A Reinforcement Learning Approach to Online Web System Auto-configuration

Xiangping Bu, Jia Rao, Cheng-Zhong Xu
Department of Electrical & Computer Engineering
Wayne State University, Detroit, Michigan 48202
{xpbu,jrao,czxu}@wayne.edu

Abstract

In a web system, configuration is crucial to the performance and service availability. It is a challenge, not only because of the dynamics of Internet traffic, but also the dynamic virtual machine environment the system tends to be run on. In this paper, we propose a reinforcement learning approach for autonomic configuration and reconfiguration of multi-tier web systems. It is able to adapt performance parameter settings not only to the change of workload, but also to the change of virtual machine configurations. The RL approach is enhanced with an efficient initialization policy to reduce the learning time for online decision. The approach is evaluated using TPC-W benchmark on a three-tier website hosted on a Xen-based virtual machine environment. Experiment results demonstrate that the approach can auto-configure the web system dynamically in response to the change in both workload and VM resource. It can drive the system into a near-optimal configuration setting in less than 25 trial-and-error iterations.

I. Introduction

Web systems like Apache and Tomcat applications often contain a large number of parameters to be configured when they are deployed and the parameter settings are crucial to systems performance and service availability. For example, an incorrect setting of the `MaxClient` parameter in Apache may even cap the throughput of a large-scale server. Traditionally, a web system configuration is performed manually, based on operator's experience. This is a non-trivial and error-prone task. Recent human factor studies on root causes of Internet service outages revealed that more than 50% was due to system misconfiguration caused by operator mistakes [6].

The configuration challenge is due to a number of reasons. First is the increasing system scale and complexity that introduce more and more configurable parameters to a level beyond the capacity of an average-skilled operator. For example, the latest version of an Apache server has more than 240 configurable parameters that relate to performance, support files, server structure, and required modules. Likewise, a Tomcat server contains more than a hundred parameters to set for different running environments. In a multi-component system, the interaction between the components makes performance tuning of the parameters even harder. In a multi-tier web system, a misconfiguration in one tier may cause misconfigurations in the others. Performance optimization of individual component does not necessarily lead to overall system perfomance improvement [2]. Therefore, to find an appropriate configuration, the operator must develop adequate knowledge about the system, get familiar with each of its configurable parameter, and run numerous trail-and-error tests.

Another challenge in configuration comes from the dynamic trait of web systems. On the Internet, the systems should be able to accommodate a wide variety of service demands and frequent component in both software and hardware. Chung et al. [2] showed that in web system no single universal configuration is good for all workloads. Zheng et al. [19] demonstrated that in a cluster-based Internet service, when the application server tier got updated, such as adding or reducing the number of application servers, the system configuration should be modified to adjust to this evolution.

Moreover, virtual machine technology and related utility and cloud computing models pose new challenges in web system configuration. VM technology enables multiple virtualized logical machines to share hardware resources on the same physical machine. This technology facilitates on-demand hardware resource reallocation [11] and service migration [3]. Next-generation enterprise data centers will be designed in a way that all hardware resources are pooled into a common shared infrastructure; applications share these remote resources on demand [10], [16]. It is desirable that the resources allocated to each VM should be adjusted dynamically for the provisioning of QoS guarantees and meanwhile maximizing resource utilization [8]. This dynamic resource allocation requirement adds one more dimension of challenge to the configuration of web systems

hosted in virtual machines. In particular, the configuration needs to be carried out on-line and automatically.

There were many past studies devoted to autonomic configuration of web systems; see [17], [18], [2], [19], [5] for examples. Most of them focused on performance parameters tuning for dynamic workload in a static environment. Xi et al. [17] and Zhang [18] used hill-climbing algorithms to search the best settings for a small number of key parameters in application servers. Chung and Hollingsworth [2] and Zheng et al. [19] suggested to construct performance functions of configurable parameters in a direct approach so as to tune the parameters by optimizing the functions. Because of the time complexity of their optimization approaches, they are not applicable to online setting of the parameters in VM-based dynamic platforms. In [5], Liu et al. proposed a fuzzy control approach to adaptively reconfigure Apache Web server to optimize response time. It was targeted at online tuning in responses to changing workload. However, because of the inherent complexity of the control approach, it was limited to the tuning of single `MaxClient` parameter.

