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; their settings are crucial to systems performance and service availability. Manual configuration based on operator's experience is a non-trivial and error-prone task. Recent studies revealed that more than 50% root causes of Internet service outages 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, both of the Apache server and Tomcat server have more than a hundred configurable 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. Performance optimization of individual component does not necessarily lead to overall system performance improvement [2]. Therefore, to find an appropriate configuration, the operator must develop adequate knowledge about the system, get familiar with each of parameters, 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 components 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. [20] demonstrated that in a cluster-based Internet service, the system configuration should be modified to adjust to cluster nodes updates.

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 [12] 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 [11], [17]. 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 [18], [19], [2], [20] for examples. Most of them focused on performance parameters tuning for dynamic workload in static environments. Their optimization approaches are hardly applicable to online setting of the parameters in VM-based dynamic platforms due to their high time complexity. There were a few control approaches targeted at online tuning in response to changing workload [5]. They were largely limited to tuning of single `MaxClient` parameter because of the inherent complexity of the control.

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 [14], [16], [9], power management [15], job scheduling in grid [1] and self-optimizing memory controller [4]. To best of our knowledge, the RAC 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.

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 policy initialization 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.

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 35	30
minSpareThreads	from 5 to 85	5
maxSpareThreads	from 15 to 95	50

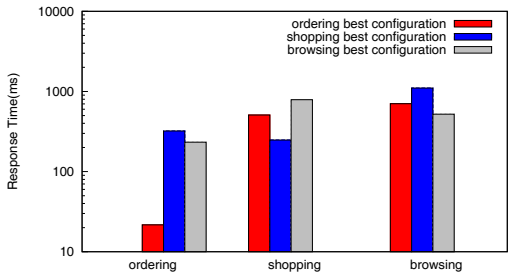


Fig. 1. Performance under configurations tuned for different workloads.

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 assumed the default settings for the MySQL parameters.

We tested the 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 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 the following, we still use `MaxClients` parameter

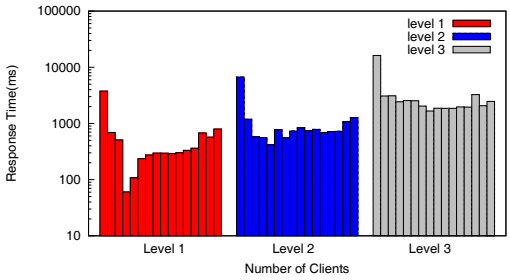


Fig. 2. Effect of `MaxClients` on performance.

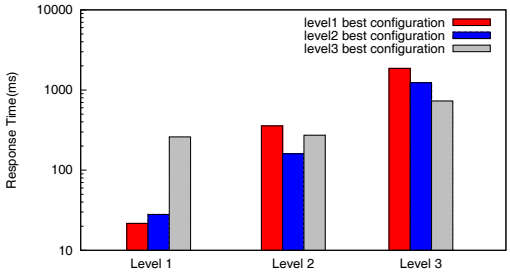


Fig. 3. Performance under configurations tuned for different VMs.

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 different `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.

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 [20], authors used parameters dependency graph to find the performance relevant parameters and the relationship among them. Our focus is 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 measures the web system performance at a predefined time interval (we set it to 5 minutes in experiments), and sends 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 requires no OS-level or hardware level information for portability. The decision maker runs a RL algorithm and produces 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 [13].

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.: 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 [13], 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

```
Initialize Q table
Initialize state s_t
error = 0
repeat
    for each state s do
        a_t = get_action(s_t) using ε − greedy policy
        for (step = 1; step <LIMIT; step++) do
            Take action a_t observe r and S_{t+1}
            Q_t = Q_t + α * (r + γ * Q_{t+1} − Q_t)
            error = MAX(error, |Q_t − Q_{previous−t}|)
            s_t = s_{t+1}, a_{t+1} = get_action(s_t), a_t = a_{t+1}
        end for
    end for
until error < θ
```

IV. Online Learning and Adaptation

RL algorithm explores system dynamic features by interacting with the external environment. 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]. The initial poor performance and long time convergence make 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.

