Proceedings of the 2011 Winter Simulation Conference
S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, eds.

USING APPROXIMATE DYNAMIC PROGRAMMING TO OPTIMIZE ADMISSION CONTROL IN CLOUD COMPUTING ENVIRONMENT

Zohar Feldman
Michael Masin

IBM Haifa Research Labs
Haifa University Campus
Haifa 31905, Israel

Asser N. Tantawi
Diana Arroyo
Malgorzata Steinder

IBM T. J. Watson Research Center
19, Skyline Drive
Hawthorne NY 10532, USA

ABSTRACT

In this work, we optimize the admission policy of application deployment requests submitted to data centers. Data centers are typically comprised of many physical servers. However, their resources are limited, and occasionally demand can be higher than what the system can handle, resulting with lost opportunities. Since different requests typically have different revenue margins and resource requirements, the decision whether to admit a deployment, made on time of submission, is not trivial.
We use the Markov Decision Process (MDP) framework to model this problem, and draw upon the Approximate Dynamic Programming (ADP) paradigm to devise optimized admission policies. We resort to approximate methods because typical data centers are too large to solve by standard methods. We show that our algorithms achieve substantial revenue improvements, and they are scalable to large centers.

1 INTRODUCTION

Today, large data centers that automate on-line deployment of virtual machines (VM) that vary in resource requirements, lifetime and revenue margins face the possible loss of revenue at high utilization. As the utilization of a data center approaches its capacity, some deployments fail due to fragmentation in the data center. The problem of resources provisioning and dynamic placement of deployments has received a great deal of attention over the years (cf. Urgaonkar, Shenoy, and Roscoe 2002 and references therein). These works propose resource provisioning techniques that satisfy service levels agreement (SLA), taking into account workload traces and profiles, possibly taken while in service. Although existing intelligent placement technology can effectively minimize fragmentation, a solution that relies entirely on such placement technology may still suffer revenue degradation. For example, some deployments have higher revenue margins than others, and therefore a system that takes that into account will offer improved overall revenue than one that does not. It may be preferable to fail a deployment with lower revenue margin even though enough capacity is available in order to leave capacity for future deployments with larger margins.

We model the cloud admission problem as a Markov Decision Process (MDP). The MDP framework is broadly used to model and solve dynamic control problems (Puterman 1994), and therefore arise as a natural framework to model the cloud admission problem. MDPs can be solved analytically or numerically using dynamic programming (DP) methods. Unfortunately, these methods are not applicable for high-dimensional problems, a phenomena which is often referred to as the curses of dimensionality (Bellman 1961). Data centers are typically large and may include hundreds and even thousands of nodes. The number of states in which the system can be, tends to be immense. This leads to the fact that only very small instances can be solved efficiently by standard DP methods.

To overcome this difficulty, we draw upon techniques from the Approximate Dynamic Programming (ADP) paradigm. Comprehensive study on ADP can be found in D.Bertsekas and J.Tsitsiklis (1996) and Powell (2005). In very general lines, ADP is a generic name for a large set of techniques that approximate the value function in one way or another. Many of these techniques utilize simulation, and do not rely on having parametric model.
ADP was shown to achieve good results in many hard problems which could not be solved by standard DP techniques. However, there are also examples in which it fails miserably. Moreover, since ADP methods are typically involved with problem-specific configuration and design parameters setting, it is a common feeling that making it work properly is a work of art.

We explore several ADP schemes to solve our problem. In the first approach, the optimal value function is approximated using sample runs produced by simulation. This is achieved by using the Bellman optimality equations for states that are visited throughout the simulation. We use several common techniques such as post-states, basis functions, and states aggregation in order to reduce the problem complexity.
In another approach, we enhance the previous schema by introducing classification (e.g. decision tree) to represent our policy. The learning set, used to build the classifiers, are pairs of visited states and actions that optimize the future approximated value. For last, we propose a schema based on parametric policies. Namely, we consider a parametric family of policies, and optimize the parameter to obtain the best policy. We show that this approach is very attractive in several aspects.
Our proposed algorithms are designed to cope with large data centers, and the policies we derive are simple and easy to implement (in fact, parametric policies can be implemented by rule engines). We also propose a way to use our algorithms together with additional components, like forecast, to form a proactive deployment management system. The system we propose is proactive in the sense that it continuously makes prediction about the future behavior, and updates the policy in order to better prepare to changes in demand and requirements.

The remainder of the paper is organized as follows. In Section 2 we describe in details the cloud settings, and define the cloud admission problem. In Section 2.2 the cloud admission problem is modeled as a Markov Decision Problem. We continue with a presentation of our ADP algorithms and their specific configuration in Section 3, accompanied by a numerical study. In Section 5, we describe the proactive admission management system that proactively handles the admission of deployment requests. Finally, we conclude this work and propose future research in Section 6.

