliances using a Model-free Learning Approach," *TPDS*, 2013.

[7] D. Tsoumakos *et al.*, "Automated, Elastic Resource Provisioning for NoSQL Clusters Using Tiramola," in *CCGRID*, 2013.

[8] E. Kassela *et al.*, "Automated Workload-aware Elasticity of NoSQL Clusters in the Cloud," in *BigData*, 2014.

[9] E. Barrett *et al.*, "Applying Reinforcement Learning Towards Automating Resource Allocation and Application Scalability in the Cloud," *Concurrency and Computation: Practice and Experience*, 2013.

[10] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris, "Elastic Resource Management with Adaptive State Space Partitioning of Markov Decision Processes," *arXiv:1702.02978 [cs]*, Feb. 2017.

[11] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement Learning With Less Data and Less Time," *Machine Learning*, 1993.

[12] W. T. Uther and M. M. Veloso, "Tree Based Discretization for Continuous State Space Reinforcement Learning," in *Aaai*, 1998, pp. 769–774.