Coordinated Self-configuration of Virtual Machines and Appliances using A Model-free Learning Approach

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

Abstract—Cloud computing has a key requirement for resource configuration in a real-time manner. In such virtualized environments, both virtual machines (VMs) and hosted applications need to be configured on-the-fly to adapt to system dynamics. The interplay between the layers of VMs and applications further complicates the problem of cloud configuration. Independent tuning of each aspect may not lead to optimal system wide performance. In this paper, we propose a framework, namely CoTuner, for coordinated configuration of VMs and resident applications. At the heart of the framework is a model-free hybrid reinforcement learning (RL) approach, which combines the advantages of Simplex method and RL method and is further enhanced by the use of system knowledge guided exploration policies. Experimental results on Xen-based virtualized environments with TPC-W and TPC-C benchmarks demonstrate that CoTuner is able to drive a virtual server cluster into an optimal or near-optimal configuration state on the fly, in response to the change of workload. It improves the systems throughput by more than 30% over independent tuning strategies. In comparison with the coordinated tuning strategies based on basic RL or Simplex algorithm, the hybrid RL algorithm gains 25% to 40% throughput improvement. Moreover, the algorithm is able to reduce SLA violation of the applications by more than 80%.

Index Terms—Cloud Computing, Reinforcement Learning, Autonomic Virtual Machine Configuration, Autonomic Application Configuration.

1 Introduction

Cloud computing, unlocked by virtualization, emerges as an increasingly important infrastructure-as-service computing paradigm. In such virtualized environments, system performance heavily depends on virtual machine (VM) configuration and parameter settings of their resident applications. For the provisioning of on-demand services, VMs need to be re-configured to improve resource utilization under the constraints of applications' service level agreement (SLA). Workload dynamics also requires dynamic tuning of performance-critical application parameters (e.g. `MaxThreads` in Tomcat server).

Auto-configuration is a non-trivial task. Heterogeneous resident applications and performance uncertainty caused by VM interferences make it challenging to generate good VM configurations [16], [10], [21]. For multi-component applications, like Apache/Tomcat/MySQL web applications, interactions between the components may even spread configuration errors into the entire system. Performance optimization of individual components does not necessarily lead to overall system performance improvement [6]. Moreover, the configurations of VMs and resident applications interplay with each other. For example, a large scale VM makes it possible to support a high `MaxThreads` setting; a small `MaxThreads` in a large scale VM would lead to low resource utilization. Independent tuning of each layer of configurations does not necessarily lead to overall optimal system performance. The objective of this study is to develop an automatic configuration strategy for both layers of configurations in a coordinated manner.

There were many studies devoted to autonomic configurations of VM resources or application parameters. For example, in [18], [17], [27], [26], [15], feedback control approaches had achieved notable successes in adaptive virtual resource allocation and web application parameter tunings. However, such control approaches rely on explicit models of target systems. Design and implementation of accurate models of complex systems are highly knowledge-intensive and labor-intensive. Other studies formulated the problem as a combinatorial optimization. In [30], [32], [6], [33], optimization approaches, like hill-climbing and Simplex were experimented with to automate the tuning process of web applications. These heuristic approaches could be highly efficient, but tend to trap the system into local optimum.

Reinforcement learning (RL) provides a knowledge-free trial-and-error methodology in which a learner tries various actions in numerous system states and learns from the consequences of each action. RL can potentially generate decision-theoretic optimal policies in dynamic environments [22]. It offers two major

advantages. First, RL does not require explicit model of either the managed system or the external process, like incoming traffic. Second, RL targets the maximization of long term reward instead of immediate performance feedback. It considers the possibility that a current decision may have delayed consequences on both future reward and future state. It can potentially deal with the delayed effect and local optimum problem. Recent studies showed the feasibility of RL approaches in many applications. Our own works [21], [20], [5] demonstrated the applicability of RL in auto-configuration of virtual resource and web applications, respectively.

While RL provides many potential benefits in automatic management, there are challenges in practice. RL suffers from poor scalability in problems with a large state space that grows exponentially with the state variables (i.e. configuration parameters of both VMs and applications). Moreover, in the absence of domain knowledge, the initial performance achieved by RL during online training may be extremely poor. It requires a RL-based auto-configuration stratergy to start with a fairly large amount of exploration actions before generating a stable policy. Normally, exploration involves a process of random action selection, which may even lead to performance degradation in a short run.

In this paper, we present a novel hybrid approach that combines the advantages of Simplex and RL method to address the above practical limitations. Instead of conducting RL search in the whole configurable state space, we first use Simplex method to reduce search space to a much smaller but "promising" state set. To avoid performance degradation caused by random exploration, we enhanced RL agents with system knowledge guided exploration policy, which uses the information of CPU and memory utilization to guide the online configuration process. Previous RL-based configuration work including [21], [5], [24] employed pre-learned performance models to address scalability and poor initial performance issues. Building an accurate performance model requires much human effort and domain knowledge, especially for the coordinated configuration involving the interplay between VM and application configurations. In contrast, our hybrid approach does not require prelearned performance models and is more suitable for high complex and dynamic systems. We summarize contributions of this paper as follows:

(1) **Model-free hybrid reinforcement learning algorithm**. We develop a model-free hybrid reinforcement algorithm for online configurations and reconfigurations of VM resources and application settings. This approach combines the advantages of RL method and Simplex method, with the enhancement of system knowledge-guided exploration policies. The hybrid approach significantly reduces the search space and improves performance in the exploration stage.