First, to learn the initial policy, we need to collect training data for the subsequent RL Learning. It is not practical to collect the performances of all the configurations due to its long time consumption. A key issue is to choose representative states for approximation. In implementation, we use a technique named *parameter grouping* to group parameters with similar characteristics together so as to reduce the state space. For example, both parameters `MaxClients` and `MaxThreads` are limited by the system capacity and both parameters `KeepAlive timeout` and `session timeout` are limited by the number of multiple connection transactions. Then the first two parameters form one group and the other two form another group. The parameters in the same group are always given the same value. Moreover, coarse granularity is used for each group during training data collection instead of the fine granularity used in online learning.

After having the state value of representative configurations, we use a simple but efficient method to predict the performance of other 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. By using polynomial regression algorithm, we formulate the performance as a function of configurable parameters and predict performance of the absent states in the data collection step.

After getting all the training data, we run a offline reinforcement learning process showed in Algorithm 1 to generate a initial Q value table for online learning. In implementation, we set α = 0.1, γ = 0.9, ϵ = 0.1 for the

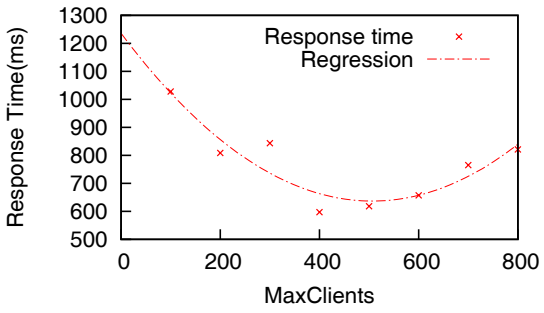


Fig. 4. Concave upward effect of `MaxClients` and regression.

offline training. Algorithm 2 gives the pseudo-code of the policy initialization algorithm.

Algorithm 2 Policy Initialization.

1: **Parameter Grouping**
2: **for** each group **do**
3: **Collect** data in coarse granularity
4: **end for**
5: **Generate** regression-based predicting function
6: **Predict** performance of unvisited configuration
7: **Run** RL process to learn an initial policy

B. Online Learning

Although the policy initialization could avoid the initial poor performance, it may not be accurate enough for reconfiguration decision. In the subsequent online learning, our agent keeps measuring the current system performance and retraining the Q value table at each time interval using Algorithm 1with $\alpha = 0.1$, $\gamma = 0.9$, $\epsilon = 0.05$. 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 using batch training so as to guarantee that most of the states are 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 System Dynamics

Recall the web system hosted in a VM-based 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 types of workloads and VM resource profiles. We call the combinations of traffic mixes and the VM resource settings system contexts. To address the problem of poor initial performance and accelerate the learning process, we construct different initialization policies for different scenarios through offline training. Algorithm 3 shows the online training and adaptation algorithm. The RL agent continuously collect the immediate reward in each configuration, and compares it with the average of the last n values. An abrupt change in the reward value is considered as a violation. If violations are detected in several consecutive iterations, the agent believes that there is a context change and switches to a corresponding initial policy. The violations are detected based on a violation threshold v_thr. Once there are s_thr times violations happened continuously, the agent will switch to a most suitable initial policy according to the current performance. The threshold s_thr controls the trade-off between the agent's adaptability and stability. Setting it to a too small value will make the agent too sensitive to system fluctuations but a too large value will harm the agent's adaptability. The effect of the s_thr will be discussed in section V.B. Empirically, we set n, s_thr, and the v_thr to 10, 5, and 0.3, respectively.

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

$$violation = \begin{cases} 0 & \text{if } pvar \leq v\_thr; \\ 1 & \text{otherwise.} \end{cases}$$

Algorithm 3 Online Learning.

1: **Input** initialized Q table
2: **Input** initialized state S_t
3: **for** each configuration iteration **do**
4: **Issue** reconfiguration action based on current Q value table
5: **Measure** current performance
6: **Check** context variations
7: **If** number of consecutive violations exceeds s_thr
8: **Then Switch** policy
9: **Update** Q value table using Algorithm 1
10: **Enter** the next step
11: **end for**

V. Experiments Results

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

A. Experimental Setup

To evaluate the effectiveness of the RAC approach, we deployed a multi-tier website in a VM-based environment. The physical machine was configured with two Intel quad-core Xeon CPUs and 8GB memory. A client machine with the same hardware configuration was used to emulate concurrent customers. All experiments were conducted in the same local network.

