2012 IEEE 26th International Parallel and Distributed Processing Symposium

Automated and Agile Server Parameter Tuning with Learning and Control

Yanfei Guo, Palden Lama and Xiaobo Zhou
Department of Computer Science
University of Colorado, Colorado Springs, USA
Email addresses: {yguo, plama, xzhou}@uccs.edu

***Abstract*—Server parameter tuning in virtualized data centers is crucial to performance and availability of hosted Internet applications. It is challenging due to high dynamics and burstiness of workloads, multi-tier service architecture, and virtualized server infrastructure. In this paper, we investigate automated and agile server parameter tuning for maximizing effective throughput of multi-tier Internet applications. A recent study proposed a reinforcement learning based server parameter tuning approach for minimizing average response time of multi-tier applications. Reinforcement learning is a decision making process determining the parameter tuning direction based on trial-and-error, instead of quantitative values for agile parameter tuning. It relies on a predefined adjustment value for each tuning action. However it is nontrivial or even infeasible to find an optimal value under highly dynamic and bursty workloads. We design a neural fuzzy control based approach that combines the strengths of fast online learning and self-adaptiveness of neural networks and fuzzy control. Due to the model independence, it is robust to highly dynamic and bursty workloads. It is agile in server parameter tuning due to its quantitative control outputs. We implement the new approach on a testbed of virtualized HP ProLiant blade servers hosting RUBiS benchmark applications. Experimental results demonstrate that the new approach significantly outperforms the reinforcement learning based approach for both improving effective system throughput and minimizing average response time.**

I. Introduction

Internet server applications have many configurable parameters. In *Apache*, a popular web server has important parameters such as MaxClients, KeepAliveTimeout, MaxSpareServers and MinSpareServers that control server concurrency level, network link alive time, and worker process generating. These parameters are very important to the performance of server applications and to the resource utilization of the underlying computer system. However, server parameter tuning is a very complex task that highly relies on an administrator's experiences and understanding of the server system. An improper configuration often leads serious consequences. According to the study [19], in average more than 50% of service failures is due to the misconfiguration. In some particular scenarios, misconfigurations caused almost 100% of service failures.

In modern data centers, user-perceived performance is the result of complex interaction of very complex workloads in a very complex underlying server system [16], [17]. The complexities are due to high dynamics and burstiness of Internet workloads, multi-tier Internet service architecture, and virtualized server infrastructure. Recent studies [16], [17], [23] have observed significantly dynamic workloads of Internet applications that fluctuate over multiple time scales, which can have a significant impact on the processing and power demands imposed on data center servers. The burstiness in incoming requests in a server system can lead to significant server overload and dramatically degradation of server performance [16]. In the worst case, bursty workload can cause service unavailability.

In today's popular multi-tier Internet service architecture, a set of servers are divided by their functionality like a pipeline. Servers in each tier use the functionality provided by their successors and provide functionality to their predecessors. Incoming workloads are often unequally distributed across different tiers. Some tiers may run in the saturated or overloaded state while others are under-loaded. Highly dynamic workloads will also result in the bottleneck shifting across tiers [5]. The inter-tier and intra-tier performance dependences further complicate the configuration of a multi-tier server system. The complexities and challenges demand for automated and agile server parameter tuning.

There are a few recent studies focusing on automated server parameter tuning with reinforcement learning for multi-tier Internet applications [2], [21], [29]. The studies demonstrated that the reinforcement learning based approaches were able to minimize the average response time of a multi-tier application under stationary workloads. There are two major limitations. First, reinforcement learning is a decision making process for the tuning direction. It does not generate a quantitative result of the server parameter value change. It needs a predefined adjustment value for each tuning action. However, it is nontrivial or even infeasible to find an optimal value for each tuning decision. Second, those approaches were executed under stationary workloads. The approaches may not be effective and agile for automated server parameter tuning under highly dynamic and bursty workloads.

In the face of the challenges of highly dynamic and bursty workloads, parameter dependences, and various application characteristics, we design a neural fuzzy control that integrates the strengths of self-constructing online learning of neural networks and fast tuning of fuzzy control. The resulted approach is model-independent, robust and agile for automated server parameter tuning under highly dynamic and bursty workloads in virtualized data centers. We use the new approach for improving both the effective system throughput and the average response time of multi-tier Internet applications. We

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.66

implement the new approach in a testbed of virtualized HP ProLiant blade system. Like others [2], [10], [11], [21], we adopt RUBiS, an e-transactional benchmark application [1], [22] for performance evaluation and use three different workload characteristics: stationary, bursty, and step-change.

We conduct extensive experiments to compare the neural fuzzy control based and reinforcement learning based approaches in automated server parameter tuning for improving performance of multi-tier Internet applications. Results find that the neural fuzzy control based automated configuration approach can achieve more than 80% higher effective throughput than that due to default system configurations. It outperforms the reinforcement learning based automated configuration approach by about average 10% to 20% in terms of effective system throughput. The improvement is mainly due to the agility of the new approach in finding ideal server parameter configurations. Importantly, under highly dynamic step-change workloads, the reinforcement learning based approach may not converge in time in finding effective server parameter configurations. It results in significant performance penalties. Its achieved effective system throughput is just about 40% of that due to the neural fuzzy control based approach. And, its resulted average response time is about 35% higher that that due to the neural fuzzy control based approach.

Our contributions lie in the design and development of an automated and agile server parameter tuning approach that can significantly improve the performance of complex multi-tier Internet applications, the use of effective system throughput as the primary performance metric, the analysis of the weaknesses of reinforcement learning for server parameter tuning, and the implementation of the proposed approach in a virtualized data center.

In the following, Section II discusses related work. Section III describes the automated server parameter tuning problem. Section IV gives the neural fuzzy control based and reinforcement learning based approaches. Section V introduces the testbed implementation. Section VI presents the experimental results and analysis. Section VII concludes the paper.

II. Related Work

Autonomic computing aims to reduce the degree of human involvement in the management of complex computing systems [6]. Recently, autonomic computing in modern data centers has become a very active and important research area [2], [6], [8], [15], [18], [20], [21], [23], [24], [27], [29]. Those studies focused on capacity planning for virtual machines (VMs) co-location and distribution across a data center [8], [15], [26], VM provisioning for applications [3], [9], [10], [12], [24], [25], resource allocation in a VM [20], [21], [7], and server parameter tuning [2], [29].

Automated server parameter tuning is one key but challenging research issue. There were early studies that explored automated server parameter tuning problem on Web servers [4], [14], [28]. Those works studied how server application parameters can affect the user perceived performance and how to automatically tuning those parameters. However, they focused the automated server parameter tuning problem in one server or one tier of servers. For example, Liu *et al.* focused on improving online response time of an Apache web server by tuning value of parameter *MaxClients* [14]. They applied rule-based fuzzy control for parameter tuning. As the rule-based fuzzy control is not model-independent, its pre-configured rules will determine the actions of the fuzzy controller. To create the rules, they modeled the application server using queueing models. However, queueing theoretic approaches are often not effective in modeling workloads of complex multi-tier Internet applications due to the inter-tier dependences and per-tier concurrency limit [5], [9].