In this paper, we propose a reinforcement learning approach, namely RAC, for automatic configuration of multi-tier web systems in VM-based dynamic environments. Reinforcement learning is a process of learning from interactions. For a web system, its possible configurations form a state space. We define actions as reconfiguration of the parameters. Reinforcement learning is intended to determine appropriate actions at each state to maximize the long-term reward. Recent studies showed the feasibility of RL approaches in resource allocation [13], [15], power management [14], job scheduling in grid [1] and self-optimizing memory controller [4]. To best of our knowledge, the RWC approach should be the first one in the application of the RL principle to automatic configuration of web systems.

The RAC approach has the following features: (1) It is applicable to multi-tier web systems where each tier contains more than one key parameters to configure; (2) It is able to adapt system configuration to the change of workload in VM-based dynamic environments where resource allocated to the system may change over time; (3) It is able to support online auto-configuration.

Online configuration has a time efficiency requirement, which renders conventional RL approaches impractical. To reduce the initial learning overhead, We equip the RL algorithm with efficient heuristic initialization policies. We developed a prototype configuration management agent, based on the RAC approach. The agent is non-intrusive in the sense that it requires no change in either server or client sides. All the information needed is application level performance such as throughput and response time. We experimented with the RAC agent for a three-tier TPC-W website/benchmark on a Xen-based virtual machine environment. Experiment results showed that the RAC agent can auto-configure the web system dynamically in response to the change in both workload and VM resource. It can drive the system into a near-optimal configuration setting in less than 25 trial-and-error iterations.

TABLE I. Tuning parameters

Configuration parameter	Candidate values	Default
MaxClients(web server)	from 50 to 600	150
Keepalive timeout	from 1 to 21	15
MinSpareServers	from 5 to 85	5
MaxSpareServers	from 15 to 95	15
MaxThreads(app server)	from 50 to 600	200
Session timeout	from 1 to 21	30
minSpareThreads	from 5 to 85	5
maxSpareThreads	from 15 to 95	50

The rest of this paper is organized as follows. Section II presents scenarios to show the challenges in configuration management in dynamic environments. Section III presents basic idea of the RL approach and its application in auto-configuration. Enhancement of the approach with initialization policies is given in Section IV. Section V gives the experimental results. Related work is discussed in Section VI. Section VII concludes the paper with remarks on limitations of the approach and possible future work.

II. Challenges in Website Configuration

A. Match Configuration to Workload

Application level performance of a web system heavily depends on the characteristics of the incoming workload. Different types of workloads may require different amounts and different types of resources. Application configuration must match the need of current workloads to achieve a good performance.

For instance, `MaxClients` is one of the key performance parameters in Apache, which sets the maximum number of requests to be served simultaneously. Setting it to a too small number would lead to low resource utilization; in contrast, a high value may drive the system into an overloaded state. With limited resource, how to set this parameter should be determined by the requests resource consumption and their arrival rates. Configurations of this parameter for resource intensive workload may lead to poor performance under lightly loaded conditions.

To investigate the effect of configuration on performance, we conducted experiments on a three-tier Apache/Tomcat/MySQL website. Recall Apache and Tomcat each has more than a hundred configuration parameters. Based on recent reports of industry practices and our own test results, we selected eight most performance relevant run-time configurable parameters from different tiers, as shown in Table I. For simplicity in testing, we assume the default settings for the MySQL parameters.

We tested the performance performance using TPC-W benchmark. TPC-W benchmark defines three types of workload: ordering, shopping, and browsing, representing three different traffic mixes. It is expected that each

Fig. 1. Performance under configurations tuned for different workloads.

workload has its preferred configuration, under which the system would yield the lowest average response time. Figure 1 shows the system performance for different workloads under the three best configurations (out of our test cases). From the figure, we observe that there is no single configuration suitable for all kinds of workloads. In particular, the best configuration for shopping or browsing would yield extremely poor performance under ordering workload.

B. Match Configuration to Dynamic VM Environments

For a web system hosted on VMs, its capacity is capped by the VM resources. it tends to change with reconfiguration of the VM (for fault tolerance, service migration, and other purposes). The change of the VM configuration renders the previous web system configuration obsolete and hence calls for reconfiguration online. Such reconfigurations are error prone and sometimes even counter-intuitive.