2 PROBLEM FORMULATION

2.1 Cloud Settings

The cloud setting can be described as follows. The data center is comprised of K physical machines called nodes. Each node k has a specific capacity c_{kj} from resource type $j = 1,\dots,J$. In our case, we consider three types of resources ($J = 3$): disk space, cpu and memory. The cloud receives deployment requests of I types which vary in different aspects. We model the arrivals of each type $i = 1,\dots,I$ as a Poisson Process with rate λ_i. Each deployment has a life time assumed to follow exponential distribution with type-dependent mean μ_i. It is assumed that the life time of a deployment is known at the time it is submitted to the data center. Moreover, each deployment consumes a certain amount of resources during the time it is hosted. We denote by d_{ij} the resource requirement of deployment type i from resource j. The resource requirements are deterministic. Each deployment of type i has a profit rate r_i, such that a successful deployment is awarded with a profit r_i times the time units it is deployed on the cloud. The reward rate for each customer is determined by contract, and is typically correlated with the resource requirements.

The handling of an incoming deployment request includes two decisions: first, it has to be decided whether to admit the request or reject it. Second, in case it is admitted, it has to be placed on a specific node that can satisfy its resource requirements. There are no backlogs, so when a request is rejected it is lost. Obviously, a request is automatically rejected when there is no node with sufficient resources.
Our objective is to maximize the long-run revenue coming from admitted deployments.

2.2 MDP Model

We model the cloud system as a Markov Decision Process (MDP) $M = (S,A,P,R)$, where S is the set of states that the system can reach in each point it time, A is the action space - the set of admissible actions that can be taken in each state and P is a transition probability function that specifies the probability to transfer from state to state when taking a specific action. R is the reward function received when taking action a in state s. The cloud system is actually a Continuous-Time Markov Decision Processes (CTMDP). We therefore describe the MDP induced by it.
In our case, a state $s \in S$ is represented by a vector $s = (p, s_{11}, \ldots, s_{1I}, \ldots, s_{KI})$. p denotes the pending deployment request type taking values in $1 \leq i \leq I$ that indicates the type of the request, or the value 0 in case there is no pending request; s_{kj} denotes the number of requests of type i hosted on node k (i.e s_{kj} constitute the unraveling of the matrix of number of requests of type j on machine k). In our model, we choose to separate the admission and placement decisions. In fact, we assume we are given a placement function $h_{pl} : S \to \{1, \ldots, K\}$ that returns for any state s the node on which the pending request should be deployed. Therefore, the actions that can be taken in each state s only refer to the admission as follows

$$a_s = \begin{cases} 1, & \text{admit pending request } p; \\ 0, & \text{otherwise.} \end{cases}$$

In principal, it is also possible to make the admission and placement decision together. This means the action space can take values between $0, \ldots, K$. However, we separate the decisions for two main reasons: first, it simplifies our problem considerably by reducing the action state from $|K+1|$ to 2; second, it enables the incorporation of general placement policies that take into consideration other factors like load balancing etc.
The reward $R(s,a)$ is given by

$$R(s,a) = \begin{cases} r_p \cdot \mu_p, & \text{a=1;} \\ 0, & \text{otherwise.} \end{cases}$$

The transition probability is induced directly from the admission decision and the following event in the system, which can be either a new request arrival or a hosting termination. First, in case there is a pending request and the admission decision is to admit, the number of hosted deployments from that type will increase by one in the node returned by the placement function h_{pl}. Then, if the following event is a request arrival, the first component of the state will change to the corresponding type; inversely, if the event is a termination of deployment of type i from node k, the corresponding number of deployed requests decreases by one and the first component is set to 0 (no pending request). The transition probability function from state s to state s' is given formally in (1). With a slight abuse of notation, we use p and s_{ik} to denote the pending request type and number of deployed requests in state s respectively, where p' and s'_{ik} are used to denote the same in state s'.

$$P_{s,a}(s') = \begin{cases} \frac{\lambda_i}{\bar{\lambda}(s,a)}, & a * p = 0, p' = i, s' = s; \\ \frac{s_{kj}\mu_j}{\bar{\lambda}(s,a)}, & a * p = 0, p' = 0, s' = s - e_{k,j}; \\ \frac{\lambda_i}{\bar{\lambda}(s,a)}, & a = 1, p = j, p' = i, s' = s + e_{h_{pl}(s),j}; \\ \frac{s_{kj}\mu_j}{\bar{\lambda}(s,a)}, & a = 1, p = l, p' = 0, s' = s - e_{k,j} + e_{h_{pl}(s),l}; \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

$\bar{\lambda}(s,a)$ is the sum of all transition rates from state s given by

$$\bar{\lambda}(s,a) = \sum_{i=1}^{I} \left[\lambda_i + \sum_{k=1}^{K} s_{ki}\mu_i \right] + a \cdot \mu_p.$$

The objective criteria is the discounted expected reward given by

$$\max_{\pi} J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{T_t} R(s_t, \pi(s_t)) \right], \tag{2}$$

where γ is the discount factor, and T_t is the time of the decision epoch t.

