Monitoring-Based Task Scheduling in Large-Scale SaaS Cloud

Puheng Zhang(✉), Chuang Lin, Xiao Ma, Fengyuan Ren, and Wenzhuo Li

Tsinghua National Laboratory for Information Science and Technology,
Department of Computer Science and Technology,
Tsinghua University, Beijing 100084, China
zhangph14@mails.tsinghua.edu.cn, chlin@tsinghua.edu.cn

Abstract. With the increasing scale of SaaS and the continuous growth in server failures, task scheduling problems become more intricate, and both scheduling quality and scheduling speed raise further concerns. In this paper, we first propose a virtualized and monitoring SaaS model with predictive maintenance to minimize the costs of fault tolerance. Then with the monitored and predicted available states of servers, we focus on dynamic real-time task scheduling in large-scale heterogeneous SaaS, targeting at jointly optimizing the long-term performance benefits and energy costs in order to improve scheduling quality. We formulate a dynamic programming problem, where both the state and action spaces are too large to be solved by simple iterations. To address these issues, we take advantage of Machine Learning theory, and put forward an approximate dynamic programming algorithm. We utilize value function approximation and candidate-heuristic method to separately solve state and action explosions. Thus, computation complexity is significantly reduced and scheduling speed is greatly enhanced. Finally, we conduct experiments with both random simulation data and Google cloud trace-logs. Qos evaluations and comparisons demonstrate that our approach is effective and efficient under bursty requests and high throughputs.

Keywords: Multi-objective optimization · SaaS cloud · Data center · Task scheduling · Approximate dynamic programming

1 Introduction

Currently, services and businesses of SOFTWARE-AS-A-SERVICE (SaaS) cloud are exponentially growing, and one cloud data center is often built with thousands of servers, equipped with complex networking and power apparatuses [2]. SaaS brings vast opportunities and enormous benefits, but introduces many new requirements and challenges.

For one thing, a more efficient scheduling algorithm is in urgent need to cope with server failures. Due to inexpensive commodity hardware equipments, cloud service providers are faced with high hardware and software failures [19]. Thus, the maintenance culture is on the move [13]. In the era of "Internet of Things", all manner of equipments are embedded with intelligent sensors, and sophisticated

Q.Z. Sheng et al. (Eds.): ICSOC 2016, LNCS 9936, pp. 140–156, 2016.
DOI: 10.1007/978-3-319-46295-0_9

analysis can then be applied to describe the health status of servers and predict the needs for repairs in advance. Therefore, predictive maintenance, which means identifying problems and executing maintenance procedures beforehand, is strongly recommended and advocated recently [8]. It can effectively minimize unplanned asset downtime, make full use of resources, and reduce maintenance costs. With predictive maintenance, health states of servers can be real-time monitored and predicted, and scheduling a request to an unhealthy server can be avoided maximally. Otherwise, if unhealthy servers break down half way, unfinished tasks should be redone. No matter "Check pointing/Restart" or other reactive fault-resilient mechanisms will lead to huge waste of time and resources.

For another, two targets, scheduling quality and speed, deserve equal attentions. In order to maximize resistance to failures, provide good performance, save energy, and cut costs, careful placement of tasks is needed but it is always ignored by sampling methods. Meanwhile, fast scheduling is also important to guarantee user experience, especially for real-time tasks, but many complex multi-objective scheduling algorithms last long. Therefore, the two targets should be reconciled, and a new approach is required.

Besides, virtual machine (VM) migrations ought to be avoided as possible, for they are time consuming and energy intensive. In most cases, there are two motives of VM migrations, one for fault tolerance and the other for energy consolidation. If under the scheme of predictive maintenance, only VMs on healthy servers can be allocated with tasks to avoid faults. At the same time, if tasks are first distributed to VMs on the busy servers for energy consolidation purposes, VM migration incidents can be minimized. What is more, servers with all their VMs switched off can then hibernate, thus auto-scaling can be realized.

In theory, multi-objective task scheduling problem for large-scale cloud data centers is rather complicated. Not only should we arbitrate the tradeoff of multi factors such as performance and power, but also we need to solve the problems of state and decision explosions. Consider the real-time queue of each VM. Each queue length can take continuous values. The number of states for one server is uncountable and infinite, let alone the combination states of a huge number of servers. When tasks or requests arrive, they should be allocated to some of the massive available servers. The static distribution problem for thousands of servers itself is NP-hard, let alone the dynamic scheduling for various types of tasks. If we pursue long-term overall rewards, the method of Markov Decision Processes (MDP) is often used. However, it does not work in this scenario due to curses of dimensionality [15]. When the number of servers is very large, many classical algorithms may also not work well within an acceptable time limit.

Tasks scheduling is a classical problem in cloud data centers, and there have already been many algorithms to address this issue. Liu et al. [10] built an analytical framework to do the task scheduling in SaaS clouds, and Alahmadi et al. [1] developed a new, energy-aware task scheduling framework. However, they are both under the assumption of homogenous servers, and do not take into account failures of servers and deadlines of tasks. Hosseinimotlagh et al. [7] proposed a cooperative two-tier approach for scheduling real-time tasks to benefit both cloud providers and their customers. Mao et al. [11] put forward

a task scheduling algorithm concerning the delay of the associated tasks in cloud computing systems. Zhu et al. [21] developed an energy-aware scheduling algorithm in cloud for real-time, periodic, independent tasks in virtualized clouds. Nevertheless, they do not consider metrics of throughput, and energy consolidation for various types of tasks. Cheng et al. [3] proposed an energy-saving task scheduling algorithm based on the queuing theory. Yet, it relied on the assumption that the coming tasks must conform to an established distribution. Moreover, most limited the number of servers, for multi-objective scheduling in large-scale commercial SaaS can not be efficiently conducted by their algorithms.