A few recent studies focused on automated server parameter tuning at multiple-tier servers [2], [21], [29]. A representative approach was proposed in [2]. It used reinforcement learning for automated tuning of web-tier server parameters and application-tier server parameters in a coordinated manner. It aimed to minimize the average response time of a multi-tier online web application. Furthermore, it employed an online monitoring based Q-table switching mechanism, which can improve the adaptiveness of the tuning approach regarding various workload characteristics such as the TPC-W benchmark's ordering, shopping and browsing workload mixes. The work provided insights on the automated parameter tuning problem in complex multi-tier Internet systems. However, it did not consider the impact of highly dynamic and bursty workloads on the agility of automated parameter tuning.

Reinforcement learning itself is a decision making process. It only decides what tuning direction to be applied to a server parameter, i.e., increasing, decreasing or hold. It does not generate a quantitative result of the server parameter value change. In practice, a reinforcement learning based approach relies on a predefined adjustment value for each tuning decision. Finding a good predefined value is crucial to the performance of a reinforcement learning based approach. However, under highly dynamic and bursty workloads, finding such a good predefined value is nontrivial or even infeasible.

To address the weaknesses of reinforcement learning based approach, we integrate the strengths of fast online learning and self-adaptiveness of neural networks and fuzzy control. We promote the use of effective system throughput, a key performance metric to online web applications, in addition to the average response time. Experimental results based on the implementation demonstrate significant performance improvement due to the new automated and agile approach.

III. Automated Server Parameter Tuning

A. Challenges and Issues

Highly dynamic and bursty workloads require an agile and robust approach for automated server parameter tuning. Application level performance heavily depends on the characteristics of the workload. Server parameters must be tuned to match current system workloads. However, Internet workloads are highly dynamic and the workload characteristics keep changing. Online matching the server parameter configuration to the changes is a very challenging problem.

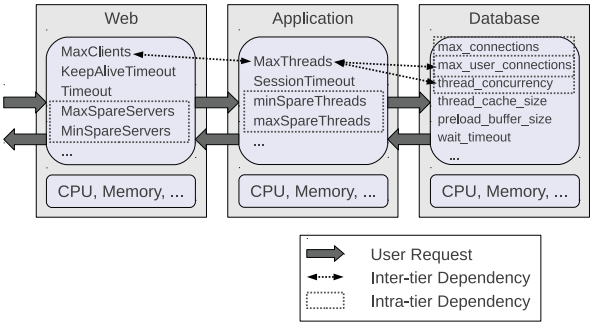


Fig. 1. Parameter dependences in a multi-tier server architecture.

There are different parameter dependences of servers in a multi-tier application, which require a coordinated approach for automated server parameter tuning across all tiers.

- Inter-tier parameter dependency: In a multi-tier application, each tier utilizes the functionality provided by its successor tier. Performance variation in one tier will affect user-perceived end-to-end system performance. As Figure 1 shows, the concurrency capacity of *Apache* web server, *Tomcat* application server, and *MYSQL* database server are controlled by parameters *MaxClients*, *MaxThreads*, *max_user_connection*, and *thread_concurrency*, respectively. These parameters need to be configured carefully to match the workload distribution on all server tiers. If we increase the concurrency capacity of web tier and leave no changes to other tiers, the web tier will try to process more user requests concurrently, which will result in more requests to the successor tiers. This will increase the response time at the successor tiers or even overload them, resulting in end-to-end performance degradation or even service outage.
- Intra-tier parameter dependency: There are intra-tier dependences of server parameters at each tier. For example, at the web tier, parameter *MaxSpareServers* must have a greater value than parameter *MinSpareServers*. It is also the case at the application tier between parameters *maxSpareThreads* and *minSpareThreads*. At the database tier, the intra-tier dependence is more complicated. Parameter *max_user_connections* has inter-tier dependency with parameter *MaxThreads* at its predecessor tier, but also intra-tier dependency with parameters *max_connections* and *thread_concurrency* at the same tier.

Dynamically changing application characteristics require a model-independent approach. The capacity of a web system is constrained by the underlying hardware resources. But the amount of hardware resources does not always provide the same capacity of a web system because it also depends on what application it hosts. For example, a web application based on dynamic pages needs more resources than one based on static pages. Performing server parameter tuning must consider the differences among various web applications.

B. Effective System Throughput

We propose to maximize the effective system throughput via automated server parameter tuning. Effective system throughput is defined as the number of requests that meet the service level agreement (SLA) requirement on the response time. While the average response time of requests is important to individual users, the effective system throughput is more important to the application provider in clouds [18].

We use a SLA with two response time bounds, hard response time and soft response time. The *absolute effective throughput* is the number of requests that are processed within the SLA time bounds. If a request is processed between the hard and soft response time bounds, its effective throughput is measured according to a utility decaying function. Figure 2 depicts three decaying functions that describe various utility of request response times. The linear decaying function implies moderate penalty on requests that exceeded the soft response time bound, the exponential decaying function imposes higher penalty on requests that exceeded the soft response time bound and the logarithmic decaying function is more tolerant to requests that exceeded the soft response time bound. The *relative effective throughput* is the ratio of the absolute effective throughput to the total number of incoming requests.

The work in [2] aimed to minimize the average response time of all requests. However, minimizing the average response time of requests does not necessarily maximizing the effective throughput of the system. To minimize the average response time, server parameter configurations will be tuned to devote more resources processing each request, which in turn would make resource scarce when needed.

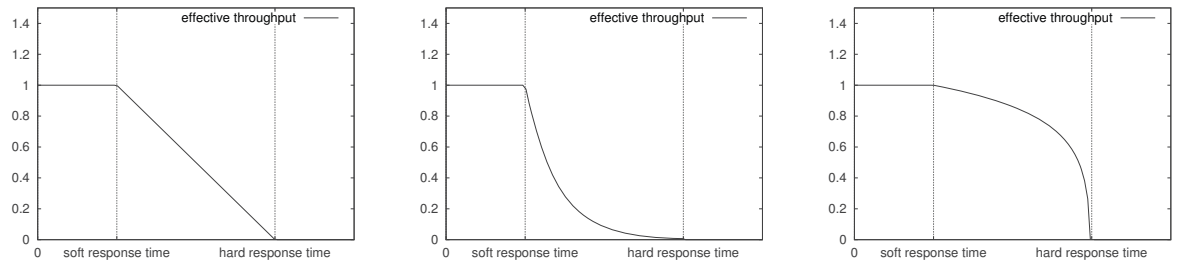
IV. Approaches with Learning and Control

A. An Enriched Neural Fuzzy Control based Approach

1) Architecture and Features: Due to the challenges of highly dynamic and bursty workloads, inter-tier and intra-tier parameter dependences, and various application characteristics, we design an enriched neural fuzzy control that integrates the strengths of self-constructing online learning of neural networks and fast tuning of fuzzy control.

A general rule-base fuzzy control consists of a fuzzification stage, a rule-base stage, and a defuzzification stage. The fuzzification stage maps numerical inputs into linguistic fuzzy values by appropriate membership functions. The rule-base stage invokes fuzzy logic rules and combines the results of those invoked rules in a linguistic output value. Finally, the defuzzification stage converts the output value back into a numerical output value for the controlled system.

We integrate a neural network with the general rule-base fuzzy control. This enables the integrated controller to automatically construct its neuron structures and adapt its parameters. Figure 3 shows the block diagram and control flows of the controller. The task of controller is to adjust server configurable parameters on a multi-tier system in order to improve the performance metric ${}^{e}T_d$ such as the effective system throughput or the average response time of requests.



(a) Linear decaying (b) Exponential decaying (c) Logarithmic decaying

Fig. 2. Three decaying functions for various utility of request response times.