In this following, we still use `MaxClients` parameter to show the challenges due to VM resource change. In this experiment, we kept a constant workload and dynamically changed the VM resource allocated to the application and database servers. We defined three levels of resource provisioning: Level-1 (4 virtual CPUs and 4GB memory), Level-2 (3 virtual CPUs and 3GB memory), and Level-3 (2 virtual CPUs and 2GB memory). Figure 2. shows the impact of `MaxClients` settings under different VM configurations. From the figure, we can see that each platform has each own preferred `MaxClients` setting leading to the minimum response time. We notice that as the capacity of the machine increases, the optimal value of `MaxClients` actually goes down instead of going up as we initially expected. The main reason for this counter-intuitive finding is that with the VM becoming more and more powerful, it can complete a request in a shorter time. As a result, the number of concurrent requests will decrease and there is no need for a large `MaxClients` number. Moreover, the measured response time included request queuing time and its processing time. The `MaxClients` parameter controls the balance between these two factors. A large value would reduce the queueing time, but at the cost of processing time because of the increased level of concurrency. The tradeoff between the queuing time and processing time is heavily dependent on the concurrent workload and hardware resource.

`MaxClient` aside, we tested the settings of other parameters under different VM configurations. Their effects are sometimes counter-intuitive due to the dynamic features of web systems. Figure 3 shows no single configuration is best for all platforms. In particular, the performance under Level-2 resource may even deliver better performance under Level-1 platform.

Fig. 2. Effect of `MaxClients` on performance under different VM platforms.

Fig. 3. Performance under configurations tuned for different VM platforms.

III. Reinforcement Learning Approach to Auto-Configuration

In this section, we will present an overview of our RL approach and its application to auto-configuration.

A. Parameter Selection and Auto-Configuration

Today's web systems often contain a large number of configurable parameters. Not all of them are performance relevant. For tractability of auto-configuration, we first select the most performance-critical parameters as configuration candidates. Because online reconfiguration is intended to performance improvement at the cost of its run-time overhead. Including a huge number of parameters will sharply increase the online search spaces, causing a long time delay to converge or making the system unstable. To select an appropriate tuning parameter, we have to deal with the tradeoff between how much the parameter affects the performance and how much overhead it causes during the online searching.

Even from the performance perspective, how to select the appropriate parameters for configuration is a challenge. In [19], authors used parameters dependency graph to find the performance relevant parameters and the relationship among them. Our focus in on autonomic reconfiguration in response to system variations by adjusting a selective group of parameters. Table I lists the parameters we selected and the ranges of their values for testing purposes. How to automatically select the relevant parameters is beyond the scope of this paper.

For a selective group of parameter in different tiers, we design a RL-based autonomic configuration agent for multi-tier web systems. The agent consists of three key components: performance monitor, decision maker, and configuration controller. The performance monitor passively measure the web system performance at a predefined time interval (we set it to 5 minutes in experiments), and send the information to RL-based decision maker. The only information the decision maker needs is the application level performance such as response time or throughput. It require no OS-level or hardware level information for

portability. The decision maker runs a RL algorithm and produce a state-action table, called Q-value table. A state is defined as a configuration of the selected parameters. Possible actions include increasing, decreasing their values or keeping unchanged; see the next section for details. Based on the dynamically updated Q table, the configuration controller generates the configuration policy and reconfigures the whole system if necessary.

B. RL-based Decision Making

Reinforcement learning is a process of learning through interactions with an external environment (or the web system in this paper). The reconfiguration process is typically formulated as a finite Markov decision process(MDP), which consists of a set of states and several actions for each state. During each state transition, the learning agent should receive a reward defined by a reward function $R = E[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$. The goal of the agent is to develop a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ to maximize the collected cumulative rewards based on iterative trial-and-error interactions [12].

We first cast the online automatic configuration problem as a MDP, by defining state space S, action set A, and immediate reward function $r(s, a)$.

a) State Space.: For the online auto-configuration task, we define a state as possible system configuration. For the selective group of n parameters, we represent a state by a vector in the form as:

$$s_i = (Para_1, Para_2, \cdots, Para_n).$$

b) Action Set.: We define three basic actions: $increase$, $decrease$, and $keep$ associated with each parameter. We use a vector a_i to represent an action on parameter i. Each element itself is a 3-element vector, indicating taken/not-taken (1/0) of three actions. For example, the following notation represents an increase action on parameter i.

$$a_i^{increase} = (\cdots, Para_i(1, 0, 0), Para_n(0, 0, 0))$$

c) Immediate Reward.: The immediate reward should correctly reflect the system performance. The immediate reward r at time interval t is defined as

$$r_t = SLA - perf_t,$$

where SLA is a reference time predefined in Service Level Agreement, and $perf$ is measured response time. For a given SLA, a lower response time returns a positive reward to the agent; otherwise the agent will receive a negative penalty.

d) Q Value Learning.: To learn the Q value of each state, the agent should continuously update its estimation based on the state transition and reward it receives. The temporal difference(TD) is most suitable for our work due to its two advantages: It needs no model of the environment and it updates Q values at each time step based on its estimation. Using such incremental fashion, the average Q value of an action a on state s, denoted by $Q(s, a)$, can be refined once after each immediate reward r is collected:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * [r_{t+1} + \gamma * Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)],$$

where α is a learning rate parameter that facilitates convergence to the true Q values in the presence of noisy or stochastic rewards and state transitions [12], and γ is the discount rate to guarantee the accumulated reward convergence in continuing task. Algorithm 1 presents the pseudo code of our Q value learning algorithm.