Consequently, all previous studies cannot simultaneously address all the requirements and challenges mentioned above. Motivated by the need of high efficient real-time tasks scheduling algorithms in large-scale heterogeneous SaaS cloud, we put forward a heuristic approximate dynamic programming (H-ADP) algorithm to jointly optimize the performance and power with predictive maintenance. The main contributions of this work are as follows:

(1) We put forward a virtualized and monitoring model of SaaS cloud with predictive maintenance, based on which we constitute a scheduling rule to minimize the costs and overheads caused by fault tolerance.
(2) We introduce popular Machine Learning theory into solving traditional stochastic dynamic programming (SDP) problems, and propose a novel task scheduling algorithm, simultaneously considering scheduling speed and quality. Both random synthetic data and real trace-logs are used in experiments to demonstrate the applicability and superiority of our approach under bursty requests and high throughputs.
(3) We solve the problems of both state and action explosions in task scheduling problems. Firstly, we carefully design the basis function with the method of value function approximation (VFA), by which state values can be parameterized and recursively estimated step by step. And in this way, we effectively solve the optimization problems of infinite states. Secondly, we develop a candidate-selection heuristic algorithm in the procedure of policy search, and effectively solve the optimization problems that contain massive decision variables. Henceforth, we make it possible to conduct time-effective multi-objective optimizations for SDP problems with extremely large state and action spaces.

The remainder of this paper is organized as follows. In Sect. 2, we demonstrate the model of a heterogeneous SaaS cloud, and formulate the scheduling problem as an SDP problem. Section 3 proposes the Heuristic ADP algorithm. Simulations and experiments are conducted to make Qos evaluations in Sect. 4. Section 5 concludes the paper.

2 Problem Formulation

In this section, we illustrate models and notions used in this paper, introduce predictive maintenance strategy, and formulate an SDP problem for task scheduling issues in SaaS.

2.1 System Model

A virtualized and monitoring model for SaaS cloud can be described in Fig. 1 [10]. A cloud data center is comprised of J heterogeneous servers, each of which is virtualized as I types of VMs to process the corresponding I different types of tasks. We set C_j as the processing capacity of the jth server, which is measured in (Million Instructions Per Second) MIPS. Without loss of generality, we assume a server's processing capacity is fairly distributed among its I hosted VMs in this work [10]. Then the processing capacity of each VM is $1/I$ of the total processing capacity of its hosting server. Each VM is equipped with a buffer queue, and we define the queue length of type-i VM on Server j, $Q_{i,j}$, as the total volume of tasks waiting to be processed, and it is measured in Million Instructions (MI). $R = \{R_1, \ldots, R_k, \ldots, R_K\}$ characterizes K independent and non-preemptive requests. Each coming task can be marked as a four tuple, and $R_k = (ArrTime, Type, Size, Deadline)^T$. $ArrTime$ means the arriving time of a task, and $Type$ represents the service type. $Size$ denotes the length of instructions measured in Million Instructions, and $Deadline$ indicates the point of time before which the task must be completed.

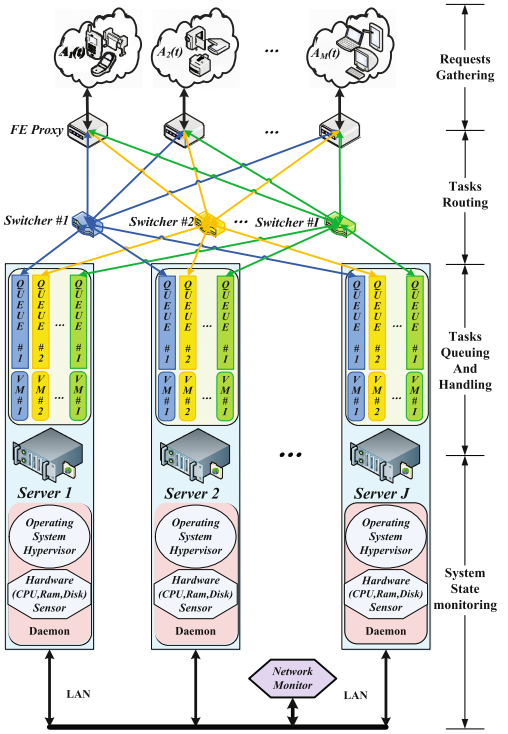

Fig. 1. A virtualized and monitoring SaaS model under predictive maintenance

When no hardware and software failures happen, the normal serving flow is as follows. Customer requests or tasks are first collected by adjacent front-end proxies, and then distributed to the relative type of request routing switchers. A switcher is responsible for distributing a specified type of requests to proper servers. Then the corresponding type of VM-queues on each server, buffer and pool the tasks. Corresponding VMs handle the tasks and return results. The serving flows for each type of tasks are marked with different colors in Fig. 1. If a task cannot be finished within its deadline, it will be abandoned and result into a penalty.

2.2 Scheduling with Predictive Maintenance Strategy

In practice, we ought to consider probabilities of various kinds of failures in SaaS. Currently, modern processors are equipped with sensors that can be used to monitor CPU temperature, fan speeds, and other parameters [9]. One of the most commonly used examples is "Lm-sensors" [20]. Operating systems and VM software can also be monitored by hypervisors [18]. Additionally, many sophisticated patents and products have involved into the network monitoring [12]. These sensors can be equipped as described in Fig. 1. So the overall health states of servers can be real-time monitored and predicted by virtue of the comprehensive analysis of hardware, software and network sensor feedbacks. Specific calculation procedure for predicting health states of servers is illustrated in [4]. We define $Sa = (Sa_1, Sa_2, \cdots, Sa_J)^T$ as the predicted result of server available states, and each element can be "1" or "0", respectively denoting a server is available or not to receive tasks.