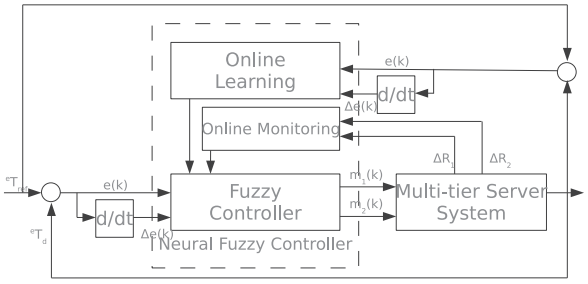


Fig. 3. The block diagram of the neural fuzzy control.

The neural fuzzy controller has two inputs: error notated as $e(k)$ and error changing rate noted as $\Delta e(k)$. We define error as the difference between the achieved performance and the tuning objective notated as ${}^eT_{ref}$ during the k_{th} tuning period. That is, $e(k) = {}^eT_d - {}^eT_{ref}$. The output is the parameter value $e_i(k)$ for the next tuning period. To support coordinated server parameter tuning for multi-tier applications, we enrich the neural fuzzy controller with an online monitoring component that keeps monitoring real-time workload distributions at each tier. Based on the monitoring data, the controller updates the server parameter values at all tiers in proportion to their workload distributions.

The neural fuzzy control has following features:

- Model-independence: Fuzzy control is suitable for non-linear, time-variant, and model-incomplete systems. The workload characteristic changes will not affect the functionality of automated server parameter tuning.
- Self-construction: The structure of the neural fuzzy controller is automatically generated. The neurons and weights are dynamically changed during the server parameter tuning process.
- Robustness: Because the self-construction and model-independence, the neural fuzzy controller can adjust itself to match dynamic workload variations. This results in the robustness of controller.
- Cross-tier coordination: The neural fuzzy controller treats the multi-tier system as whole. According to parameters inter-tier dependency and intra-dependency, each tuning will be applied to related parameters at each tier at once.

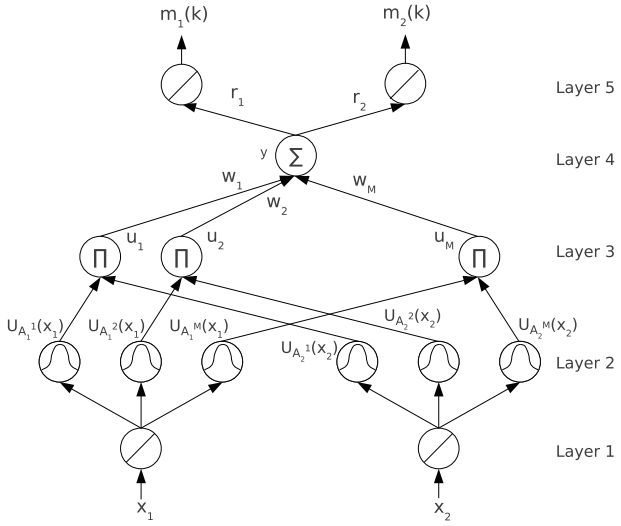


Fig. 4. Schematic diagram of the neural fuzzy controller.

2) Design of the Neural Fuzzy Controller: The design of the neural fuzzy controller is shown in Figure 4. We develop a five-layer neural network. The interconnected neurons play the role of membership functions and the rule base as in a traditional fuzzy control. However, unlike a traditional fuzzy controller, the neural fuzzy controller has multiple outputs, one output for one tier in a multi-tier system. Figure 4 shows a two-tier example. The neural fuzzy controller is initialized in a neural network with eight neurons. This minimal structure only contains two membership functions and one rule. More membership functions and rules will be dynamically constructed and adapted as the neural network grows and learns.

The design details of each layer are as follows:

Layer 1: This is the input layer. At this layer, each neuron is associated with one input variable. There are two neurons for inputs $e(k)$ and $\Delta e(k)$, respectively. The activation functions of these two neurons pass the inputs to the next layer for fuzzification.

Layer 2: This is the fuzzification layer. At this layer, each neuron represents a linguistic term. The activation functions of neurons determine how to transform input values into linguistic terms. We set the activation functions using a

Gaussian function Eq. (1). We cluster the linguistics terms into two groups, i.e., error $e(k)$ and change in error $\Delta e(k)$. The Gaussian function uses the average of inputs m_{ji} and the standard deviation of inputs σ_{ji} to determine the j_{th} linguistic term.

$$u_{A_i^j} = \exp\left(-\frac{(x_i - m_{ji})^2}{\sigma_{ji}^2}\right) \tag{1}$$

Layer 3: This is the rule-base layer. At this layer, each neuron represents one fuzzy logic rule. The activation functions of neurons are given by Eq. (2). The output of a layer 3 neuron is the result of r_{th} fuzzy logic rule.

$$u_r = \prod_{i=1}^{n} u_{A_i^j} \tag{2}$$

Layer 4: This layer is the defuzzification layer. It converts results of fuzzy logic rules from layer 3 into a numeric parameter value. There is only one neuron in this layer. It sums all results of fuzzy logic rules from Layer 3 and obtains the numeric parameter value. The w_r in function Eq. (3) is the link weight of r_{th} rule. It is adapted from the online learning process.

$$y = \sum_{r=1}^{M} w_r \cdot u_r \tag{3}$$

Layer 5: This is the output layer. It converts the outcome of defuzzification layer into the server parameter value $m_p(k)$ for each tier. In the implementation, there are two concurrency parameters for web and application tiers. Thus, there are two neurons in layer 5 generating the parameter values. The activation function of each node is given by Eq. (4). The weight r_p is determined by the workload distribution that was obtained through the online monitoring.

$$m_p(k) = y \cdot r_p \quad p = 1, 2. \tag{4}$$

We use a threshold to determine when the neural fuzzy controller stops the server parameter tuning process. A complete tuning iteration consists of two steps. First, the neural fuzzy controller generates server parameter values by forwardly feeding inputs through the five layers. Second, after new parameter values are applied to the multi-tier system, the neural fuzzy control evaluates performance changes. If the new server parameters cause performance improvement, there is no change on the parameters and weights as they are making the positive effect. But if the new server parameters cause performance degradation, the neural fuzzy control will amend its parameters and weights using online learning. After several tuning iterations, the neural fuzzy control stops parameter tuning when either error $e(k)$ or change in error $\Delta e(k)$ is less than the threshold. In our implementation, we set the threshold value to be 10%, same as that in work [2].

3) Learning Process of Neural Fuzzy Controller: Initially, the neural fuzzy controller only has a minimal skeleton structure. Then, the online learning process will add more fuzzification neurons and rule neurons, and dynamically refine the parameters and weights in the neural network. We achieve this by two processes: structure learning and parameter learning.

1. Structure Learning: The structure learning process decides when and how to add new membership function nodes (layer 2) and associated rule nodes (layer 3). We use Gaussian function (1) as the membership function. It has a certain recognition range of the input values. The range is determined by the mean and the standard deviation. If the input value has been recognized by a membership function, the output value of the Gaussian function is relatively high, which is notated as high firing strength. When a new input value comes in, the algorithm checks it with all pre-existing membership functions. If all get low firing strength, it means the value has not been recognized by the current neural network. Therefore, the network grows and new membership function nodes are added. We assign the mean of new node m_i^{new} to input x_i and the standard deviation σ_i^{new} to a predefined value.