Algorithm 1 Q value Learning Algorithm

```
1: Initialize Q table
2: Initialize state s_t
3: error = 0
4: repeat
5:    for all states s do
6:       a_t = get_action(s_t) using ε − greedy policy
7:       for (step = 1;step <LIMIT;step + +) do
8:          Take action a_t Observe r and S_{t+1}
9:          Q_t = Q_t + α * (r + γ * Q_{t+1} − Q_t)
10:         error = MAX(error, |Q_t − Q_{previous−t}|)
11:         s_t = s_{t+1}, a_t = a_{t+1}
12:      end for
13:   end for
14: until error < θ
```

IV. Online Learning and Adaptation

RL algorithm explores system dynamic features and learns the configuration policy by interacting with the external environment. Even a basic RL algorithm can learn by itself during online running. A practical problem with the basic algorithm is that the number of Q values that need to explore increases exponentially with the number of attributes used in state representations [4]. When it is applied to autonomic configuration, the agent would suffer from poor performance in the initial stage, and needs a large amount of time to converge to a good state. This time complexity issue makes the online learning challenge.

A. Policy Initialization

The initial poor performance and poor scalability would limit the potential of RL algorithms for online auto-configuration. For a remedy, our RL agent assumes an external policy initialization strategy to accelerate the learning process. Briefly, it first samples the performance of a small portion of typical configurations and uses these sample data to predict the performance of other similar configurations. Based on these information, the agent runs another reinforcement learning process to generate an initial policy for the online learning procedure.

A key issue is to choose representative states for approximation. A tradeoff exists between the number of states to be considered and their coverage of the states representing

Fig. 4. Concave upward effect of `MaxClients` and regression.

the dynamics of the system. In implementation, we use a technique of *parameter grouping* to group parameters with similar characteristics together so as to reduce the state space in a coarse granularity. For example, both parameters `MaxClients` and `MaxThreads` are limited by the system capacity and often set to the same value in practice; they could be put in the same group. Likewise, both parameters `KeepAlive timeout` and `session timeout` are limited by the number of multiple connection transactions and they could be put in another group.

After having the state value of representative configurations, we use a simple but efficient method to predict the performance of pother configurations. It is based on the fact that all parameters have a concave upward effect on the performance, as revealed in [5]. Figure 4 shows the concave upward effect of a single parameter `MaxClient` on response time, observed in one of our experiments. We use a polynomial regression algorithm to predict the performance due to different settings of the parameter. Algorithm 2 gives the pseudo-code of the policy initialization algorithm.

Algorithm 2 Policy Initialization.

1: **Combine** $MaxClients$ and $MaxThreads$ as single parameter Max
2: **Combine** $KeepAlvietimeout$ and $Sessiontimeout$ as single parameter $Timeout$
3: **for all** parameters **do**
4: **Collect** data using coarse granularity
5: **Generate** performance predictor using polynomial regression
6: **Predict** absent performance values
7: **end for**

B. Online Learning

An appropriate policy initialization can avoid the initial poor performance of online learning and help the system quickly converge to a good state. It may not be accurate enough for reconfiguration decision. In the subsequent online learning, our agent keeps collecting the current system performance and retraining the Q value table at each time interval. For each retraining procedure, the agent updates the performance information for current configuration but still keep the old information for other configurations. Based on these updated performance information, it updates the Q value table so as to guarantee that most of the states be aware of the new changes in the system. After each retraining, the agent will then direct the system to the next state based on the new policy derived from the updated Q value table.

C. Adaptation to Workload and Systems Dynamics

Recall the web system hosted in a VM-based dynamic environment, there are two dynamic factors: the changing of incoming workloads and VM resource variation. As we discussed in Section II, there is no single best configuration for all kinds of workloads and all types of VM platforms. Each system scenario has its own favorite configuration and the best configuration for one situation could be a misconfiguration for the others.