Scheduling with predictive maintenance strategy can be summarized in one phrase: only assigning applications to the servers that are "healthy" enough to handle and finish the tasks. In other words, after predictions, if one server is imminent to break down because of hardware, software or network reasons, and whatever the reason, new tasks are not permitted to be allocated to that server and it must receive repairs or treatment right away. When the server is fixed out and all states return to normal, it can receive tasks as before.

2.3 An SDP Problem

A standard SDP problem is usually comprised of 5 ingredients as follows.

Decision Time Epochs. Decisions are executed at the end of each time slot, i.e., at $t = \tau, 2\tau, 3\tau \cdots$, and τ is the decision time interval.

States. State S_t can be categorized as two components: endogenous states matrix Q_t, which change with different actions, and exogenous states tuple $EX_t = (R_t, Sa_t, C)$, which are only determined by external factors, irrelevant to specific actions. Endogenous state matrix Q_t characterizes the queue length of all VMs on all servers at time t, and it is a matrix comprised of elements $Q_{t,i,j}$, which means the type-i VM queue length on server j. $Q_{t,i,j}$ varies over time due to the arriving tasks and serving rates. Exogenous state R_t denotes the coming tasks during one time slot. Vector $Sa_t = (Sa_{t1}, Sa_{t2}, \ldots, Sa_{tJ})^T$ captures the available state of each server at each time slot. Vector C represents the processing capacity of servers mentioned in Sect. 2.1, and it does not change with time.

Suppose that a data center is comprised of 10000 nodes, each node is in possession of a buffer queue, and the length of each queue is separated into 1000 discrete values. Suppose there are totally 8 types of requests, and a VM can be in a binary state of available or not. Then the total number of states would be $(1000 \times 2)^{10000 \times 8}$, absolutely an astronomical number! Moreover, the queue length of each VM can take continuous values, so the number of states is uncountable and infinite.

Actions. At each system state S_t, there is a corresponding control action $x_t(S_t)$. At each time epoch t, considering n_t tasks ($n_{t,i}$ tasks of type-i) are coming, and then the decision (action) is to assign each task to the proper VM among huge number of nodes. The actions at each time slot comprise a set of n_t-dimensional vector, i.e. $(100, 2098, 298, \ldots, 3980)$, and the lth element represents a server index for assigning the lth task. Under the same assumption of a cloud data center with 10000 servers, then one task may have 10000 choices, and let $n_t = 5000$. Therefore, for only a time period of 100 time slots, the size of action spaces would become 100×10000^{5000}, an astronomical number again!

Transition Function. Genetic transition function can be described as $S_{t+1} = S^M(S_t, x_t, Ex_{t+1})$ and the superscript M stands for "model" [15]. In this work, we focus on the transitions of endogenous state matrix Q_t. The dynamic forward transition function of each element in Q_t can be written as in Eq. (1). If the lth type-i task is allocated to server j at time t, $R_{t,i,l,j}$ equals the size of that task, otherwise, $R_{t,i,l,j} = 0$. Equation (1) shows that the queue length at time $t+1$ is associated with both the coming tasks and the processing capacity of each VM during time t.

$$Q_{t+1,i,j} = max\left\{\left(Q_{t,i,j} + \sum_{l=1}^{n_{t,i}} R_{t,i,l,j} - \frac{C_{t,j}}{I}\right), 0\right\} \tag{1}$$

Rewards and Value Function. Rewards of S_t means the income or cost when choosing an action in a given state at time t. Value function of S_t denotes the supremum over all policies of the expected total rewards from decision epoch t onwards [16].

Rewards. The rewards gained by the SaaS cloud can be defined in many ways in accordance with the engineering requirements in business. Whatever the reward formula, they all can be applied into our algorithm. In this work, we focus on the performance benefit, energy consumption, as well as penalty for unfinished tasks.

The performance benefit is in proportion to throughputs and has an inverse relationship with the response time [14]. At each time epoch, if task l of type i is handled, throughput equals the size of the task, measured in MIPS. Response time T can be calculated as Eq. (2). The performance benefit is in proportion to throughput and in inverse proportion to response time. Performance benefit of n_t tasks in the period of time $[t, t+\tau]$ can be expressed as Eq. (3), where $n_{t,i}$ means the number of type-i tasks, and σ is a constant coefficient.

$$T_{t,i,l,j} = \frac{Q_{t,i,j} + R_{t,i,l,j}}{C_{i,j}/I} \tag{2}$$

$$BR_t = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{l=1}^{n_{t,i}} \sigma\left(\frac{R_{t,i,l,j}}{T_{t,i,l,j}}\right) \tag{3}$$

$$P = P_{idle} + \mu\left(P_{busy} - P_{idle}\right) \tag{4}$$

In addition, power consumption of a server grows linearly with the growth of the CPU utilization from the idle to fully utilized state, as is found in [5]. It can be expressed in Eq. (4), where P is the estimated power consumption of one node. P_{idle} and P_{busy} respectively represent the power consumed when the server is idle and fully utilized. μ is a parameter in proportion to the CPU utilization. Worth to mention, there may be some other forms of expressions for the assessment of power consumption, and they all can be applied to this algorithm after slight alterations to VFA. In SaaS, when all VM queues of a server are empty, the server may sleep or hibernate, and then P_{idle} of this server equals 0. Otherwise, P_{idle} is a fixed value and does not change with the number of VMs at work in the server. As a result, VM consolidation, which refers to aggregating VMs on minimal physical nodes, provides a good way to save energy. Power consumption of the jth server at time t, $P_{t,j}$ can be calculated by Eq. (4). μ can be indirectly but easily deduced from matrix Q_t. The overall energy consumption during the period of time $[t, t+\tau]$ can be calculated as $EC_t = \sum_{j=1}^{J} P_{t,j}\tau$.

Finally, the penalty of tasks, which cannot be handled within their deadlines, should be subtracted. Assume there are $n_{t,fail}$ tasks unfinished during the period of epoch t, then the relative penalty can be denoted as Eq. (5), where δ can be both a constant or a function changing with time or throughput.

$$PE_t = n_{t,fail} \cdot \delta \tag{5}$$

To sum up, the overall rewards in the period of $[t, t+\tau]$ can be calculated as $R_t = BR_t - EC_t - PE_t$.