The newly generated membership function nodes could be similar to an existing one. To eliminate it, we perform a similarity check before adopting the membership function nodes as a part of neural network. The similarity measurement method was originally proposed in [13]. Suppose $u_A(x)$ and $u_b(x)$ are two Gaussian functions and their means and standard deviations are m_A, m_B, σ_A, σ_B, respectively. The similarity of two Gaussian functions are measured as:

$$E(A, B) = \frac{|A \cap B|}{\sigma_A\sqrt{\pi} + \sigma_B\sqrt{\pi} - |A \cap B|}. \tag{5}$$

Assuming $m_A \geq m_B$,

$$\begin{aligned} |A \cap B| = &\frac{1}{2}\frac{h^2(m_B - m_A + \sqrt{\pi}(\sigma_A + \sigma_B))}{\sqrt{\pi}(\sigma_A + \sigma_B)} \\ &+ \frac{1}{2}\frac{h^2(m_B - m_A + \sqrt{\pi}(\sigma_A - \sigma_B))}{\sqrt{\pi}(\sigma_B - \sigma_A)} \\ &+ \frac{1}{2}\frac{h^2(m_B - m_A - \sqrt{\pi}(\sigma_A - \sigma_B))}{\sqrt{\pi}(\sigma_A - \sigma_B)} \end{aligned} \tag{6}$$

where $h(x) = max(0, x)$. In the case of scenario $\sigma_A = \sigma_B$,

$$|A \bigcap B|) = \frac{1}{2}\frac{h^2(m_B - m_A + \sqrt{\pi}(\sigma_A + \sigma_B))}{\sqrt{\pi}(\sigma_A + \sigma_B)}. \tag{7}$$

Only when the measured similarity is greater than a predefined threshold value, the new membership function node is added to the neural network. Since the newly generated membership function node is associated with a new fuzzy logic rule, we add a new node in layer 3 with the predefined output link weight.

2. Parameter Learning process: The parameter learning process is to refine the parameters and weights in the neural fuzzy control. This improves the performance of the neural fuzzy control under highly dynamic and bursty workloads. The parameter learning trains the neural network using back-propagation algorithm. The objective of the neural fuzzy control is equivalent to minimizing the function (8):

$$E = \frac{1}{2}({}^eT_{ref} - {}^eT_d)^2 = \frac{1}{2}(e(k))^2 \tag{8}$$

where ${}^eT_{ref}$ and ${}^eT_d$ are the objective and measured value of the performance metric, respectively. The learning algorithm recursively obtains a gradient vector in which each element is defined as the derivative of the function with respect to a parameter of the neural network. This is done by the chain rule. The method is referred to as the back-propagation learning rule, because the gradient vector is calculated in the direction opposite to the flow of the output of each node. The back-propagation parameter learning algorithm is described in the following.

Layer 5: The weights r_1 and r_2 are updated with monitored request arrival rates ΔR_1, ΔR_2. The back-propagation algorithm calculates output of layer 4 as:

$$y = \sum_{p=1}^{2} m_p(k) \cdot r_p. \tag{9}$$

Layer 4: The error term to be propagated is calculated as

$$\delta^{(4)} = -\frac{\partial E}{\partial y} = [-\frac{\partial E}{\partial e(k)}\frac{\partial e(k)}{\partial y}] = [-\frac{\partial E}{\partial e(k)}\frac{\partial e(k)}{\partial {}^eT_d}\frac{\partial {}^eT_d}{\partial y}]. \tag{10}$$

The link weight w_j is updated by:

$$\Delta w_r = -\eta_w \frac{\partial E}{\partial w_r} = -\eta_w \frac{\partial E}{\partial y}\frac{\partial y}{\partial w_r} = \eta_w \delta^{(4)} u_r \tag{11}$$

where η_w is the learning rate of the link weight. The link weights in layer 4 are updated by function Eq. (12).

$$w_r(k+1) = w_r(k) + \Delta w_r \tag{12}$$

Layer 3: At layer 3, the error term is calculated and propagated as described by function Eq. (13).

$$\delta_r^{(3)} = -\frac{\partial E}{\partial u_r} = [-\frac{\partial E}{\partial y}\frac{\partial y}{\partial u_r}] = \delta^{(4)} w_r \tag{13}$$

Layer 2: The error term is calculated by

$$\delta_{ji}^{(2)} = -\frac{\partial E}{\partial u_{A_{ji}}} = -\frac{\partial E}{\partial u_r}\frac{\partial u_r}{\partial u_{A_{ji}}} = \delta_r^{(3)} u_r. \tag{14}$$

The updating function for m_{ji} is

$$\Delta m_{ji} = -\eta_m \frac{\partial E}{\partial m_{ji}} = 2\eta_m \delta_{ji}^{(2)} \frac{(x_i - m_{ji})}{(\sigma_{ji})^2}. \tag{15}$$

The updating function for σ_{ji} is

$$\Delta m_{ji} = -\eta_\sigma \frac{\partial E}{\partial m_{ji}} = 2eta_\sigma \delta_{ji}^{(2)} \frac{(x_i - m_{ji})^2}{(\sigma_{ji})^3} \tag{16}$$

where η_m and η_σ are the learning-rate parameters of the mean and the standard deviation of the Gaussian function. The mean and standard deviation of the membership functions at layer 2 are updated using functions Eq. (17) and Eq. (18), respectively.

$$m_{ji}(k+1) = m_{ji}(k) + \Delta m_{ji} \tag{17}$$

$$\sigma_{ji}(k+1) = \sigma_{ji}(k) + \Delta \sigma_{ji} \tag{18}$$

B. A Reinforcement Learning based Approach

Reinforcement learning is a process of learning through interactions with an external environment. Recently people have adopted it as a methodology for resource management in autonomic computing [2], [21]. The server configuration tuning can be formulated as a finite Markov decision process. It consists of a set of states and several actions for each state. During the transition of each state, the learning agent perceives a reward defined by a function $R = E[r_{t+1}|S_t = s, a_t = a, s_{t+1} = s']$. The goal of the reinforcement learning agent is to develop the policy $\pi : S- > A$, which can maximize the cumulative rewards through iterative trial-and-error interactions.

In the context of applying reinforcement learning for maximizing the effective system throughput of multi-tier applications, we define the state space S and action set A as follow:

a) State Space: For the server parameter tuning, we define the state space S as the set of possible parameter values for each tier. A predefined adjustment value will determine what states will be in the state space. Therefore, the possible configurations are limited by the adjustment value. We represent the state space as a collection of state vectors:

$$s_i = (P_{1,tier1}, P_{2,tier1}, \ldots, P_{1,tier2}, \ldots, P_{n,tier_k}). \tag{19}$$