To deal with the problem of poor initial performance and to accelerate the learning process when the web system changes from one scenario to another, we may construct different initialization policies for different scenarios through offline training. Algorithm 3 shows the online training and adaptation algorithm. The RL agent continues to capture the state performance, i.e. immediate reward got from this state and compares it with average value of the measurements of past n iterations. We define an initiation policy switching criteria to distinguish any performance abrupt change (i.e. policy violation). If there are k times violations happened continuously, the agent will shift to a most suitable initial policy according to the current performance. In implementation, we set n, k, and the $threshold$ as 10, 5, and 0.3, respectively.

$$pvar = \alpha * |rptime_{cur} - rptime_{aver}|/rptime_{aver},$$

$$violation = \begin{cases} 0 & \text{if } pvar \leq threshold; \\ 1 & \text{else.} \end{cases}$$

Algorithm 3 Online Learning.

1: **Initialized** Q table
2: **Initialized** state S_t
3: **for all** configuration steps **do**
4: **Collect**current performance
5: **Check** violation
6: **If** ($Num_{violations} < Threshold$) **Shifting** policy
7: **Update**Q value table using Algorithm 1
8: **Choose** a_t from S_t using $\epsilon - greedy$ policy
9: **end for**

V. Experiments Results

In this section, we evaluate the effectiveness of the RL-based auto-configuration agent on a real multi-tier web system running TPC-W benchmark. The application level performance is measured by the response times.

A. Experimental Setup

To test our RL-based agent in a dynamic environment with varying client traffic and changing resource supply, we deployed the multi-tier website in a VM-based dynamic environment. The host machine is with two Intel Xeon

quad-core CPUs and 8GB memory. A client machine with the same hardware configuration was used to simulate simultaneous customers. All devices were interconnected by a fast Ethernet.

We used Xen version 3.1 as our virtualization environment. Both the driver domain and the VMs were running CentOS 5.0 with Linux kernel 2.6.18. The TPC-W benchmark [9] was installed on two VMs, with Apache web server in the first VM and the Tomcat application server and MySQL data base server in the other VM. The auto-configuration agent resided in another VM.

We dynamically varied the client traffic to the website by switching among the ordering, browsing and shopping mixes defined by TPC-W benchmark. Only the resources available to the VM hosting the application server and data base server were changed due to the fact that the last two tiers are the bottlenecks for TPC-W benchmark and affect the selection of the values of the tuning parameter.

B. Performance of Configuration Policies

The optimization target of the RL-based agent is to find a policy that automatically reconfigures multi-tier website based on previous experiences. In online auto-configuration, the agent should be able to direct the system to a good configuration within a short period of time without incurring performance penalty due to bad settings. In this section, we will evaluate the performance of our RL-based agent during online adaptation.

We compare the online performance of our RL-based agent with other two configuration methods. The first one is to follow the recommended default configuration, whose parameter settings are listed in Table I. The other is a trial-and-error method which configures the system based on testings. In this method, the agent tuned the parameters one by one. For one parameter, the agent started from the default setting and tuned the parameter in both direction (increase and decrease) until the performance became flat or even dropped. Then, the value of the best performance in above process was selected. The configuration process ended when all the parameters were tried. The method mimics the way an administrator may use to tune the system manually. In this experiment, we dynamically changed workload and VM resources to test the adaptability of the agent. The traffic mix and the VM resource provisioning level comprised the environment of the agent and were changed randomly. Each workload and resource combination lasted for 30 iterations. Then, a new combination was generated. We deployed the RL-based agent online and tested the agent's performance by randomly picking up three consecutive combinations. The three cases were selected were: shopping workload under Level 1 resource provision, ordering workload under Level 1 resource provision, and ordering workload under Level 3 resource provision. Figure 5 shows the results.

Due to no adaptation to the dynamic environment, the static default configuration had the worst performance, whose average response times were much higher than the other two for the majority of the time. The trial-and-error method enumerated configurations started from the initial parameter value. It may be trapped with local optimal settings. From the figure, although trial-and-error can find fair configurations in a short time, its resulted performance is around 30% worse than the RL-based agent in terms of response time. The RL-based agent performed best among the three methods. It was always able to converge to a configuration within 25 interactions with the environment and the response times for these configuration were much better than other methods.

Fig. 5. Performance changes with workload and VM resources due to different auto-configuration policies.

Fig. 6. Agent performance with and without online training.