Objective Value Function. Let $x_t^{\pi}(S_t)$ denotes the decision made in state S_t under policy π. $\pi = (x_0, x_1, x_2, \dots)$ specifies a series of decisions made at one time slot. Then our objective is to find the best policy $\pi^* \in \Pi$ with the largest expected total discounted rewards over the infinite horizon [16]:

$$V^{\pi^*} = \max \mathbb{E}^{\pi} \left\{ \lim_{N \to \infty} \sum_{t=1}^{T} \gamma^{t-1} R_t(S_t, x_t^{\pi}(S_t)) \right\}. \tag{6}$$

$\gamma \in [0, 1)$ is a discount factor, which measures the value at time t of one unit reward received at time $t+\tau$ [16]. Therefore, one unit of reward received t periods later, only has the present value of γ^t and it is discounted. Equation (6) can also be expressed by the recursive Bellman Equations:

$$V_t(S_t) = \max_{x_t \in X_t} \left\{ R_t(S_t, x_t) + \gamma \mathbb{E}\{V_{t+1}(S_{t+1}) | S_t\} \right\}. \tag{7}$$

Altogether, if different types of tasks are assigned to as least number of servers as possible, there would be more servers that can hibernate, and then the overall P_{idle} will be low. However, in that case, the queue length of each VM at work will be long, which leads to high response delay and low profit of performance.

Algorithm 1. Outline of H-ADP algorithm

INPUT: Sample path ω, server available states Sa, CPU processing capacity of servers C, an iteration number T, feature basis functions $\phi_f(S)$, a discounted factor λ.
OUTPUT: Recursively estimated parameter θ_T.

1: Initialize θ_1, $\overline{V}(S)$ and$\phi_f(S)$ for all states.
2: Choose an initial state S_1.
3: **for** $t = 1, 2, \ldots, T$ **do**
4: **if** $t > 1$ **then**
5: Derive $\phi_f(S)$ from Q_t.
6: Calculate $\overline{V}_t^x(S_{t-1}^x|\theta_{t-1})$ by Eq. (9).
7: **end if**
8: Solve Eq. (8), and let x_t' the value of x_t that solve the maximization problem.
9: Change PDSV S_t^x by substituting x_t' into $S_t^x = S_{t+1} = S^M(S_t, x_t, Ex_{t+1})$.
10: Compute $\hat{v}_t(S_t)$ using Eq. (8).
11: Update θ_t using Eq. (11).
12: Choose ω_{t+1} and update S_{t+1} with Eq. (1).
13: **end for**
14: return θ_T.

Accordingly, to jointly arbitrate the tradeoff between performance and energy efficiency is a daunting and arduous work especially when the number of servers is large. Besides, exogenous probability distributions of state R_t and Sa_t may not be known beforehand in practice, so the expectation in Eq. (7) cannot be calculated directly. Thus, we put forward H-ADP algorithm in the next section.

3 Heuristic ADP Algorithm

In this section, we first introduce the framework of H-ADP, as is shown in Algorithm 1. Then the VFA approach by virtue of basis functions is illustrated. At last, we elaborate the candidate heuristic (C-H) method.

3.1 Outline of the Algorithm

Firstly, in order to take advantage the time-sequenced sample data path ω_t, which represents exogenous values at time t, the algorithm should step forward in time. In classical dynamic programming, it proceeds by stepping backward in time, and Eq. (7) has to be solved for each state S_t, such as in MDP. But the states are infinite in this paper, and an exhaustive algorithm does not work. Thus, we first extract features (a term widely used in the field of artificial intelligence), and construct an approximate value function to appraise the value of all states. We then propose a heuristic method to find the global optimal action in line 8 of Algorithm 1 under each sample. These are the two important steps to simplify the computation, which are separately illustrated in Sects. 3.2 and 3.3.

Secondly, to conveniently compute the expectation in Eq. (7), we introduce concepts of post-decision states variables (PDSVs) S_t^x. S_t^x means the system state at time t immediately after making decision x, but before time $t+1$. After a decision is made in line 8, then PDSV is equivalent to the state value at the next time slot, $S_t^x = S_{t+1}$. With PDSVs, the hard-to-calculate expectation in Eq. (7) can be eliminated, and Bellman equations can be rewritten as Eq. (8).

$\hat{v}_t(S_t)$ represents the sample values at time t, and $\overline{V}^x_{t+1}(S^x_t)$ denotes approximate post-decision value which equals the state value at the next time epoch. $\overline{V}^x_{t+1}(S^x_t)$ might be captured in the parameter form using the basis functions $\phi_f(S)$. $f \in F$, where f is a feature, and $\phi_f(S)$ is a vector of feature values that can be calculated by extracting feature information from the state Q_t. The approximate value function $\overline{V}^x_t(S^x_{t-1})$ might be rewritten as Eq. (9), where θ is an $|F|$ dimensional vector. Specific definitions of $\phi_f(S)$ in SaaS cloud are described in Sect. 3.2.

$$\hat{v}_t(S_t) = \max_{x_t \in X_t} \{R_t(S_t, x_t) + \gamma \overline{V}^x_{t+1}(S^x_t)\} \tag{8}$$

$$\overline{V}^x_t(S^x_{t-1}|\theta) = \sum_{f \in F} \theta_t \phi_f(S^x_{t-1}) \tag{9}$$

Thirdly, we use a stochastic gradient updating strategy, which stems from Machine Learning theory, to progressively train vector θ following ω_t. Vector θ represents parameters for estimating feature values. We aim at finding the most suitable θ^* that produces the minimum expected squared error (MESE) between $\hat{v}_t(S_t)$ and $\overline{V}^x_t(S^x_{t-1}|\theta)$, as is illustrated in Eq. (10). And θ can be updated step by step to approach θ^*, as is shown in Eq. (11). α_{t-1} means step size, and ∇ denotes the Nabla Operator for gradient calculation.