Dynamic Programming (DP) is based on estimating the value function $V(s)$, which represents the value of being in state s. The value function is given by the Bellman's equations

$$V(s) = \max_a \left\{ R(s,a) + \gamma(s,a) \sum_{s'} P_{s,a}(s') V(s') \right\}. \tag{3}$$

In CTMDP, the discount factor depends on the state and action, and can be expressed by $\gamma(s,a) = \frac{\bar{\lambda}(s,a)}{\bar{\lambda}(s,a) - ln\gamma}$. Once the optimal value function is obtained, the optimal policy is derived using

$$\pi(s) = \arg\max_a \left\{ R(s,a) + \gamma(s,a) \sum_{s'} P_{s,a}(s') V(s') \right\}. \tag{4}$$

The curses of dimensionality associated with DP refer to its sensitivity to the size of the state space, the action space, and the complexity of the probability transition function (the number of states reachable from each state). DP becomes impractical for problems with high dimensionality in any of these dimensions. In our case, it is easy to see that even very small examples yield a very large state space. For instance, consider a cloud with 10 nodes and 2 types of requests. The first type occupies half of a node resources and the second type occupies a quarter. Each node has 9 possible states, and overall 9^{10} states which is around 3.5 billion states. This number is several orders larger than what standard DP methods can handle. We therefore resort to an approximate approach called Approximate Dynamic Programming (ADP). We describe how ADP is applied to our problem in the next section.

3 ADP ALGORITHMS

ADP in general is concerned with finding near optimal policies by making some form of approximation to the value function. Powell (2005) and Powell (2009) describe a generic ADP schema that we use in Algorithm 1. In each iteration n, we progress according to a sample path ω_n. In each step t, we update the value function approximation $\hat{V}(s_t)$. We proceed to the next state according to the dynamic function $S(s_t, a_t, W_{t+1}(\omega_n))$, which calculates the next state given the current state s_t, the chosen action a_t and the new information (event) W_{t+1} arriving at time $t+1$.

Algorithm 1 (Sample Value Iteration)

```
Initialization   Initialize $\hat{V}(s)$, set $n = 1$
Step 1   Generate sample path $\omega_n$
Step 2   For t=0,...,T do:
    Step 2a  Set $a_t = \arg\max_a R(s_t,a) + \gamma(s_t,a) \sum P_{s_t,a}(s_{t+1}) \hat{V}(s_{t+1})$
    Step 2b  Set $v_t = R(s_t,a_t) + \gamma(s_t,a_t) \sum P_{s_t,a_t}(s_{t+1}) \hat{V}(s_{t+1})$
    Step 2c  Set $\hat{V}(s_t) = (1-\beta_n)\hat{V}(s_t) + \beta_n v_t$
    Step 2d  $s_{t+1} = S(s_t, a_t, \omega_n)$
Step 3   if $n = N$ return $\hat{V}$, else increment n and go to Step 1.
```

Note that the implementation of Algorithm 1 requires a look-up table in order to represent the value function approximation $\hat{V}(s)$ for each visited state. In addition, for each visited state, an expectation is needed to be calculated. To avoid this, we use the concept of post-state (cf. Powell 2005).

3.1 Post States

A post-state s_t^a is used to describe the state of the system right after an action was taken at state s_t but before the arrival of a new event. In our setting, s_t^a is the state right after we make our admission decision but before the occurrence of a new request arrival or deployment termination event. Since s_t^a can be calculated independently of events that occur after step t, The expectation on **Step 2a** and **Step 2b** in 1 is replaced with $\max_a R(s_t,a) + \hat{v}(s_t^a)$. In this case, instead of using (4), the policy is derived by

$$\pi(s) = \arg\max_a R(s_t,a) + \hat{v}(s_t^a). \qquad (5)$$

We also note that our post-states can never have a pending request ($p = 0$), and therefore incorporating them in the schema brings additional value of reducing the state space. The value function is fitted around the post-state, so **Step 2c** is replaced with $\hat{V}\left(s_{t-1}^a\right) = (1-\beta_n)\hat{V}\left(s_{t-1}^a\right) + \beta_n \gamma\left(s_{t-1}^a, 0\right) v_t$.
Algorithm 1 evolve by exploring all the states traversed by simulation. However, their number may still be very large, especially for large instances. This results with highly inaccurate approximation of the value function. In what follows, we apply two common techniques to represent the value function in a more compact way: basis functions and state aggregation.