In some cases, the RL-based agent became the worst one due to policy shifting latency. For example, at the second environment transition at the 60th iteration. The system experienced five iterations poor performance before the agent noticed the environment changes. Such performance penalty can be reduced by decreasing the policy shifting threshold, which we set to 5 times violations in this experiment. However, setting the threshold to a much smaller value will make the agent too sensitive to system fluctuations. Mistaking the normal fluctuations for environmental changes, the system could become unstable by frequently shifting the initial policies. In our work, our original setting of 5 worked well and several initial iterations penalty was acceptable for the web system.

Recall that the online training and updating process will continue through the agent's lifetime. As the algorithm converges to a state, the agent will direct the system to loop at the final state but will not stop the retraining due to the requirement of adaptations to system variations. Figure 6 compares the agent's performances between with and without the online refinements. The agent without online learning converged much faster than the agent with online refinement because the former just used the prelearned policy and involved no updating. Initially, the agent with online learning suffered from the exploring new states and updating Q values whose performance even worse than the agent without online learning. However, within 20 iterations, it converged to a stable configuration whose performance was much better. This case shows that through online learning the agent can find a much better configuration than the initialized one.

C. Effect of Policy Initialization

The time RL algorithm needs to generate an optimal policy increases with the size of state space exponentially . Without an appropriate initial policy, the RL algorithm experiences a large amount of explorations before a policy can be obtained, which is not acceptable in online auto-configuration. In this work, we proposed initial policies for different specific situations to avoid the initial poor performance and to accelerate the agent's policy generation.

We used average response time as the application level performance metric. Figure 7(a) and Figure 7(b) plot the response times when the system is configured by the agent with and without the initial policy in different resource provisioning levels. In Figure 7(a), we evaluate the initial policy for Level 1 resource provision with ordering workload. And in Figure 7(b), we evaluate the initial policy for Level 2 resource provision with shopping workload.

For each evaluation, we run 30 iterations. In Figure 7(b), the agent without initial policy leads the configurations to states in which the average response time was 10 times larger than the one with initial policy. These two figures show the necessity of an external policy for the web system configuration task.

Furthermore, the experiments showed the effectiveness of the initial policies. The agents with initial policies was able to converge to a stable state within 25 iterations. More importantly, they both successfully avoided bad configurations during the process. Besides finding the optimal configuration for the web system, the RL-based agents reconfigure the system in a way that the configuration process is also optimal. We can see that after 10 iterations, both of the systems show a relatively low response time because of the selection of good configurations each iteration.

However, in a dynamic web system, it is impossible to derive sufficient environment specific initial policies for all the workload and resource combinations. In this section, we show that any one of the initial policies attained for a certain workload and resource combination can be used throughout the online learning due to the adaptation of the RL algorithm. We compared the performance of the agent using only one initial policy with the case that initial policies were changed to specific ones when environment changed. We call the only one policy used a unified policy and the policies selected at the transition of the environment a specified policy.

In the experiment, we picked up the initial policy derived from ordering workload over resource Level 1 as the unified policy. Then the unified policy was compared to the specified policies on two different situations. In Situation 1, we ran ordering workload under VM resouce Level 2 twice using its own initial policy and unified policy separately. In Situation 2, shoppingt workload was take under VM resource Level 1. Figure 7(c) and Figure 7(d) plot the response times for both policies under different situations. In both figures, initially the agent with the unified policy suffered from bad response time and large fluctuations. After some interactions, unified policy was gradually refined and was able to lead to good configurations.

Fig. 8. Performance changes with workload and VM resouce due to different RL policies.

Compared with the case with no initial policy, the agent with unified initial policy worked much better. Moreover, after a limited number of iterations, 12 iterations in Figure 7(c) and 19 iterations in Figure 7(d), the response time had been relatively low before the convergence.

The effectiveness of the unified initial policy was due to the online batch updating of the Q-value table and the characteristics of web systems. During each iteration, new collected performance information was used to retrain the Q value table. The recent collected rewards spreed the environment dynamics to all the states. Therefore, although the unified initial policy can not accurately reflect the system dynamic, the interactions between agent and the external environment could calibrate the system mapping within acceptable time. Moreover, for a web system, some extreme configurations rarely happen in practice. For example, setting the `KeepAlive timeout` a value higher than 20 is always a bad decision. Because few web pages contain so many objects requiring the TCP connection to be kept for such a long time. The unified policy can automatically mask the system from these extreme configurations and avoid huge performance penalties.