$$\theta^* = \arg\min_{\theta} \mathbb{E}\left\{ \frac{(\hat{v}_t - \overline{V}^x_t(S^x_{t-1}|\theta))^2}{2} \right\} \tag{10}$$

$$\begin{aligned} \theta_t &= \theta_{t-1} - \alpha_{t-1}\big(\hat{v}_t(S_t) - \overline{V}^x_t(S^x_{t-1}|\theta_{t-1})\big)\nabla_\theta \overline{V}^x_t(S^x_{t-1}|\theta_{t-1}) \\ &= \theta_{t-1} - \alpha_{t-1}\big(\hat{v}_t(S_t) - \overline{V}^x_t(S^x_{t-1}|\theta_{t-1})\big)\phi(S^x_{t-1}) \end{aligned} \tag{11}$$

Fourthly, we should care more about the step size α_{t-1}. To choose the proper step size is really an art form which is based on experience and specific problem structures. A step size that is too large can produce unstable behaviors. However, if it is small, the procedure of regression may be too slow. In general, it should satisfy the constraints in Eq. (12) [15]. Several ways of setting the step sizes may be possible, and we will carefully design it in Sect. 4.

$$\begin{cases} \sum_{t=1}^{\infty} \alpha_{t-1} = \infty \\ \sum_{t=1}^{\infty} (\alpha_{t-1})^2 < \infty \\ \alpha_{t-1} \geq 0 \\ t = 1, 2, \ldots \end{cases} \tag{12}$$

Algorithm 2. Candidate-Heuristic algorithm

INPUT: A request Runit, estimated parameter θ, queue state of VMs Q, CPU processing capacity C, number of candidate Nc, server available states Sa, a discounted factor λ.
OUTPUT: The optimal server index p of the target VM.

1: $i = Runit.Type$.
2: Create and derive a J-dimensional vector Nen, and Nen_j represents the number of VM engagement excluding type i on server j.
3: In server comparisons, set the weighting orders from high to low as C_j, Nen_j and $Q_{t,i,j}$.
4: Find the indexes of top Nc healthy nodes giving priority to severs with the highest C_j.
5: In server comparisons, set the weighting orders from high to low as $Q_{t,i,j}$, C_j and Nen_j.
6: Find the indexes of top Nc healthy nodes giving priority to VMs with the least $Q_{t,i,j}$.
7: In server comparisons, set the weighting orders from high to low as Nen_j, $Q_{t,i,j}$ and C_j.
8: Find the indexes of top Nc healthy nodes giving priority to servers that more VMs are engaged (with the highest Nen_j).
9: Merge the indexes of $3Nc$ candidates by removing duplication items.
10: Rank the rewards of these candidates with Eq. (8) and (9), and choose the best indexed by p.
11: Return p.

3.2 Value Function Approximation

As is illustrated in Eq. (9), approximated value function of each state can be derived with live updated θ_t and $\phi_f(S)$, which are tailored for specific scenarios. In the task scheduling scenario, rewards during one time epoch, are mainly comprised of two parts, performance and energy. Performance benefit for the type-i tasks is in proportion to the sum of available CPU processing capacity on all type-i VMs ($Avai\_CPU(i)$), and is in negative proportion to the average queue length of type-i ($Ave\_Q(i)$). Energy consumption for the type-i tasks is also in proportion to $Avai\_CPU(i)$, and in proportion to the number of all physical machines that are at work (N_PM_on). So if there are I type of tasks, $2I + 1$ features are needed. Thus $\phi_f(S)$ can be expressed as Eq. (13). Each element in $\phi_f(S)$ at each time t can be easily derived from endogenous state Q_t.

$$\phi_f(S) = \begin{pmatrix} Avai\_CPU(1) \\ Avai\_CPU(2) \\ \dots \\ Avai\_CPU(I) \\ Ave\_Q(1) \\ Ave\_Q(2) \\ \dots \\ Ave\_Q(I) \\ N\_PM\_on \end{pmatrix} \tag{13}$$

3.3 Candidate Heuristic Algorithm

Generally, there are two ways of finding the optimum decision in line 8 of Algorithm 1. First, we can compute the complicated rewards of all actions, and then rank and find the best. However, the action space is extremely large, so this method is too intricate to implement in practice. The other method is to equivalently transform it into classical resource allocation problems, using linear programming (LP) approach. The number of variables is equal to that of servers,

and there are thousands of servers in a large-scale SaaS cloud. Thus, it is also infeasible to solve this LP in an acceptable time. Accordingly, an elegant and ingenious method tailored for the task scheduling problem is in urgent need.

After analyzing the optimal choices of servers for tasks, target VMs usually possess three characteristics: (1) with high processing capacity to diminish response time and increase profits, (2) with low queue length for the same reason, (3) on servers where more different type of VMs are at work. Here are illustrations for (3): If three types of requests are assigned to VMs on three distinct servers, all servers will be on. Yet, if they are allocated to VMs on only one server, the other two can hibernate. It is similar to the reason of VM consolidations.

Therefore, in order to find the optimal nodes, there are mainly three strategies, each focusing on one characteristic mentioned above. In each strategy, we can choose Nc candidates for comparisons. Then we merge the indexes of $3Nc$ candidates by removing duplication items. Nc should be set according to specific circumstances. If Nc is large, there will be more chances of finding the best candidate, but higher computation complexity, and vice versa. Usually, $Nc = 5$ is enough. At last, we compute and rank the rewards of all these candidates in order to choose the best. In this approach, for every task, we reduce calculating the complex rewards of thousands of nodes into computing rewards of only a few candidates. As the number of requests accumulates with time, the computation complexity can be greatly reduced. We compute the state values with Eqs. (8) and (9), simultaneously considering current and future circumstances.