3.2 Basis Functions

A very well known approach to simplify the value function representation, and consequently the learning procedure, is to use basis functions (also called features). A basis function $\phi_i(s)$ is a real-valued function of the states. Basis functions are used to approximate the value function by a weighted sum as follows $\hat{V}_\theta(s) = \sum_i \theta_i \phi_i(s)$. Our problem reduces from approximating the value for each state to approximating only the weights θ that define the value for all states. Since the number of basis functions is significantly smaller than the number of states, we are facing with a much simpler task. There are several alternative methods to calculate the weights. A very well known method is temporal differences learning (cf. Boyan 2002, Lagoudakis and Parr 2003).
We use a simple schema as in Powell (2008), according to which the weights are updated using stochastic gradients $\theta = \theta + \beta_t \left(v_t - \theta^T \phi(s_t)\right) \phi(s_t)$.
The simplest form of basis functions are polynomial functions of the state components. In lack of good insights about the shape of the value function, they are probably the most appealing choice. We consider several other types of basis functions. The first one is based on the number of admissible deployments given by

$$\phi_{i,k}^{AD}(s) = \min\{k, AD_i(s)\},$$

where $AD_i(s)$ is the number of admissible requests of type i when the state is s. The rational behind the bound k is to make the value function rather constant when the cloud is empty, and then when the cloud is quite utilized, the value changes as the number of potential request deployments changes.
Another form of basis function is

$$\phi_i^{UL}(s) = UL_i(s),$$

where $UL_i(s)$ is the usage level of resource i in the cloud. This is calculated by dividing the resource consumption over all physical nodes by their total capacity.

3.3 States Aggregation

Another alternative to represent the value function in a compact way is to use aggregation functions. Aggregation functions $G_g(s)$ map each state s in the original state space S into a significantly smaller state space S'. S' can be a sub-space of S, or comprise states with entirely different form. Aggregation functions are used to approximate the value function as follows $\hat{V}(s) = \sum_g w_g \hat{V}_g(G_g(s))$. That is, we use a value function approximation for each aggregation level g and the overall approximation is expressed as a linear combination of these value function. Consequently, in each state we update the value function of each aggregation level according to $\hat{v}_g\left(G_g\left(s_{t-1}^a\right)\right) = (1-\beta_t)\hat{v}_g\left(G_g\left(s_t^a\right)\right) + \beta_t \bar{v} \quad \forall g$.
Again, we experiment with several types of aggregation functions. One alternative is to aggregate states by the maximal resource usage level rounded to the nearest grid point with granularity α.

$$G^{UL}(s) = round\left(\max_i UL_i(s), \alpha\right).$$

$round(x,\alpha)$ is the mathematical function that rounds x to the closest grid point with resolution α. More formally, $round(x,\alpha) = round(x/\alpha)\cdot\alpha$, where $round(x)$ is the function that rounds the number x to its closest integer. For instance, $round(0.77, 0.05) = 0.75$ since 0.75 is the closest number to 0.77 on the grid $\{n\cdot\alpha | n \in \mathbb{Z}\}$. In a variation of this aggregation function, we also looked at node-specific usage level aggregation. For example we aggregated states according to the usage level in their least utilized node, or several nodes. Alternatively, we also consider aggregation function based on the number of admissible requests.

$$G^{AD}(s) = (AD_1(s), \ldots, AD_I(s)).$$

3.4 Classification-Based Policy

The methods proposed above assist us with approximating the value function more efficiently (but less accurate, of course) than DP methods. However, in order to derive the policy by the value function, we are still required to calculate the best decision for each state either by (4) or (5). Obviously, calculating the policy in advance for each possible state is troublesome, since we would have to keep a huge look-up table. In contrary, using the value function approximation to calculate the optimized decision in real-time entails run-time performance issues. Ideally, we would like to have a more compact representation of the policy. Classification has been used for this purpose in several works (cf. (Liu et al. 2010); Rexakis and Lagoudakis 2008). In addition to being more consumable, classification-based policies has an appealing added value of providing structural insights.
In our schema, in conjunction to updating the value function approximation of each visited state, we also store the state and its optimal decision in a training set that is used for building a classifier. We found decision trees to be most appealing classification method in particular to our setting. The formal procedure is depicted in Algorithm 2.

Algorithm 2 (Sample Value Iteration with Classification)

```
Initialization   Initialize $\hat{V}(s)$, set $n = 1$
Step 1   Step 1a Generate $\omega_n$
         Step 1b Set $TS = \emptyset$
Step 2   For t=0,...,T do:
         Step 2a Set $a_t = \arg\max_a R(s_t, a) + \gamma(s_t, a) \sum P_{s_t,a}(s_{t+1}) \hat{v}(s_{t+1})$
         Step 2b $TS = TS \bigcup \{\langle s_t, a_t \rangle\}$
         Step 2c Set $\bar{v} = R(s_t, a_t) + \gamma(s_t, a_t) \sum P_{s_t,a_t}(s_{t+1}) \hat{V}(s_{t+1})$
         Step 2d $\hat{V}(s_t) = (1 - \beta_t) \hat{V}(s_t) + \beta_t \bar{v}$
         Step 2e $s_{t+1} = S(s_t, a_t, \omega_n)$
Step 3   Use training set $TS$ to obtain a classifier $\pi_{\mathscr{C}}$
Step 4   if $n = N$ return $\hat{V}$, else increment n and go to Step 1.
```