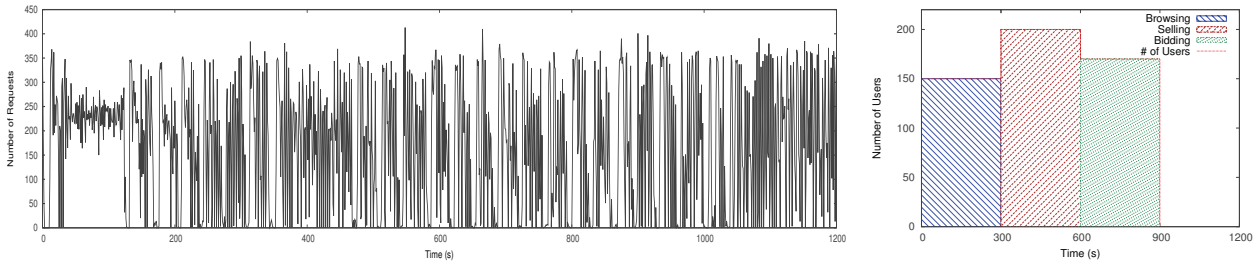
b) Action Set: We define three tuning actions for each parameter: keep, increase, and decrease. The potential action for each state a_i of one parameter is represented as a vector $P_{n,tier_k}(1,0,0)$. Each element in the action vector determines if one tuning action is taken or not-taken. The action set of one state is described as:

$$a_i^{increse} = P_{1,tier1}(1,0,0), \ldots, P_{n,tier_k}(0,0,0). \tag{20}$$

c) Q Value Learning: The Q value determines the action choice on each state. To learn the Q value, the learning agent should continuously update its estimation based on the state transition and reward what it receives. We use a temporal difference agent to implement the Q value learning because it is model independent and it updates Q values at each time step based on estimation. Therefore, the average Q value of an action a on state s, denoted as $Q(s,a)$, is refined by the function

$$Q_t(s_t,a_t) = Q_t(s_t,a_t) + a * [r_{t+1} + \gamma Q_{t+1}(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)] \tag{21}$$

in which s is the learning rate that facilitates the convergence to the true Q values in the presence of noisy or stochastic rewards and state transitions, and γ is the discount rate to guarantee the accumulated reward convergence in continuing tasks. Because the basic reinforcement learning would take many trial-and-error iterations to converge, we use training data that is collected offline to initialize the Q value table. The training can reduce the iterations needed for reinforcement learning to converge.



(a) A bursty workload.

(b) A step-change workload.

Fig. 5. Highly dynamic and bursty RUBiS workloads used in the experiments.

V. System Implementation

A. The Testbed

We implement a multi-tier web system in a university prototype data center. It consists of three racks of HP ProLiant BL460C G6 blade servers. Each server is equipped with 2-way Intel quad-core Xeon E5530 CPUs and 32GB memory. The blade servers are connected with 10 Gbps Ethernet. VMware vSphere 4.1 is used to for server virtualization.

We construct the web system with three VMs, Apache web server in the first, PHP application server in the second, and MYSQL database server in the third. We configure the web server and applications server with 2 VCPUs and 256 MB memory each. We set the CPU usage cap to 200 MHz for stationary workloads and 400 MHz for bursty and step-change workloads. We allocate another three VMs as clients to emulate different workloads. Both the web system and client VMs run Ubuntu server 10.04 with Linux kernel 2.6.35.

As many others in [2], [21], [11], [10], We use RUBiS [1], [22] e-transationcal benchmark application for multi-tier Internet applications. RUBiS provides a web auction application modeled in a similar way of ebay.com and an emulation client used as the workload benchmark. RUBiS characterizes the workload into three categories, seller, visitor, and buyer. They have different combinations of selling, browsing, and bidding requests. RUBiS client emulates user requests at different concurrent levels. To evaluate two automated server parameter tuning approaches, we use multiple clients to emulate the dynamics in Internet workloads. To provide the runtime performance monitoring, we modify the original RUBiS client to support reporting performance statistics of requests. Each parameter tuning iteration is executed every 30 seconds.

B. Workloads

We use three workloads with different densities: a stationary workload, a bursty workload, and a step-change workload. The stationary workload is generated to emulate 200 concurrent users. For the bursty workload, we implement a workload generator for RUBiS benchmark using the approach proposed in [17]. Figure 5(a) shows the bursty workload generated by changing the think time of each user. This workload emulates bursty workload in a 1200-seconds time span from 200 concurrent users with the average think time of 7 seconds.

TABLE I
Default vendor configuration of parameters.

Parameter	Value Range	Default Value
MaxClients	[50,600]	150
KeepAliveTimeout	[1,21]	15
MinSpareServers	[5,85]	5
MaxSpareServers	[15,95]	15

We record the trace of the bursty workload and reuse it for each experiment under bursty workloads.

To examine the adaptiveness of the neural fuzzy controller under highly dynamic workloads, we design a step-change workload that contains dynamics in both number of users and workload characteristics. Figure 5(b) shows the workload. Each time when the number of users changes, the workload mix also changes. At the 300_{th} second, the workload mix changes from browsing to selling. At the 600_{th} second, the workload mix changes from selling to bidding.

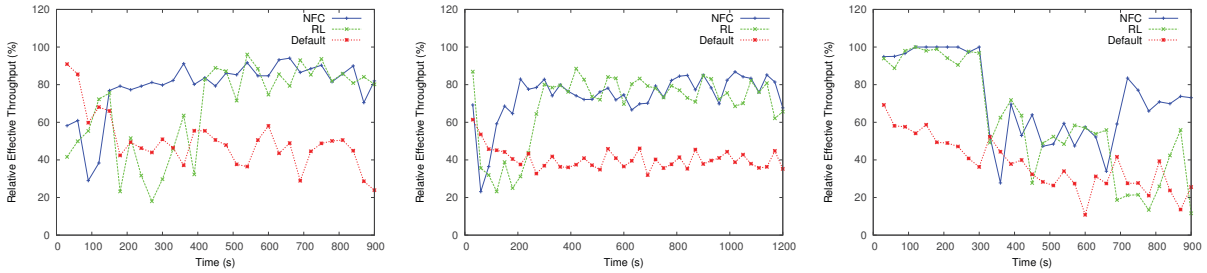
C. Performance Metrics

We conduct experiments using the effective throughput of the system and the average response time of requests as performance metrics. We use both the absolute effective throughput and the relative effective throughput. To demonstrate the agility of the two different automated server parameter tuning approaches, we compare their converging times. The converging time is the number of iterations before the configurable parameters reach their stable states.

VI. Performance Evaluation

A. Impact of Automated Tuning on Effective Throughput

We compare two automated server parameter tuning approaches, the neural fuzzy control based and the reinforcement learning based, with the default configuration provided by the vendor of servers. The default vendor configuration is shown in Table I. For the neural fuzzy control, we feed the measured effective system throughput directly into the control. The control starts with the default values of the server parameters. It generates new parameter values to reconfigure the server until the effective system throughput change meets the threshold-based target. The reinforcement learning based approach followed the same process. The default configuration remains as it is during the whole process.



(a) Under Stationary Workload. (b) Under Bursty Workload. (c) Under Step-change Workload.

Fig. 6. Performance of two automated server parameter tuning approaches compared to that of the default vendor configuration.

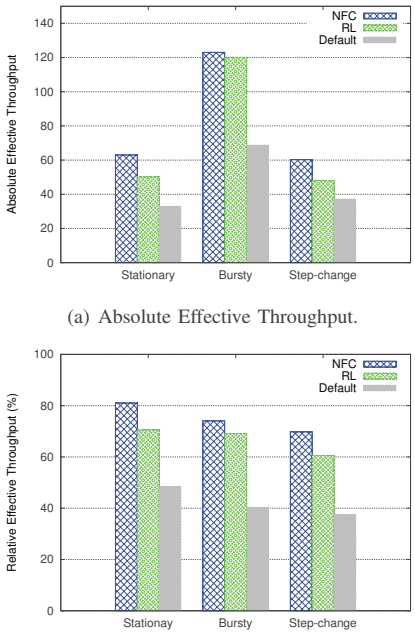
Figure 6 shows the relative effective throughput achieved by the three approaches under the stationary workload, bursty workload and step-change workload. Experimental results show that both the neural fuzzy control based and the reinforcement learning based server parameter tuning approaches can significantly improve the relative effective throughput. Under the stationary, bursty, and step-change workloads, the neural fuzzy control based approach achieves 67.5%, 83.3%, and 85.9% higher relative effective throughput than the default configuration does. The reinforcement learning based approach achieves 45.8%, 71.1%, and 61.3% higher relative effective throughput than the default configuration does.