The whole algorithm is demonstrated in Algorithm 2. It is worth noting that, in line 4, 6 and 8, only "healthy" nodes (indicated by $Sa_{t,j} = 1$), can be listed and selected. Besides, we do the task scheduling in consideration of the deadline of each request, and if all VMs are unable to finish a task before its deadline, the request will be marked and rejected. Of course, unhandled tasks will bring additional losses, which can be calculated in Eq. (5).

4 Qos Evaluation

In this section, we conduct simulations in Matlab 2012a, on a PC with i5 processor at 3.5 GHz and memory size of 4GB. After steps of approximations in Sect. 3, complicated scheduling procedure of each task is simplified into comparing metrics of only a few candidates with elementary operations. Computation complexity is reduced from $O(J^2)$ to $O(3Nc)$, and $J = 10000, Nc = 5$ in this work. It is obvious that scheduling speed is no more a problem with modern computers. Due to space limit, demonstration for scheduling speed is omitted here. We focus on evaluating scheduling quality of the "oversimplified" H-ADP. We make evaluations based on both random synthetic workloads and Google trace-logs of real-world data. Due to the large scale of servers, many classical multi-objective algorithms are unavailable. Thus, we conduct comparisons of our approach with two commonly used algorithms in task scheduling of data centers, and they are load-balancing (L-B) and randomized-selection (R-S) algorithms.

4.1 Simulations on Random Workloads

In order to testify the generality and versatility of H-ADP algorithm, we conduct simulations under various input parameters in high throughputs.

Rules of Generating Main Input Parameters.

Number of servers: Due to characteristics of large-scale SaaS cloud, we set the number of nodes $J = 10000$ throughout this paper.

Task type: In most cases, the task type I is an integer not more than 10.

Task count: The number of coming tasks per unit of time is assumed to conform to Poisson distribution with a mathematical expectation $\lambda \in [10^3, 5 \times 10^3]$. Then the total number of tasks arriving during time t is λt.

Task size: The task sizes of all types are assumed to conform to exponential distribution, each with the expectation $ER \in [5 \times 10^4, 10^5]$ MI.

Task deadline: The deadline of all types of tasks is designated as $d = fb + v$ [19], where fb is a fixed base value, and v conforms to exponential or uniform distribution. We may set $fb = 15$, and set $E(v) = 20$.

Server available state: According to [17], we assume that the mean time to failure (MTTF) of each server conforms to Weibull distribution. We set the shape parameter as 0.8, and scale parameter as 500. In [17], it also argues that repair time is better modeled by a lognormal distribution. Thus, in this simulation, we assume the mean and standard deviation of the variable's natural logarithm respectively equal 4 and 0.9. Thus, as time goes by, the available state of each server may alternatively change along with its MTTF and repair time.

Server capacity: In this work, as in most data centers, we assume that servers are divided into several categories of servers, and servers of the same category possess the same processing capacity. Then the capacity of each server can be chosen from a set of configurations.

Step size: Three methods [15] of setting the step size are used in this paper. (1) In a constant rule, α_{t-1} takes a fixed value all the time. (2) In a harmonic way, α_{t-1} is large at the beginning, but gets smaller with the increase of t. Step size $\alpha_{t-1} = b/(b + t)$ at time t, where b is a fixed base value. (3) In a search-then-converge (STC) learning rule, it produces delayed learning compared with harmonic step size. Step size can be calculated as $\alpha_{t-1} = \alpha_0(\frac{a}{t} + b))/(\frac{a}{t} + b + t^{\varphi})$. Under the constant rule, we set the step size as 0.01, simultaneously ensuring the convergence rate and stability of the algorithm. Under the last two rules, we can adjust parameters along with arriving tasks, and if we get divergent results, it demonstrates that the step size is so large that we should adjust parameters to diminish the step size, and vice versa.

A Typical Simulation and Analysis. In our first experiment, we set $\lambda = 3 \times 10^3$, $I = 6$ and $ER = 8 \times 10^3$ MI. The corresponding step sizes changing with iterations under three rules are depicted in Fig. 2. Under each rule of step sizes, we can recursively get the convergence of all the 13 elements in vector θ. Random examples, $\theta(Ave\_CPU(1))$, $\theta(Ave\_Q(2))$ and $\theta(N\_PM\_on)$, are separately plotted in Figs. 3, 4 and 5. In the constant step-size rule, as the step size

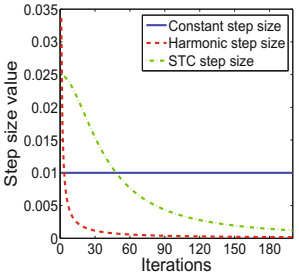

Fig. 2. Evolvement of three step sizes

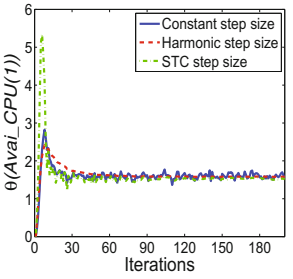

Fig. 3. Convergence of $\theta(Avai\_CPU(1))$

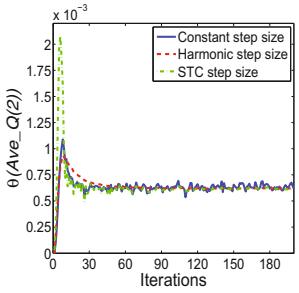

Fig. 4. Convergence of $\theta(Ave\_Q(2))$

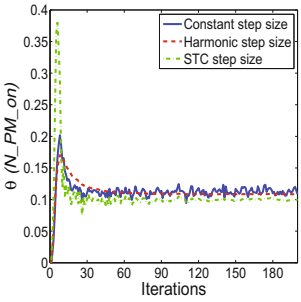

Fig. 5. Convergence of θ (N_PM_on)