3.5 Parametric Policy

For last, we also consider parametric policies as an alternative to use classification for approximating a policy. A parametric family of policies $\pi_\theta(s) = H(s, \theta)$ is defined by a deterministic function $H : S \times \Theta \to A$, which returns an action for each state s and parameter choice θ. The parameter θ can be in general a vector defined on a support set Θ. The goal in the parametric approach is to find the optimal parameter value $\theta^* \in \Theta$ that gives rise to the best policy π_θ. Parametric policies are appealing mostly because of the fact that it is much more intuitive and sometimes insightful to come up with decision schemes than thinking about value function approximation, which is a more abstract concept. In addition, parametric policy can capture parametric business rules, which are common approach to implement policies in practical applications.
The parametric approach to policy learning is today one of the main research issues in reinforcement learning, usually under the name of Policy Gradient Algorithms (cf. (Bhatnagar et al. 2009) and references therein).

Algorithm 3 (Parametric Policy Optimization)

```
Initialization   Initialize $\theta$
Step 1   Sample $k$ states and set $S$ to be that set.
Step 2   For each $s$ in $S$ do:
         Step 2a For each $a$ in $A(s)$ approximate $\hat{Q}^\pi(s, a)$
Step 3   Solve $\theta^* = \arg\max_\theta \sum_{s \in S} \hat{Q}^\pi(s, H_\theta(s))$
Step 4   if $\theta \approx \theta^*$ return $\pi(\theta)$,else set $\theta = \theta^*$ and go to Step 1.
```

Algorithm 3 describes our parametric policy optimization procedure. In each iteration, a set of states is sampled using simulation. We use one long simulation run, and take any other i'th state we visit. Next, for each one of the states and admissible action, we approximate the state-action value $\hat{Q}^\pi(s, a) = R(s, a) + \gamma(s, a) \sum_{s'} P_{s,\pi(s)}(s) V^\pi(s)$ using roll-outs. A roll-out used to approximate $\hat{Q}^\pi(s, a)$ is a simulated trajectory of the process starting from state s with decision a, and proceeding according to the policy π thereafter T time units into the future. Given the state-action value approximations, we proceed by solving the optimization problem $\theta^* = \arg\max_\theta \sum_{s \in S} \hat{Q}^\pi(s, H(s, \theta))$. Since $H(s, \cdot)$ is not guaranteed to have any specific form, we use a heuristic called the *Cross-Entropy* method, to approximate θ^*. Finally, if the obtained policy π_θ is not changed relative to previous iteration the policy is returned as the optimal policy, otherwise the procedure is repeated with the new policy.

We have experimented with several simple parametric rules. The most prominent among them, is the following threshold policy

$$H^{TH}(s,\theta) = \begin{cases} 1, & UsageLevel(s,n) < \theta_p; \\ 0, & \text{otherwise.} \end{cases}$$

$UsageLevel(s,n)$ is the utilization of the most utilized resource, in the least utilized node (in the same sense), in state s. In general, this rule can refer not only to the least utilized node, but also to the second least utilized or any other order.
A results summary of a selection of our experiments is provided in Section 4.

4 NUMERICAL EXPERIMENTS

Our experiments are based on the following basic settings. There are seven types of deployment requests. Their characteristics are summarized in Table 4.

Table 1: Basic Cloud Settings.

Type	B2	S2	G2	B4	S4	G4	P4
λ	24	3	2	2	0.25	0.25	0.0625
μ	1	8	8	2	16	16	64
d_{cpu}	1	2	4	2	4	8	16
d_{mem}	2	4	4	4	8	16	16
d_{disk}	1	2	2	4.86	5.85	5.85	11.70
r	0.085	0.17	0.68	0.34	0.68	1.2	2.4
$\frac{r}{d_{cpu}}$	0.085	0.085	0.17	0.17	0.17	0.15	0.15

The cloud is comprised of eight identical nodes with the following resource capacities: cpu capacity is $c_1 = 32$, memory capacity $c_2 = 64$ and disk space $c_3 = 29.26$.
We produce several scenarios with varying cloud sizes and load factors, by scaling the basic settings. This is achieved by multiplying the number of nodes, and the overall arrivals rate. The load factor in our case is defined by

$$LF(\lambda,\mu,d,c,K) = \max_j \frac{\sum_i \frac{\lambda_i}{\mu_i} d_{ij}}{K \cdot c_j}. \tag{6}$$