Figure 7 shows the absolute effective throughput and the relative effective throughput due to the three approaches. Both automated parameter tuning approaches significantly improve the achieved performance due to the default configuration. The results demonstrate the significance of automated parameter tuning in performance improvement of complex multi-tier applications. Under the stationary and the step-change workloads, we observe that there are significant differences in achieved performance between the two automated approaches.

The absolute effective throughput is the number of requests that were responded within the time bounds. In RUBiS benchmark, the request generation rate is affected by the responsiveness of requests. That is, a new request will only be generated until the response of the former request belonging to the same session is returned. Thus, high response time will slow down the workload generation rate, which would result in fewer requests incoming to the system. When using different approaches, the number of total incoming requests indeed could be different. In this case, the absolute effective throughput cannot truly reflect the performance of different approaches. Therefore, in the following, we use the relative effective throughput as the major performance metric.

B. Comparison of Two Automated Tuning Approaches

In this section, we focus on the performance difference between the neural fuzzy control based and the reinforcement learning based approaches for improving the effective system throughput and minimizing the average response time. We also compare their converging times during the parameter tuning process. For the step-change workload, we compare their converging times for each workload stage separately.



(a) Absolute Effective Throughput.

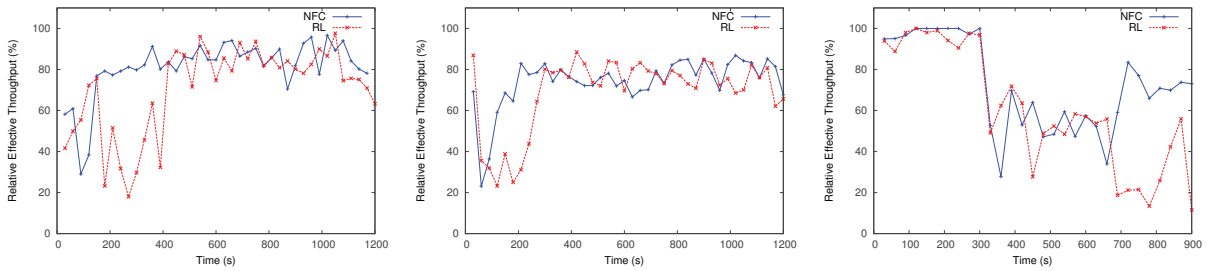
(b) Relative Effective Throughput.

Fig. 7. Performance comparison of three parameter configuration approaches.

1) Improving the Effective Throughput: Figure 8 shows the relative effective throughput of the neural fuzzy control and the reinforcement learning based approaches. Experimental results show that the neural fuzzy control based approach achieves on average 14.8%, 7.1%, and 19.5% higher relative effective throughput than the reinforcement learning based approach under stationary, bursty, and step-change workloads, respectively.

Table II shows the comparison of the converging time due to the two automated tuning approaches. Under the stationary workload, the neural fuzzy control based approach converges 3 times faster than the reinforcement learning based approach. Under the bursty workload, the neural fuzzy control based approach converges about 2.9 times faster than the reinforcement learning based approach.

We note that performance variations during the server



(a) Under Stationary Workload. (b) Under Bursty Workload. (c) Under Step-change Workload.

Fig. 8. Relative effective throughput of the neural fuzzy control and the reinforcement learning based server parameter tuning approaches.

parameter tuning process are significantly different due to the two automated approaches. As Figures 8(a) and 8(b) show, the performance due to the reinforcement learning based approach varies significantly during the initial parameter tuning process (the first 400 seconds). That results in significantly lower effective throughput compared to that due to the neural fuzzy control based approach. There are two major reasons that the reinforcement learning based approach suffers performance penalty from high variations. First, a parameter value change is upper bounded by the pre-defined adjustment value. This limits the capability of the approach in agilely adjusting sever parameter values. Second, in reinforcement learning, the Q-table is initialized by the offline training data. The training outcome may not accurately describe the complex interaction between parameter tuning and performance outcome.

Figure 9 shows the neural fuzzy control based approach achieves better average relative effective throughput than the reinforcement learning based approach. Under the step-change workload, the improvement is particularly significant. The neural fuzzy control based approach doubles the achieved relative effective throughput. This is due to the fact that the approach can agilely tune server parameters for performance improvement. Table II shows the convergence time of two automated parameter tuning approaches. Note that at the bidding stage, the reinforcement learning based approach does not converge during the experimental period (600_{th} second to 900_{th} second). This results in significantly lower relative effective throughput compared that due to the neural fuzzy control based approach. As Figure 8(c) shows, the reinforcement learning based approach suffers from high variations in the system stability due to the slow convergence under the highly dynamic step-change workload.

Table III gives the standard deviations of the achieved relative effective throughput due to the two automated parameter tuning approaches. Results show that compared to the reinforcement learning based approach, the neural fuzzy control based approach achieves much better system stability under various workloads. This is due to its fast online learning and the use of a quantitative output value for each parameter tuning iteration. Note that the improvement in system performance stability is amortized by the long experimental time period (1200 seconds).

TABLE II
CONVERGING TIME OF TWO AUTOMATED TUNING APPROACHES.

	NFC	RL
Stationary Workload	6	18
Bursty Workload	7	20
step-change Workload - Browsing Stage	1	4
step-change Workload - Selling Stage	5	8
step-change Workload - Bidding Stage	7	10 (did not converge)

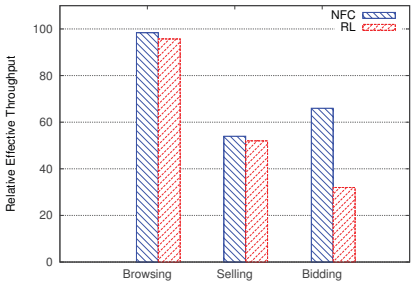
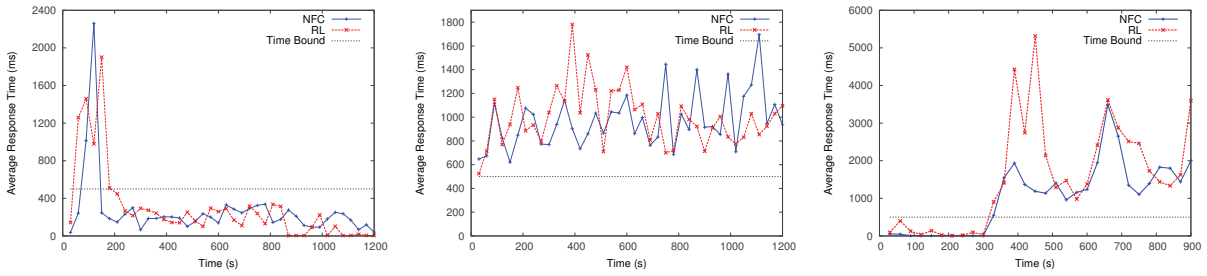


Fig. 9. Average relative effective throughput under the step-change workload.