The physical machine hosting the multi-tier website installed Xen virtual machine monitor(VMM) version 3.1. Xen is a high performance resource-managed VMM, which consists of two components: a hypervisor and a driver domain. The hypervisor provides the guest OS the illusion of occupying the actual hardware devices. The driver domain is in charge of managing other guest VMs and executes resource allocation policies. In our experiments, both the driver domain and the VMs were running CentOS 5.0 with

TABLE II. System Contexts

	Workload mixes	VM resources
Context-1	Shopping	Level 1
Context-2	Ordering	Level 1
Context-3	Ordering	Level 3
Context-4	Shopping	Level 2
Context-5	Ordering	Level 2
Context-6	Browsing	Level 1

Linux kernel 2.6.18. The multi-tier website was deployed on two VMs, with Apache web server in the first one and Tomcat application server and MySQL database server in the other one. The RAC agent resided in the driver domain.

We evaluated the RAC approach using the TPC-W benchmark [10]. TPC-W benchmark defines three different workload mixes: ordering, shopping, and browsing. They have different combinations of browsing and ordering requests. To evaluate the RAC approach's adaptation to system dynamics, we changed the client traffic as well as the resources allocated to the VMs hosting the website. Considering the fact that the application server and data base server are the bottleneck for our system, only the resources allocated to the VM hosting the last two tiers are changed. Table II lists the combinations of the client traffic and VM resources.

B. Performance of Configuration Policies

In this section, we studied the effectiveness of the RAC approach for online auto-configuration. We compared the RAC agent with the other two configuration approaches. The first one is with static default parameter settings as listed in Table I. The other is a trial-and-error method based on tuning individual parameters. This approach assumes that all parameters have a concave upward effect on system performance. More specifically, the trial-and-error method tunes the system starting from an arbitrary parameter and fixes the remaining parameters. The parameter setting that produces the best performance is selected as the optimal value for this parameter. Then the agent goes to the next parameter. Once all the parameters are processed, the resulted parameters settings are considered as the best configuration for the system. This approach mimics the way an administrator may use to tune the system manually.

In this experiment, we dynamically changed the system contexts to evaluate the adaptability of the RAC agent. The system stayed in one context for 30 iterations before switching to a new one. Figure 5 plots the online performance of the RAC agent compared to the other two methods in three consecutive system contexts: context-1(from 0 to 30 iteration), context-2(from 31 to 60 iteration) and context-3(from 61-90 iteration).

From Figure 5, we can see that RAC agent performed best among the three approaches. It was always able to drive the system to a stable state in less than 25 interactions with the external environment. Its overall performance was around 30% better than the trail-and-error agent and 60% better than the static default configuration. Furthermore, the RAC agent was able to adapt to context change in a timely way. After

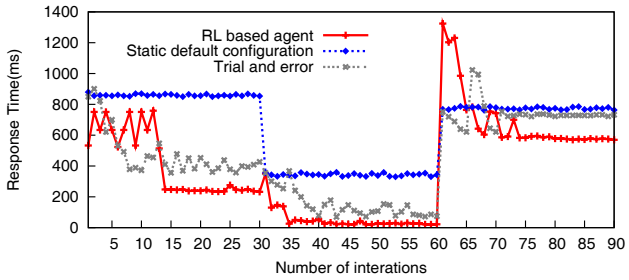


Fig. 5. Performance of different auto-configuration policies.

the client traffic changed at the 30th iteration, the RAC agent continuously observed performance violations and switched policy at iteration 35. The response time dropped more than 60% after RAC agent got the new initial policy. The new policy can lead the system to a stable configuration within 15 iterations. More importantly, the RAC agent consistently improved the performance during the process of parameter reconfiguration. It could optimized the cumulative reward during the reconfiguration steps avoiding severe performance degradation.