We also performed experiments to compare online adaptation performance of the three RL algorithms: the algorithm without any initial policy, with a unified policy, and with specified policies. We dynamically changed the system situations the same as the previous experiments in Section V.B. Figure 8 shows the results.

At iteration 30 and iteration 60, the system experienced a workload change and a VM resource reallocation, respectively. Our agent with specific initial policies performed very well during the online adaptation. After the workload change at iteration 30, the agent continuously observes performance violations and starts a policy change at iteration 35 when the number of continuous violations exceeded the threshold. Consequently, the response time dropped sharply after the agent got its specific initial policy. In Figure 8, we can see that, after iteration 35, the average response time reduced by more than 60% and became very close to the performance of a convergent state after iteration 46. All policy shifting and optimal searching activities were completed within 15 iterations. Furthermore, during the online learning process, the agent always kept the system performance in an acceptable level. We observed a similar good performance of our agent when the VM resource changes when system transfer from the second situation to the third one.

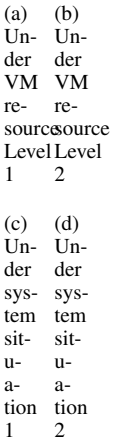

Fig. 7. Agent performance with and without specific initialization policies.

The agent with the unified policy also worked well. In Figure 8, we can see that its performance curve was close to that of the specific equipped agent except suffering a short time initial fluctuations. After each system variation, the agent could always find a good performance state within 20 iterations. Such generally equipped agent is more practical and useful in real systems. It relies on interactions with environment instead of policy shifting to continuously update the Q values table, and assumes much less knowledge of the dynamic system.

As we expected, the agent without any initial policy did a terrible work. Nearly all the iterations were used to "query" the environment because 30 iterations is far from the time it needed for convergence. The variations in average response time were not from the algorithm's adaptation but just from the system itself.

D. Effect of Exploration

In the proceeding sections, we demonstrated that our RL based automatic configuration agent was efficient for online system adaptations. It could drive the system into high performance states quickly and into a stable configuration within less than 30 iterations. The high time efficiency implies the system won't suffer the initial poor performance for too long.

Another issue is whether the rapid convergence implies insufficient exploration and suboptimal convergence states. How to balance the exploration and exploitation is one of the key issues in RL algorithms. In our work, there were two kinds of exploration rates: one for batch training and one for online learning. The agent keeps doing both of the two learning procedures through lifetime.

The batch training is a part of the online learning. During each iteration, agent updates the performance information for current state and uses batch training to generate a latest version Q values table, based on which, the online learning algorithm takes the action and goto next state using $\epsilon - greedy$ policy. Although both two learning algorithms keep happening online, they have different affections to the system performance. The online learning algorithm is the one actually controlling which state the system should enter next. The online performance is much more sensitive to the online learning exploration rate. For each batch training, the agent accesses different states by reading data from reward table which come from either initialization or online updating. During the batch training, the agent keep the system state unchanged and wait for updated Q values table.

The duty of batch training is to generate the Q value table and to update the policy. And the duty of online training is to direct the system and to collect current information for calibrating the policy. Therefore, although the agent actually visit a very small portion of the states online, it indeed considers all the states when training the Q values table based on stored data. In our experiment, to make the batch training explore more states, we set its rate as 0.1. To avoid too many fluctuations after the convergence, we set the online learning exploration rate 0.05.

We have to accept that our algorithm can not guarantee to find the global optimal configuration. Due to the performance consideration, the agent can not explore too many sates. The gap between real system performance and our initial predicted one sometimes make the agent inaccurate. In Figure 9, we study the performance of different online learning exploration. We used three exploration rates: 0.05, 0.1 and 0.3. From the figure, we can see that the performance of the convergence states for different exploration rates were nearly the same. But, the high exploration rate caused more than one response time spikes within 30 iterations. The result shows the rate 0.05 performed best. The goal of our RL based agent is to direct the system to a good configuration according to the system SLA. Setting a vary high online exploration to find the global optimal

Fig. 9. Agent performance due to different online exploration rates.

state is not practical and causes huge performance penalty during online searching.

VI. Related Work

Many past works were devoted to autonomic configuration of web systems; see [19], [7], [17], [2], [5], [18] for examples. Xi et al. [17] and Zhang et al. [18] applied Hill-climbing algorithms to search optimal configurations for application servers by adjusting a small number of parameters. They treated the system as a black-box and assumed that the application tier configurations were independent of other tiers.