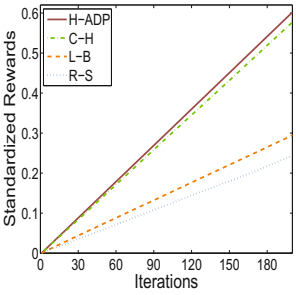

Fig. 6. Rewards changing with iterations

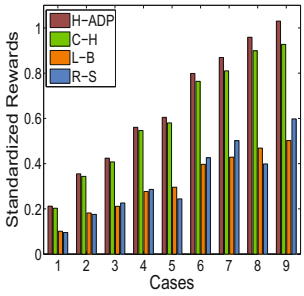

Fig. 7. Overall rewards in all cases

does not diminish with iterations, all the curves may slightly oscillate in the end. Meanwhile, in the harmonic and STC step-size rules, the curves are steep at the beginning and the peaks are pretty high especially under the STC step size, but both curves are relatively flat in the end.

What can be summarized from these figures are as follows: (1) All parameters (elements) in vector θ experience similar evolving processes; (2) Our approach can quickly extract information from raw data, and only after 30 iterations, all the curves oscillate around relatively fixed values; (3) Each parameter in θ respectively converges into consistent values, despite of the different processes under three different step sizes; (4) The convergence results of each parameter under three step sizes are slightly different.

Under different step-size rules, each element converges into similar results, which demonstrates the validity and correctness of our algorithm. Besides, due to the different convergent values under three step-size rules, we always set the final result of $\overline{\theta}$ as the average value in practice. In this simulation, $\overline{\theta}$ is a 13-dimentional vector expressed in Eq. (14). All elements are different due to various incoming workloads of each task type.

$$\overline{\theta} = (1.54292, 1.65964, 1.76832, 1.64586, 1.45819, 1.69651, 0.00057, 0.00062, 0.00066, 0.00059, 0.00058, 0.00046, 0.10530)^T \quad (14)$$

Utilizing the trained vector $\overline{\theta}$, we can conduct H-ADP algorithm for task scheduling. Meanwhile, the C-H algorithm can be used standalone in scheduling tasks as a kind of myopic or greedy algorithm. There are also two commonly used algorithms, load-balancing and randomized-selection. For the same incoming workloads, we simultaneously run four algorithms. Four overall rewards changing with iterations are respectively depicted in Fig. 6.

We can clearly see that all rewards are approximately in linear growth over time but with different slopes. Rewards of H-ADP and C-H are remarkably higher than L-B and R-S at each time epoch. H-ADP is in pursuit of long-term profits, and it takes into consideration of outcomes for both current and future decisions. Meanwhile, the C-H algorithm makes decisions only in accordance with immediate circumstances. Therefore, the overall reward of H-ADP in the long term is superior to that of C-H, and the gap between them cumulates with time. For a SaaS cloud that is in service day after night, the incremental benefit of H-ADP over the other three approaches will be quite substantial.

Comparative Experiments and Analysis. Next, we do simulations under different λ that increases from 1000 to 5000 with steps of 500. Meanwhile, we randomly set the number of task types and expected size of requests in their respective ranges mentioned above in this section. Thus, there are 9 cases altogether, and the main parameters are depicted in Table 1.

Table 1. Workload parameters in all cases

Case	λ	I	ER (MI)
1	10^3	5	10^4
2	1.5×10^3	3	8×10^3
3	2×10^3	4	8×10^3
4	2.5×10^3	10	9×10^3
5	3×10^3	6	7×10^3
6	3.5×10^3	7	7.5×10^3
7	4×10^3	5	6.5×10^3
8	4.5×10^3	9	8.5×10^3
9	5×10^3	4	5.5×10^3

In each case, we conduct H-ADP algorithm under three step sizes, and get similar convergence curves of all elements in θ as Figs. 3, 4 and 5. Besides, all the four algorithms possess similar reward changing curves that increase with iterations as in Fig. 6. As space is limited, both θ and reward curves changing with iterations are omitted here. We only care about the total rewards after 200 iterations. Figure 7 shows that the overall rewards of the four approaches keep an ascending trend with the increase of λ (task count per unit of time). The comparison results and the causes are in accordance with the previous experiment analyzed above. A group of experiments demonstrates that our approach is superior in all circumstances. We can also discover that with the increase of λ, the gaps between the rewards of H-ADP and C-H are becoming more apparent. Meanwhile, the rewards of L-B and the rewards of R-S are alternately higher than each other, and which is higher depends on specific random data of R_t and Sa_t in each case.

4.2 Evaluation Based on Google Trace-Logs

The group of simulations above demonstrates the Qos improvement and strong applicability of our approach in random synthetic workloads. In order to verify the feasibility of H-ADP in practical use, we further conduct experiments

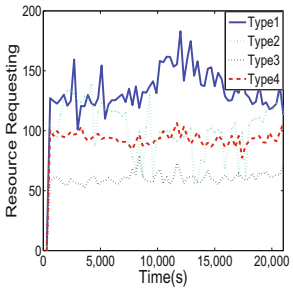

Fig. 8. Resource requesting statistics for Google trace-logs

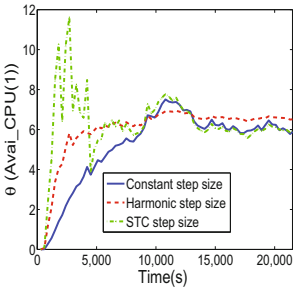

Fig. 9. Convergence of $\theta(Ave\_CPU(1))$ with Google trace-logs

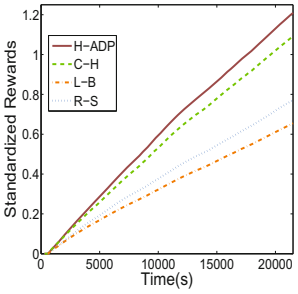

Fig. 10. Rewards changing with time with Google trace-logs

with Google cloud trace-logs [6]. The trace-logs describe workload information of 25 million tasks that span 29 days. It is particularly difficult and not essential to utilize all the log data, so we select the data set of *TraceVersion1*, which recorded a 7-h period of traces containing 4 type of 3,535,031 tasks. Each task is marked with arriving time, type, normalized resource consuming and so on. Total standardized resource requesting of each type of tasks at each time epoch are depicted in Fig. 8. We can see clearly that the coming tasks of each type are not evenly distributed and accompanied with bursty requests. Therefore, it is hard to discover any distribution law of the trace-logs, and we need to take advantage of Machine Learning theory.