$\frac{\lambda_i}{\mu_i} d_{ij}$ is the offered load on resource type j, coming from requests type i, and the summation of them yield the overall offered load on resource j. The offered load can be interpreted as the expected resource usage in steady state, assuming the cloud has infinite resource capacities. Dividing the offered load on resource j by the total resource j capacity $K \cdot c_j$ gives the load factor, indicating how loaded resource j is. It is easy to see that the basic settings yield a load factor 1. Moreover, the restricting resource in our case is cpu. The ratio between the cpu requirement and the revenue rate would give us some sense about which are the more favorable request types and which are less. For instance, the last row in Table 4 shows that types $B2$ and $S2$ are ”less profitable” than others.
The baseline policy we use for comparison is a naive ”Admit All” policy. According to this policy, each deployment request that arrives to the system and can be hosted on one of the nodes, is accepted; otherwise, it is rejected. We use the same placement function h_{pl} in all our experiments. Our experiments revealed that the choice of a placement function has a strong impact over the profitability. The placement function we chose is one that is rather simple to implement, and performs not worse than any other complex placement

function we experimented with. This placement function keeps all nodes ordered by their usage level. The first node is the node with the greatest usage level, and the last is the least utilized node. Whenever a new request needs to be placed, the selected node would be the the first node, starting from the most utilized, that can accommodate the request. Note that in order to maintain the order, it is only required to permeate the place of the node whose state was changed, either because of a new deployment or a termination. Simpler placement functions like Round Robin or First Fit were found to be significantly inferior.

Table 4 summarizes the results of 9 scenarios. The size of the cloud was varied between 8, 80 and 800 nodes; the load factor was varied between 1, 1.2 and 1.4. Altogether, there are 9 combinations corresponding to the 9 scenarios. We ran all of our algorithms on each of these scenarios, and compared the obtained policies to the naive "Admit All" policy. All of the policies were evaluated by simulation. We constructed confidence intervals using conventional formulas with confidence level of 0.95 and accuracy of 0.01. The discount factor we use in each of the experiments is 0.99. We denote by J_p the discounted reward of policy p, divided by the number of nodes, where p takes the following values

AA Admit All policy
BF The policy obtained by the generic schema with basis functions.
AGG The policy obtained by the generic schema with state aggregation
CLSS The policy obtained by applying classification to the best between the basis function schema or aggregation schema
PAR The policy obtained by the parametric policy schema

The displayed results always refer to the best design choice in each schema. For the schema with basis functions, the number of hosted deployments of each type produced the best results. Namely, there are two basis function for each deployment type, corresponding to the number of hosted deployments of that type in the power of $i = 1,2$. For the state aggregation schema, we found that aggregation based on the usage level of the least x utilized nodes, with granularity α works best. In each experiment, we used several levels of aggregation corresponding to each combination of $\alpha = 0.05, 0.1$ and $x = 3, 10, 20$. The classification is always added on top of the basis function approximation or the aggregation schema. When it is incorporated with state aggregation, we use the most aggregate level (i.e., smallest α and largest x) state attributes to train the decision tree. With basis functions, the values of each basis function were used as the attributes.

Table 2: ADP Algorithms Results.

#Nodes	Load Factor	$\hat{J}_{AA}$	$\hat{J}_{BF}$	$\hat{J}_{AGG}$	$\hat{J}_{CLSS}$	$\hat{J}_{PAR}$
8	1.0	3.65	3.7(+1.1%)	3.73(+1.64%)	3.71(+1.64%)	3.78(+3.56%)
	1.2	3.78	3.98(+5.3%)	4.01(+6.1%)	3.98(+5.3%)	4.05(+7.1%)
	1.4	3.82	4.31(+12.8%)	4.28(+11.5%)	4.26(+11.5%)	4.29(+12.3%)
80	1.0	4.13	4.14(+0.2%)	4.17(+1%)	4.17(+1%)	4.18(+1.2%)
	1.2	4.14	4.38(+5.8)	4.18(+1%)	4.37(+5.5%)	4.43(+7%)
	1.4	4.09	4.79(+17%)	4.21(+2.6%)	4.77(+16.5%)	4.79(+17%)
800	1.0	4.31	4.32(+0.2%)	4.33(+0.5%)	4.32(+0.2%)	4.37(+1.4%)
	1.2	4.26	4.38(+2.8%)	4.35(+2.1%)	4.36(+2.3%)	4.46(+4.7%)
	1.4	4.17	4.77(+14%)	4.36(+3.8%)	4.77(+14%)	4.79(+14.5%)

Table 4 highlights several major findings. First, it is interesting to see that with Admit ALL (AA) policy, the revenue increases with the load factor in a small data center (8 nodes), and decreases in a large data center (800 nodes). Since there is no penalty for rejecting requests, we expect the optimal revenue to increase with the load factor. Therefore, there is a good potential for improvements in revenue. Second, in all cases, our algorithms achieved significant improvement. The percentage in parenthesis next to the estimated revenues, represents the improvement over Admit All policy. The parametric policy seems to achieve the best results in all cases, and it works well also in large instances. As expected, the improvement is more significant as the load factor increases. It is also interesting to see that the incorporation of classification to the basic schema does not come at the expense of performance, yet it produces simpler policies.