Actually, the configurations for interconnected web system components interfere with each other. In [19], Zheng et al employed a CART algorithm to generate the parameter dependency graph through a three tier web system, which explicitly represented relationship between configurable parameters. Chung et al. demonstrated that the performance improvement cannot easily be achieved by tuning individual component of web system [2]. These two works suggested to construct performance functions of parameters in a direct approach so as to tune the parameters by optimizing the functions. However, the huge number of initial testings made their works not applicable to online adaptations.

In [5], Liu et al. proposed a fuzzy control based algorithm to online optimize response time of a web server. Zhang et al. [18] developed a online tuning agent to reconfigure the application sever according to system variations. However, the inherent complexity of the control approach considerably limited capacity of their auto-configuration method. Therefore, both of these works limited themselves to the tuning of single parameter of single tier applications.

Moreover, the traditional hill-climbing and control approaches require system knowledge and suffer from delay consequences. The RL algorithm inherently avoid such problems by taking the long term rewards. Several works have employed reinforcement learning in other contexts. Tesauro et al. applied a hybrid reinforcement learning algorithm to optimize server resource allocation in a server farm [15]. Also a reinforcement learning based self-optimizing memory controller was designed in work [4]. To avoid the poor initial performance, function approximation and coarse-grain Q value table were adopted separately in these two works. In this work, we used typical data collection and pre-learning to solve this problem.

There were other works on autonomic configuration in virtual machines. Padala et al. applied classical control theory to auto-configure the resource shares allocated to each VM in order to increase resource utilization [8]. In [11], an VM advisor automatically configured VMs to adapt to different database workloads. What they focused on was resource configurations of VMs, which complements to the work in this paper on web systems configuration under VM-based dynamic platforms.

VII. Conclusion

In this paper, we propose a reinforcement learning approach, namely RAC, towards automatic configurations of multi-tier web systems in VM-based dynamic environment. To avoid initial learning overhead, we equip our RL algorithm with efficient heuristic initialization policies. Experiments in a multi-tier web system showed that RAC is applicable to online system configuration adaptation in the presence of variations in both workload and VM resources. It is able to direct the web system to a near-optimal configuration within less than 25 trial-and-error iterations .

Although our RL-based auto-configuration agent performed well in the experiments, it still has room for improvement. First, the quality of collected data and accuracy of the predictor will affect the agent's online performance. Designing a more accurate initial model or function approximation is one of our future extended work. Furthermore, due to the presence of policy shifting delay, our agent still suffer a short period of initial poor performance. A quicker adaptive auto-configuration agent is expected to lead to better performance.

References

[1] A. Bar-Hillel, A. Di-Nur, L. Ein-Dor, R. Gilad-Bachrach, and Y. Ittach. Workstation capacity tuning using reinforcement learning. In *SC*, 2007.

[2] I.-H. Chung and J. K. Hollingsworth. Automated cluster-based web service performance tuning. In *HPDC*, pages 36–44, 2004.

[3] S. Fu and C.-Z. Xu. Service migration in distributed virtual machines for adaptive grid computing. In *ICPP*, pages 358–365, 2005.

[4] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[5] X. Liu, L. Sha, Y. Diao, S. Froehlich, J. L. Hellerstein, and S. S. Parekh. Online response time optimization of apache web server. In *IWQoS*, pages 461–478, 2003.

[6] D. L. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[7] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *SIGMETRICS*, pages 145–156, 2007.

[8] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[9] Rice University Computer Science Department. http://www.cs.rice.edu/CS/System/DynaServer.

[10] J. S.Graupner and S.Singhal. Making the utility data center a power station for the enterprise grid. In *Technical Report HPL-2003-53, Hewlett Packard Laboratories, March 2003*, 2003.

[11] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath. Automatic virtual machine configuration for database workloads. In *SIGMOD Conference*, 2008.

[12] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
[13] G. Tesauro. Online resource allocation using decompositional reinforcement learning. In *AAAI*, 2005.
[14] G. Tesauro, R. Das, H. Chan, J. Kephart, D. Levine, F. Rawson, and C. Lefurgy. Managing power consumption and performance of computing systems using reinforcement learning. In *Advances in Neural Information Processing Systems 20*. 2007.
[15] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, 2007.
[16] J. Wildstrom, P. Stone, and E. Witchel. Carve: A cognitive agent for resource value estimation. In *ICAC*, 2008.
[17] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW*, pages 287–296, 2004.
[18] Y. Zhang, W. Qu, and A. Liu. Automatic performance tuning for j2ee application server systems. In *WISE*, pages 520–527, 2005.
[19] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *EuroSys*, pages 219–229, 2007.