As we expected, the static default configuration yielded the worst performance in most of the test cases. Because there was no adaptation to the system context variations, the static configuration was not suitable for dynamic environment. For most of the time, the trail-and-error agent produced a much better performance and it was able to drive the system to a stable state. However, because this approach was based on tuning individual parameters independently, the agent was prone to being trapped in local optimal settings. Figure 5 shows that the performance of the stable states found by trail-and-error agent were at least 30% worse than those found by the RAC agent. Moreover, with the increase of the number of tunable parameters, convergence to a stable state would become a challenge in this approach due to huge size of the search space.

We notice that, in some cases, the resulted performance of the RAC agent was not so good as others. For example, in the second context transition at the 60th iteration, the system experienced five iterations of poor performance before the agent detected the context changes. We refer to the detection delay as policy switching delay. The policy switching delay can be mitigated by reducing s_thr. However, this may cause the system unstable due to false detection of context change and frequently switch of initial policies. In our work, our original setting of 5 worked well and several iterations policy switching delay should be acceptable for the web system.

C. Effect of Online Learning

Recall that the initial policies are learned from offline traces for specific system contexts. In this experiment, we study whether it is necessary to refine the learned policies through online learning.

Figure 6 compares the agent's performance with and without online learning. The agent without online learning

drove the system to a stable configuration in 12 iterations less than the one uses online learning. However, the policy refined by online learning achieved better stable performance. There is approximately 50% performance improvement from the online refinement than the offline trained policy. The slower convergence to a stable state and the fluctuations at the beginning in online learning attribute to the process of online interactions which involve a certain amount of exploration actions. Explorations are often considered as sub-optimal actions that explore environment dynamics. Although online learning suffered a longer convergence time and initial fluctuation, it was able to find a better configuration than the offline policy.

D. Effect of Policy Initialization

The state space of the RL algorithm grows exponentially with the number of configurable parameters. Without an initial policy, the RL algorithm performs a large amount of explorations, which are believed to be suboptimal actions leading to bad performance, before obtaining a stable policy. In this section, we studied the effect of policy initialization on improving RL online performance.

Figure 7(a) and Figure 7(b) show the performance of the RAC agents with and without policy initialization in system context-2 and context-4. From the figures, we observe that the agent with policy initialization led to considerable performance improvements. In Figure 7(b), after 8 iterations, the response time due to the RAC agent without policy initialization was always above 6 times longer than the one with initialization. In Figure 7(a), the performance gap was not as big as in Figure 7(b), but remains substantial. The differences may be due to the fact that the optimal state in context-2 was much closer to the default configurations than in context-4. In both cases, the agents with policy initialization were able to drive the system to a stable state in less than 12 iterations. In contrast, the agents without policy initialization failed to generate fixed policies with stable configurations in a small number of interactions.

Note that, in a dynamic web system, it is not always possible to derive sufficient environment specific initial policies for all the contexts. In this experiment, we show that the initial policies attained for certain system contexts can be used as an good starting point for online learning. The RL algorithm continuously revises the policy through online interactions. We compared the performance of the agent using static policy with the one that adaptively switches policies upon context changes. In the experiment, we randomly selected the initial policy derived from system context-2 as the static policy. We evaluated the performance of the static initial policy and adaptive initial policy under system context-5 and system context-6 separately. Figure 8(a) and Figure 8(b) show the results, respectively.

The figures suggest that the agents with static policy initialization were able to drive the systems to stable states in less than 27 iterations. The static policies were gradually refined by online learning and the response time improved as more interactions were performed. Although the agents with static policies needed more time to converge to a stable configuration, the resulted configurations yield similar performance to the ones generated by dynamic policies. In contrast with the agent without any policy initialization, the agent with static policy initialization was able to keep the response time at a relatively low level after a limited number of iterations, for example, 12 iterations in Figure 8(b) and 19 iterations in Figure 8(a).