5 PROACTIVE CLOUD DEPLOYMENT MANAGEMENT

In previous sections, we described efficient methods to produce optimized admission policies for cloud systems. In this section we take one step further and describe a deployment management system that handles the admission of deployment requests while taking into account predicted behavior. This system derives the relevant parameters needed to carry out the optimization, and also decides when it is necessary to update the policy. Ideally, we would like this system to have advanced predictive capabilities, so that we could prepare for irregular behavior before they happen. For instance, if there is a football game, and during this time, it is predicted that the demand of a certain deployment type (e.g. football site) will increase, we would like the system to update its policy to reflect this modified behavior. We may want to stop accepting requests from other types or conversely we would like to accept more. This way of thinking falls into what we call *proactive event-driven computing paradigm*. Figure 1 proposes a conceptual model of such a proactive system.

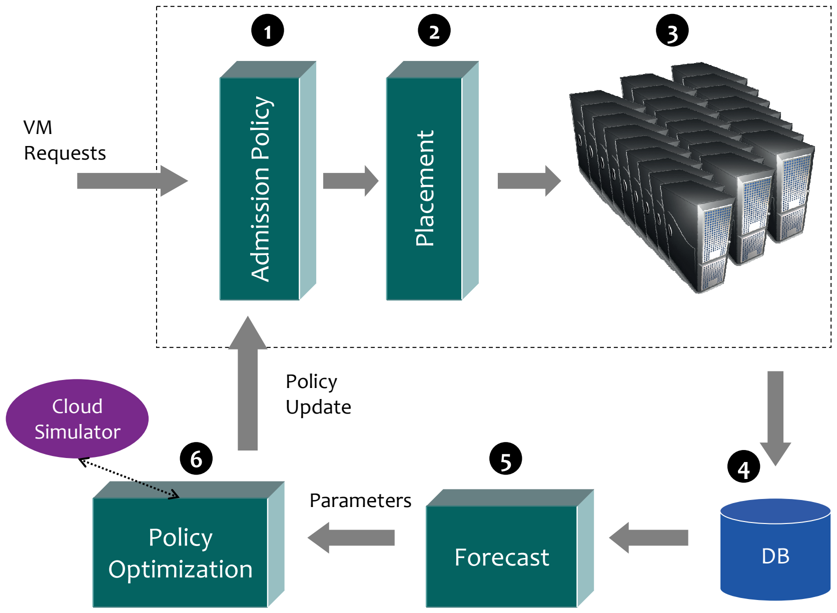

Figure 1: Deployment Management System.

The deployment management system consists of several modules:

1. **Admission Control Module** - handles the executing the admission policy
2. **Placement Module** - places the admitted request on a specific physical node
3. **Data Center** - contains the physical nodes
4. **Database** - stores historical transaction and relevant data that can be used to forecasting
5. **Predictive Module** - analyze historical data and apply statistical tools to predict the future load
6. **Policy Optimization Module** - calculates optimal policy for specific settings

The flow between the modules goes as follows: Virtual Machines (VM) requests that arrive to the cloud first go through the **Admission Control Module**, on which the admission policy calculated by the **Policy Optimization Module** is deployed. Based on the request type and the state of the system, the request may either be admitted or rejected. In case it is admitted, it is sent to the **Placement Module** which in turn places the request on a physical node according to a placement function. Each event of the system, either a request arrival or a deployment termination is logged in the **Database**. The **Database** contains historical requests and deployments data. More specifically, the type of event, arrival time, termination time, resource consumption as well as external relevant data. Based on this data, the **Predictive Module** periodically polls the tables in order to calculate a forecast on future arrival rates for all types, the expected life-time and resource requirements. Whenever the forecast differs from the current model, the new parameters are sent to the **Policy Optimization Module**. This module generates a simulation according to the new parameters and runs the ADP algorithms to obtain a policy. This policy is updated back in the **Admission Module**. Special considerations as to how much time to run the optimization, what is considered significant change, and how far into the future to look, are interesting, but are out of scope of this paper.