2) Minimizing Average Response Time: Figure 10 shows the average response time of the multi-tier application due to the neural fuzzy control based approach and the reinforcement learning based approach. For the stationary and bursty workloads, the achieved average response time of the two automated approaches are close. Under the stationary workload, both approaches are able to decrease the average

TABLE III
STANDARD DEVIATION OF THE EFFECTIVE THROUGHPUT.

	NFC	RL	Difference
σ of RET under stationary workload	0.1391	0.2144	54.1%
σ of RET under bursty workload	0.1226	0.1819	48.4%
σ of RET under step-change workload browsing stage	0.0219	0.0373	70.3%
σ of RET under step-change workload selling stage	0.1145	0.1194	4.3%
σ of RET under step-change workload bidding stage	0.1430	0.1805	26.2%



(a) Under Stationary workload. (b) Under Bursty workload. (c) Under Step-change workload.

Fig. 10. The average response time due to the two automated server parameter tuning approaches.

TABLE IV
STANDARD DEVIATION OF EFFECTIVE THROUGHPUT.

	NFC	RL	Difference
σ of bursty workload	357.9	407.3	13.8%
σ of bursty workload	972.8	999.2	2.7%
σ of step-change workload browsing stage	18.40	116.3	632.1%
σ of step-chang workload selling stage	364.4	1519.3	416.9%
σ of step-chang workload bidding stage	709.1	832.7	117.4%

response time below the predefined time bound. Note that the bound is needed for the reinforcement learning approach to generate the reward. Under the bursty workload, both approaches cannot assure that the average response time below the predefined time bound. But both maintain the average response time close to the bound. The main difference between the two approaches is the convergence time for minimizing the average response time. Figure 11 shows that the neural fuzzy control based approach converges much faster than the reinforcement learning based approach.

Figure 12 shows the average response time comparison between the two automated parameter tuning approaches. Under the step-change workload, the neural fuzzy control based approach achieves more than 30% lower average response time than the reinforcement learning based approach does.

Figure 13 illustrates significant differences in the achieved average response time by the neural fuzzy control based approach and by the reinforcement learning based approach during three stages of the step-change dynamic workload.

3) Analysis: The experiments have demonstrated that the neural fuzzy control based approach outperforms the reinforcement learning based approach for both maximizing the relative effective throughput and minimizing average response time. The differences in performance and agility are mainly due to the weaknesses of reinforcement learning used for automated server parameter tuning.

Essentially, reinforcement learning is a decision making process. It does not directly generate the actual parameter value for server parameter configuration. It needs a predefined adjustment value for each parameter tuning iteration. Experimental results show that during the server parameter

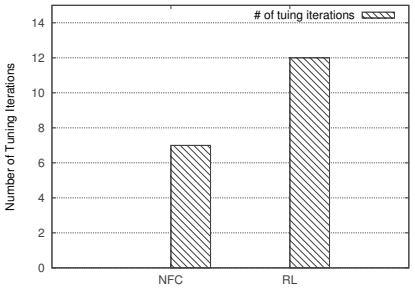


Fig. 11. Converging time in minimizing the average response time.

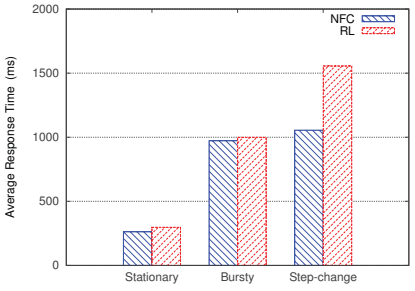


Fig. 12. Average response time comparison.

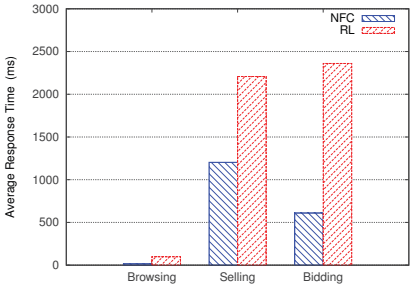


Fig. 13. Average response time under the three-stage step-change workload.

tuning process, some configurations that the reinforcement learning based approach chooses is not as effective as those chosen by the neural fuzzy control based approach. This is due to the fact that the predefined adjustment value by the reinforcement learning based approach will limit the number of states in its state space. This will make it possible that some effective parameter configurations are not reachable by the reinforcement learning based approach. On the other hand, the neural fuzzy control based approach does not have such a constraint on the reachable configurations.

At each iteration, the reinforcement learning based approach can only move from one state to another one. Its converging speed is dependent on the size of the state space. With the same range of parameter tuning, the smaller the adjustment value is, the larger the state space is, which often leads to longer converging time. Finding a good adjustment value for the reinforcement learning based approach is a difficult problem. In contrast, the neural fuzzy control based approach is designed to search all possible states and generate the ideal parameter configuration value at runtime.

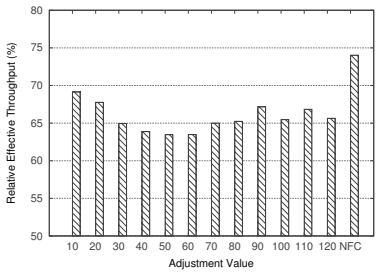
Moreover, the reinforcement learning based approach is initialized by the training data and updated during the learning process. During the server parameter tuning process, the relation among workloads, parameters, and resulted performance is very complex and non-linear. The training result does not necessarily describe the correct interaction between the performance and parameters. Therefore, the reinforcement learning based approach can be misled by the training data.

C. Impact of Different Pre-defined Adjustment Values

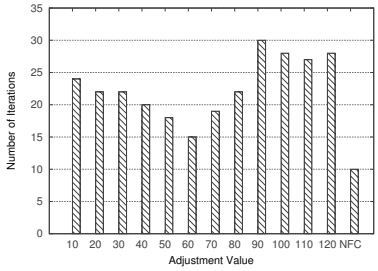
For reinforcement learning, the choice of the pre-defined adjustment value is very important. The value affects the parameter tuning precision and the converging speed. It also determines the parameter state space size. We study the impact of different pre-defined adjustment values on the performance of the reinforcement learning based approach under the bursty workload. We vary the adjustment value from 10 to 120 to build the parameter state space. We measure both the relative effective throughput and the convergence time.

Figure 14 shows the impact of the pre-defined adjustment value on the relative effective throughput and the convergence time. For reference, the results due to the neural fuzzy control based approach are included at the end of each plot. First, the achieved relative effective throughput and the convergence rate are a tradeoff when using different pre-defined adjustment values. Note that the ideal value for the maximum effective throughput is not the same as the ideal value for the smallest convergence time. Second, a smaller adjustment value will provide more states in the state space, which can provide opportunities for the reinforcement learning based approach to achieve better performance. But more states in the state space also increase the number of iterations for converging.

From the results in Figure 14(a), we observe that when the pre-defined adjustment value equals to 10, the reinforcement learning based approach achieves its highest effective throughput among all scenarios. However, it is still about 7% lower that the effective throughput achieved by the neural fuzzy control based approach. Meanwhile, Figure 14(b) shows the convergence time of the reinforcement learning based approach is 2.5 times of that due to the neural fuzzy control based approach. When the pre-defined adjustment value is 60, the number of iteration required by the reinforcement learning based approach decreases to its minimum number 15, but it is still 50% higher than that due to the neural fuzzy control based approach. Under this scenario, the effective throughput of the reinforcement learning based approach is 15% lower than that due to the neural fuzzy control based approach.