Both the online batch training and the characteristics of web systems contributed to the effectiveness of the static initial policy. During each iteration, newly measured performance information was used to retrain the Q table. The recent attained rewards spread the environment dynamics to all the states. Therefore, although the static initial policy can not accurately reflect the system dynamics, the interactions between the agent and the external environment was able to calibrate the mapping from configuration to performance within an acceptable amount of time. Moreover, for a web system, some extreme configurations are rarely used in practice. For example, in our experiment, setting the `KeepAlive timeout` to a value higher than 20 seems a bad decision. Because few web pages in TPC-W benchmark requires the TCP connections to be kept for such a long time. The static policy can automatically mask these impractical configurations and avoid possible performance degradations.

Such RAC agent with carefully designed initial policy is more practical in real systems. It relies on interactions with environment instead of policy switching to continuously update the Q values table, and assumes much less knowledge of the dynamic system.

In the next experiment, we evaluate the online adaptation of three RL algorithms: the algorithm without policy initialization, with static policy initialization, and with adaptive policy initialization. We dynamically changed the system contexts in the same way as the previous experiments in Section V.B.

Figure 10 plots the performance of the three RL algorithms. From the figure, we can see that the agent with adaptive policy initialization performed best during online adaptations. The agent with static policy initialization was also able to adapt to the system context variations and achieved comparable performance as the adaptive agent. At iteration 30 and iteration 60, the system experienced a workload change and a VM resource reallocation, respectively. The agent with static policy initialization successfully detected the variations and refined the policy within 25 iterations based on interactions with the new environment. Its resulted configuration generated good performance which only have less than 10% loss compared to the best possible performance obtained by the adaptive agent. As expected, the agent without any initial policy can not lead the system to a stable state and its performance was much worse than the other two agents.The variations in average response time were not from the algorithm's adaptation but just from the system itself.

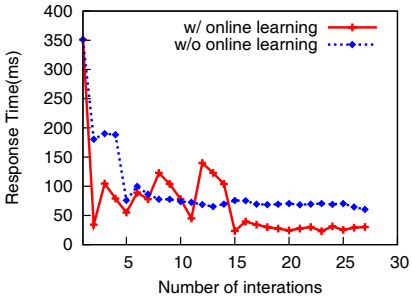
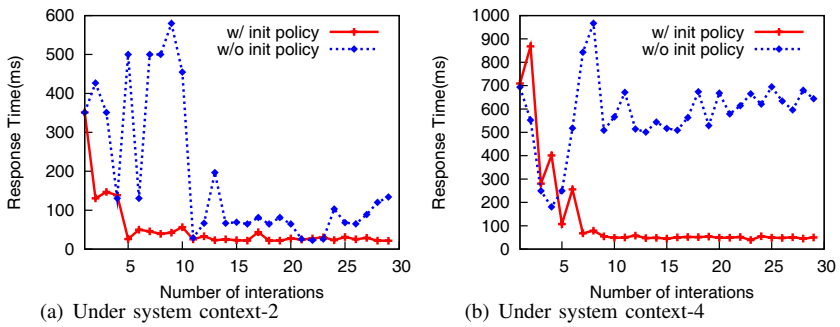
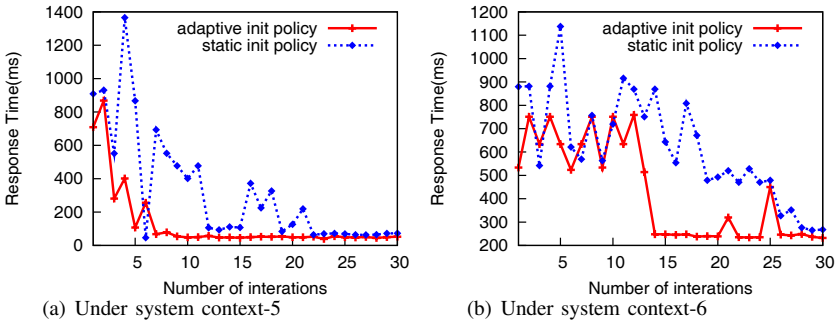


Fig. 6. Effect of online training.



(a) Under system context-2 (b) Under system context-4

Fig. 7. Agent performance with and without policy initialization.



(a) Under system context-5 (b) Under system context-6

Fig. 8. Agent performance with static and adaptive policy initialization.