6 CONCLUSIONS

In this work we use MDP to model the admission control of deployment requests to a cloud hosting system. We focus on the goal of optimizing the long-run reward received from the deployments that were admitted to the cloud. However, the same approach can be applied to other objectives, such as service level. Since the state space tends to be immense, we resort to approximative methods. We have introduced three main algorithms built upon common approximation approaches from the field of ADP and Reinforcement Learning.
We show through a numerical study that our proposed algorithms achieve substantial improvement in profitability compare to naive admission policies. Moreover, the main advantage of our proposed algorithms is that they are applicable to large cloud systems in the sense that both the optimization and the execution of the derived policies are almost independent of the size of the cloud. The parametric policy approach seems the most attractive for several reasons. It produced the best results, it is compact and easy to implement, and it is much easier to design than the other approaches.
We also use the admission control in a broader context of a deployment management system. We describe a complete solution that includes prediction and optimization in order to pro-actively prepare for predicted workload.

A very interesting future direction is to address service level agreements (SLA) implicitly in the model. The most popular type of SLA in the context of the cloud, is the blocking probability - the percentage of requests from a specific customer that are rejected and hence lost. These can be measured on different time basis such as a day, a week, a month or a year. Another important research question is what should be the optimal resource capacities and design of the cloud. In this work, the capacities were assumed to be given. In practice, there is a trade-off between factors that tend to increase the capacities such as SLAs and revenues, and factors that tend to decrease the capacities such as energy costs, maintenance costs, depreciation etc. In order to obtain a truly optimized cloud system, the design, capacity planning and admission control should be addressed together.
There are also more advanced settings that were not regarded in this work. Some of these include time varying parameters, batch arrivals of requests, deployments with changing resource requirements and more.

REFERENCES

Bellman, R. E. 1961. *Adaptive control processes: A guided tour*. Princeton University Press.
Bhatnagar, S., R. Sutton, M. Ghavamzadeh, and M. Lee. 2009. "Natural Actor-Critic Algorithm". *Automatica* 45:2471–2482.

Boyan, J. A. 2002. "Least-Squares Temporal Difference Learning". *Machine Learning* 49:233–246.

D.Bertsekas, and J.Tsitsiklis. 1996. *Neuro-dynamic programming*. Belmont, MA: Athena Scientific.

Lagoudakis, M. G., and R. Parr. 2003. "Least-Squares Policy Iteration". *Journal of Machine Learning Research* 4:1107–1149.

S. Liu and A. Panangadan and A. Talukder and C. S. Raghavendra 2010. "Compact Representation of Coordinated Sampling Policies forBody Sensor Networks".

Powell, W. B. 2005. *Approximate Dynamic Programming for Operations Research*. New York: John Wiley.

Powell, W. B. 2008. "Approximate dynamic programming: lessons from the field". In *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, 205–214: Winter Simulation Conference.

Powell, W. B. 2009. "What you should know about approximate dynamic programming". *Naval Research Logistics* 56:239–249.

Puterman, M. L. 1994. *Markov Decision Process: Discrete stochastic dynamic programming*. New York: John Wiley.

Rexakis, I., and M. G. Lagoudakis. 2008. "Classifier-Based Policy Representation". In *Proceedings of the 2008 Seventh International Conference on Machine Learning and Applications*, 91–98. Washington, DC, USA: IEEE Computer Society.

Urgaonkar, B., P. Shenoy, and T. Roscoe. 2002, December. "Resource overbooking and application profiling in shared hosting platforms". *SIGOPS Oper. Syst. Rev.* 36:239–254.

AUTHOR BIOGRAPHIES

ZOHAR FELDMAN is a Research Staff Member at IBM Research Labs in Haifa, focusing on the development and application of operation research methods for practical business optimization problems. He is currently a Ph.D. candidate at the Technion. His main research interests include stochastic optimization and control, and simulation-based methods. His email address is zoharf@il.ibm.com.

MICHAEL MASIN is a Research Staff Member in the Business Optimization group at IBM Research Haifa Lab (HRL) and has strong teaching and research ties to the Technion and Tel Aviv University, Israel. Michael received his Ph.D. degree in Industrial Engineering from the Technion in 1998. His research is in the area of deterministic and stochastic combinatorial multi-objective optimization including (1) Systems Engineering and System of Systems design and (2) design, control, and integration of production, service, and logistics systems. His email address is michaelm@il.ibm.com

ASSER N. TANTAWI is a research staff member at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. He received his Ph.D. degree in computer science from Rutgers University in 1982. During his tenure at IBM, he has worked on several areas including performance modeling and analysis. His email address is tantawi@us.ibm.com.

DIANA ARROYO is a Software Engineer at IBM T. J. Watson Research Center. Her work focuses in the field of virtualization, placement and performance management in the cloud. Diana received a M.S. degree in Software Engineering from the University of Texas at Austin in 2004. Her email address is darroyo@us.ibm.com.

MALGORZATA (GOSIA) STEINDER is a scientist and a manager in IBM T. J. Watson Research Center, Hawthorne, NY. She is working on automating the performance management of large scale cloud environments. She received M.S. degree in Computer Science from AGH University of Science and Technology, Poland and a Ph.D. in Computer and Information Sciences from the University of Delaware. Her email address is steinder@us.ibm.com.