With H-ADP, we can also recursively get the convergence curves of all the 9 $(2 \times 4 + 1 = 9)$ elements in vector θ under three rules of step sizes. As real workloads are disordered and unpredictable, the curves fluctuate more acutely but they all converge in the end. Convergence for $\theta(Ave\_CPU(1))$ with three step sizes is plotted as an example in Fig. 9. We can then calculate the average $\overline{\theta}$ for H-ADP and do algorithm comparisons.

Our approach does not rely on pre-knowing the probability distribution function (PDF) of requests, and H-ADP can learn recursively from unpredictable and bursty requests, but myopic algorithms can not. Therefore, H-ADP is remarkably superior to C-H in terms of rewards as time accumulates, as is demonstrated in Fig. 10. Again, these two approaches significantly outperform L-B and R-S.

5 Conclusions

In this paper, we tackle the real-time task scheduling problems in SaaS cloud. We first construct an SDP problem and analyze the ingredients. With reference to Machine Learning theory, we put forward an ADP algorithm combined with candidate-heuristic method. Comprehensive experiments and comparisons are conducted to evaluate the algorithm. Results demonstrate that the proposed work provides an elegant and effective approach to handle complex scheduling problems in large-scale heterogeneous SaaS.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (No. 61472199 and No. 61370132).

References

1. Alahmadi, A., Che, D., Khaleel, M., Zhu, M.M., Ghodous, P.: An innovative energy-aware cloud task scheduling framework. In: 2015 IEEE 8th International Conference on Cloud Computing, pp. 493–500. IEEE (2015)
2. Barroso, L.A., Clidaras, J., Hölzle, U.: The datacenter as a computer: an introduction to the design of warehouse-scale machines. Synth. Lect. Comput. Archit. **8**(3), 1–154 (2013)
3. Cheng, C., Li, J., Wang, Y.: An energy-saving task scheduling strategy based on vacation queuing theory in cloud computing. Tsinghua Sci. Technol. **20**(1), 28–39 (2015)
4. Egwutuoha, I.P., Cheny, S., Levy, D., Selic, B., Calvo, R.: Energy efficient fault tolerance for high performance computing (HPC) in the cloud. In: 2013 IEEE Sixth International Conference on Cloud Computing, pp. 762–769. IEEE (2013)
5. Fan, X., Weber, W.D., Barroso, L.A.: Power provisioning for a warehouse-sized computer. In: ACM SIGARCH Computer Architecture News, vol. 35, pp. 13–23. ACM (2007)
6. Google: Cloud trace-logs. code.google.com/p/googleclusterdata/wiki
7. Hosseinimotlagh, S., Khunjush, F., Hosseinimotlagh, S.: A cooperative two-tier energy-aware scheduling for real-time tasks in computing clouds. In: 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 178–182. IEEE (2014)
8. IBM: Predictive maintenance (2015). www-01.ibm.com/software/analytics/solutions/operational-analytics/predictive-maintenance/
9. Kumar, A., Shang, L., Peh, L.S., Jha, N.K.: System-level dynamic thermal management for high-performance microprocessors. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **27**(1), 96–108 (2008)
10. Liu, F., Zhou, Z., Jin, H., Li, B., Li, B., Jiang, H.: On arbitrating the power-performance tradeoff in saas clouds. IEEE Trans. Parallel Distrib. Syst. **25**(10), 2648–2658 (2014)
11. Mao, Y., Xu, Z., Ping, P., Wang, L.: Delay-aware associate tasks scheduling in the cloud computing. In: 2015 IEEE Fifth International Conference on Big Data and Cloud Computing (BDCloud), pp. 104–109. IEEE (2015)
12. Nakamura, H., Matsuda, H., Akazawa, F., Shiraga, M.: U.S. Patent No. 8,195,985. U.S. Patent and Trademark Office, Washington, DC (2012)
13. O'Brien, J.: Datacenter facilities maintenance (2014). www.datacenterjournal.com/datacenter-facilities-maintenance-time-change-culture
14. Peterson, L.L., Davie, B.S.: Computer Networks: A Systems Approach. Elsevier, Amsterdam (2007)
15. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality, vol. 703. Wiley, Hoboken (2007)
16. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, Hoboken (2014)
17. Schroeder, B., Gibson, G.: A large-scale study of failures in high-performance computing systems. IEEE Trans. Dependable Secure Comput. **7**(4), 337–350 (2010)

18. Tchana, A., Broto, L., Hagimont, D.: Approaches to cloud computing fault tolerance. In: 2012 International Conference on Computer, Information and Telecommunication Systems (CITS), pp. 1–6. IEEE (2012)
19. Wang, J., Bao, W., Zhu, X., Yang, L.T., Xiang, Y.: Festal: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds. IEEE Trans. Comput. **64**(9), 2545–2558 (2015)
20. Wikipedia: Lm-sensors. en.wikipedia.org/wiki/Lm_sensors
21. Zhu, X., Yang, L.T., Chen, H., Wang, J., Yin, S., Liu, X.: Real-time tasks oriented energy-aware scheduling in virtualized clouds. IEEE Trans. Cloud Comput. **2**(2), 168–180 (2014)