(a) On the relative effective throughput.



(b) On the convergence time.

Fig. 14. Impact of pre-defined adjustment values on performance.

TABLE V
PERFORMANCE-CRITICAL SERVER PARAMETERS.

Parameter	Value Range	Step Change Size
MaxClients	[50,600]	10
KeepAliveTimeout	[1,21]	1
MinSpareServers	[5,85]	2
MaxSpareServers	[15,95]	2

D. Selection of Server Parameters for Tuning

There are many parameters in a server. In the performance evaluation, we have highlighted the significant performance impact due to automatically tuning the parameter $MaxClients$. But we have also studied how tuning of other parameters would affect the system performance.

We have conducted a series of experiments by tuning numerous parameters that have been confirmed highly performance relevant. Table V gives the ranges and step change sizes of those server parameter values.

The experiment is done by step-by-step tuning of one parameter while others maintain the default values under the bursty workload. We use the relative effective throughput as

TABLE VI
STANDARD DEVIATIONS FOR DIFFERENT PARAMETER.

	KeepAlive Timeout	MaxClients	MaxSpare Servers	MinSpare Servers
Standard Deviation	0.036	**0.108**	0.028	0.033

the performance metric. We measure the impact of different parameters on performance by the use of the standard deviation of the relative effective throughput. Experimental results in Table VI show that parameter *MaxClients* dominates the performance impact.

VII. CONCLUSIONS

This paper tackles the important but challenging problem of automated server parameter tuning in virtualized environments. We have proposed an automated and agile approach that integrates the strengths of fast online learning and control to maximize the effective system throughput of multi-tier Internet applications. We have implemented the approach on a testbed of virtualized HP ProLiant blade servers. Experimental results based on multi-tier benchmark applications have demonstrated that the proposed approach significantly outperforms a reinforcement learning based approach for both improving the effective throughput and minimizing the average response time. We have analyzed the weaknesses of the reinforcement learning based approach due to the use of a predefined adjustment value and inaccurate training of Q-table. Our developed neural fuzzy control based approach avoids the weaknesses, providing the self-management capability for automated and agile server parameter tuning under highly dynamic and bursty workloads. It is also applicable to single-tier server architecture.

Our future will be coordinate server parameter tuning with VM capacity planning in Cloud-computing datacenters.

Acknowledgement

This research was supported in part by U.S. National Science Foundation CAREER Award CNS-0844983. The authors thank the NISSC for providing blade server equipments for conducting the experiments.

REFERENCES

[1] C. Amza, A. Chanda, A. Cox, S. Elnikety, R. Gil, K. Rajamani, W. Zwaenepoel, E. Cecchet, and J. Marguerite. Specification and implementation of dynamic Web site benchmarks. In *Proc. IEEE Int'l Workshop on Workload Characterization (WWC)*, pages 3 – 13, 2002.

[2] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online Web system auto-configuration. In *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS)*, 2009.

[3] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content Web servers. In *Proc. IEEE Int'l Conference on Autonomic Computing (ICAC)*, 2006.

[4] I.-H. Chung and J. K. Hollingsworth. Automated cluster-based Web service performance tunning. In *The IEEE International Symposium on High-Performance Distributed Computing*, 2004.

[5] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaihk, and M. Surendra. Controlling quality of service in multi-tier Web applications. In *Proc. IEEE Int'l Conference on Distributed Computing Systems (ICDCS)*, 2006.

[6] M. C. Huebscher and J. A. McCann. A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys*, 40(7):1–28, 2008.

[7] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *Proc. IEEE Int'l Conference on Autonomic Computing (ICAC)*, 2008.

[8] M. Korupolu, A. Singh, and B. Bamba. Coupled placement in modern data centers. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[9] P. Lama and X. Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *Proc. IEEE/ACM Int'l Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 151–160, 2010.

[10] P. Lama and X. Zhou. aMOSS: Automated multi-objective server provisioning with stress-strain curving. In *Proc. IEEE Int'l Conference on Parallel Processing (ICPP)*, pages 345–354, 2011.

[11] P. Lama and X. Zhou. PERFUME: Power and performance guarantee with fuzzy mimo control in virtualized servers. In *Proc. IEEE/ACM Int'l Workshop on Quality of Service (IWQoS)*, pages 1–9, 2011.

[12] P. Lama and X. Zhou. Efficient server provisioning with control for end-to-end delay guarantee on multi-tier clusters. *IEEE Transactions on Parallel and Distributed Systems, 19 pages*, 23(1), 2012.

[13] C. Lin and C. S. G. Lee. Real-time supervised structure/parameter learning for fuzzy neural network. In *Proc. IEEE Int'l Conference on Fuzzy Systems*, pages 1283–1291, 1992.

[14] X. Liu, L. Sha, and Y. Diao. Online response time optimization of apache Web server. In *Proc. Int'l Workshop on Quality of Service (IWQoS)*, 2003.

[15] X. Meng, C. Isci, J. Kephart, L. Zhang, and E. Bouillet. Efficient resource provisioning in compute clouds via vm multiplexing. In *Proc. Int'l Conference on Autonomic Computing (ICAC)*, 2010.

[16] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in multi-tier applications: Symptoms, causes, and new models. In *Proc. ACM/IFIP/USENIX Int'l Middleware Conference (Middleware)*, 2008.

[17] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Injecting realistic burstiness to a traditional client-server benchmark. In *Proc. IEEE Int' Conference on Autonomic Computing (ICAC)*, 2009.

[18] J. Moses, R. Iyer, R. Illikkal, S. Srinivasan, and K. Aisopos. Shared resource monitoring and throughput optimization in cloud-computing datacenters. In *Proc. of IEEE Int'l Symposium on Parallel and Distributed Processing (IPDPS)*, 2011.

[19] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[20] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proc. of the EuroSys Conference (EuroSys)*, pages 13–26, 2009.

[21] J. Rao, X. Bu, C. Xu, L. Wang, and G. Yin. Vconf: A reinforcement learning approach to virtual machines auto-conguration. In *Proc. IEEE Int'l Conference on Autonomic Computing Systems (ICAC)*, 2009.

[22] RUBiS. Rice university bidding system. http://www.cs.rice.edu/CS/Systems/DynaServer/rubis.

[23] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy. Autonomic mix-aware provisioning for non-stationary data center workloads. In *Proc. IEEE Int'l Conference on Autonomic Computing (ICAC)*, pages 21–30, 2010.

[24] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proc. IEEE Int'l Conference on Autonomic Computing (ICAC)*, 2006.

[25] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Trans. on Autonomous and Adaptive Systems*, 3(1):1–39, 2008.

[26] M. Wang, X. Meng, and L. Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. In *Proc. IEEE INFOCOM*, 2011.

[27] Q. Wang, S. Malkowski, D. Jayasinghe, P. Xiong, C. Pu, Y. Kanemasa, M. Kawaba, and L. Harada. The impact of software resource allocation on n-tier application scalability. In *Proc. of IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2011.

[28] Y. Zhang, W. Qu, and A. Liu. Automatic performance tuning for J2EE application server systems. *Proc. of Web Information System Engneering (WISE)*, pages 520–527, 2005.

[29] W. Zheng, R. Bianchini, and T. Nguyen. Automatic configuration of Internet services. In *Proc. of the EuroSys Conference*, 2007.