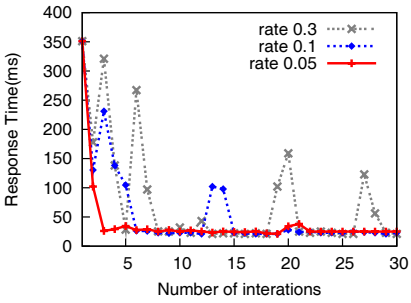


Fig. 9. Effect of the different online exploration rates.

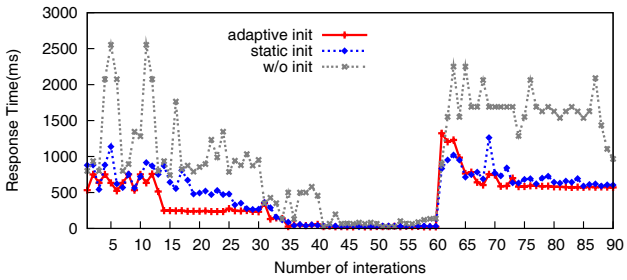


Fig. 10. Performance of different RL policies.

E. Effect of Exploration

How to balance exploration and exploitation is one of the challenges in online RL algorithms. Insufficient explorations would result in suboptimal configurations while too much exploration would incur prohibitive performance degradation. In this section, we studied the effect of the exploration rate in RAC performance. Two types of explorations were considered: in batch training and in online learning.

Batch training is a part of the online learning process. During each iteration, agent updated the performance information for current state and uses batch training to generate a latest Q value table. The online learning algorithm makes a reconfiguration decision based on the newly updated Q table. The online performance of RAC is more sensitive to the online learning exploration rate. In our experiment, we set a higher exploration rate of 0.1 for batch training in order to make best use of collected performance information. A smaller exploration rate of 0.05 was used for online learning to avoid fluctuations and performance degradation.

Figure 9 shows the effect of online learning exploration in three exploration rates: 0.05, 0.1 and 0.3. From the figure, we can see that the performance of the resulted stable state for different exploration rates were nearly the same. But, a higher exploration rate may lead to more suboptimal exploration actions in turn and resulted in response time spikes. For example, Figure 9 shows 2 spikes in the case of rate 0.1 and 4 spikes in the case of rate 0.3. Moreover, during such fluctuation, the response times increased at least 4 times. The result shows that the rate 0.05 performed best.

VI. Related Work

Many past works were devoted to autonomic configuration of web systems; see [20], [7], [18], [2], [5], [19] for examples. Xi et al. [18] and Zhang et al. [19] 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 [20], 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. [19] 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 [16]. 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 [12], an VM advisor automatically configured VMs to adapt to different database workloads. What they focused on were 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 the 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 the RL-based auto-configuration agent performed well in the experiments, it still has room for improvement. First, the quality of collected training data will affect the agent's online performance. Designing a more accurate initial model or function approximation is one of our future extended work. We used parameter grouping and coarse granularity techniques to reduce time for collecting training data. It still took more than ten hours to get sufficient system information. Furthermore, the time for data collection will increase with the number of parameters exponentially. Therefore, scalability of the approach remains a challenge. To apply our approach in a real complex system, configurable parameters need to be selected automatically in a more efficient way.

Acknowledgement

We would like to thank the anonymous reviewers for their constructive comments and other group members for their help. This research was supported in part by U.S. NSF grants CCF-0611750, DMS-0624849, CNS-0702488, and CRI-0708232

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] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. Vconf:a reinforcement learning approach to virtual machine auto-configuration. In *ICAC*, 2009.

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

[11] 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.

[12] 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.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[14] G. Tesauro. Online resource allocation using decompositional reinforcement learning. In *AAAI*, 2005.

[15] 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.

[16] 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.

[17] J. Wildstrom, P. Stone, and E. Witchel. Carve: A cognitive agent for resource value estimation. In *ICAC*, 2008.

[18] 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.

[19] Y. Zhang, W. Qu, and A. Liu. Automatic performance tuning for j2ee application server systems. In *WISE*, pages 520–527, 2005.

[20] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *EuroSys*, pages 219–229, 2007.