(2) **A coordinated tuning framework**. The framework facilitates an automatic tuning of VM resource allocations and resident application parameter settings in the presence of cloud dynamics. It is able to drive the system into an optimal (or near optimal) state within tens of interactions. Unlike previous auto-configuration studies that only considered either virtual resource management or application parameter tuning, our framework should be the first approach towards coordinated auto-configurations of both VM resource and running appliances in clouds, dealing with the interplay between them.

(3) **Design and implementation of CoTuner**. Our prototype implementation of the coordinated tuning framework, namely CoTuner, demonstrated its effectiveness in an Xen-based virtualized environment. With heterogeneously consolidated applications, including TPCW and TPCC, CoTuner is able to adapt VM resource allocation and appliance parameter settings to cloud dynamics in a coordinated way. It improves system throughput by more than 30% over independent tunings. In comparison with the coordianted tuning strategies that is only based on basic RL or Simplex algorithm, our hybrid algorithm gains 25% to 40% throughput improvement. Moreover, the algorithm is able to reduce the SLA violations of all applications by more than 80%. We also demenstrated the effectiveness of CoTuner in a large virtualized environment with 100 VMs.

The rest of the paper is organized as follows. Section 2 discusses about the challenges of coordinate auto-configuration. Section 3 presents the overview of the CoTuner framework. Section 4 shows the design of our hybrid RL algorithm. Evaluation methodology and settings are given in Section 5. Section 6 shows experimental results. Related work is discussed in Section 7. Section 8 concludes the paper with remarks on limitations and possible future work.

2 Challenges of Coordinated Configuration

Automatic configuration in virtual environment is a non-trivial task. Heterogeneous resource demands and interference among sharing hardware VMs make it challenging to generate optimal VM configuration. The resident applications require on-the-fly tuning to adapt to dynamic workload and allocated resources. Detailed discussions about the challenges of VM and application auto-configurations could be found in [21], [5], respectively. In this section, we put emphasis on the challenges of coordinated auto-configuration. We illustrate why the coordinated configuration strategy is necessary through simple experiments.

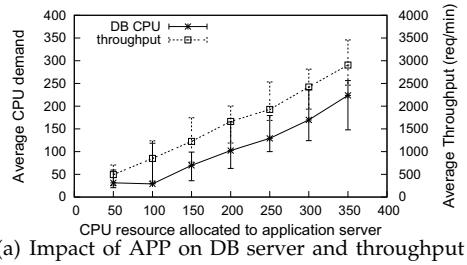
2.1 Interplay between VM and Application Configurations

In this experiment, we demonstrate interactions between the two levels of configurations using two-tier TPCW application. TPCW mimics a transactional web system with an application server deployed in the front and a database server in the backend. Each server was running within one VM and the two VMs were deployed on different physical machines; see Section 5 for the details of the applications and experimental settings. Tomcat server contains a key performance parameter, `MaxThreads`, which sets the maximum number of requests to be served simultaneously. A too high setting would lead to resource contention but a conservative setting would cause resource under-utilized. `MaxThreads` should be configured according to the capacity of hosting machine. To some extent, it determines the resource demand of the web application.

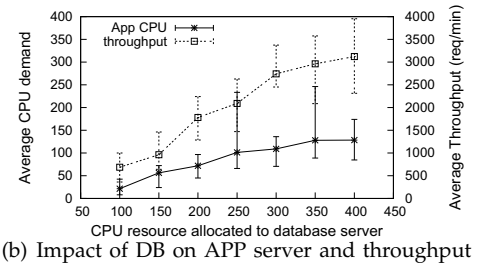
Figure 1(a) and Figure 1(b) show the average resource (CPU) demand of the two tiers due to different `MaxThreads` settings under various workloads in terms of the number of clients. Figure 1(c) shows the corresponding system throughputs. CPU resource is expressed in percentage of one physical CPU: 100 means one physical CPU and 50 means half a CPU. From the figures, we can see that the CPU demand and the throughput are insensitive to parameter `MaxThreads` in light loaded case due to small number of concurrent requests. Both of them increase with the `MaxThreads` value in medium and heavy loaded cases. The CPU demand and the throughput saturate at certian levels after `MaxThreads` reach certain values because the setting have exceeded the maximum concurrent requests. The impact of `MaxThreads` on resource demand suggests that this parameter may affect the VM configuration decisions considerably. The system performance would also vary as application and VM configurations interplay with each other.

2.2 Coordination in Virtual Server Clusters

Components of a multi-tier application may be distributed over multiple physical machines in the form of virtual server cluster. Such related physical servers require a coordinated configuration strategy due to interactions among different tiers. In this experiment, we still use TPCW application to demonstrate cross-tier interplay and its impact on system performance. The same experimental setup was used as Section 2.1. Application tier and database tier were running on two VMs, which were deployed on different physical servers. Resource allocated to one tier was restricted and increased step by step. Figure 2(a) shows that when more resource allocated to application server, not only the system throughput but also the database resource demand get increased. Figure 2(b) also shows the similar result as database resource allocation is restricted. These observations suggest virtual resource



(a) Impact of APP on DB server and throughput



(b) Impact of DB on APP server and throughput

Fig. 2. Interactions of application (APP) and database (DB) server configurations.

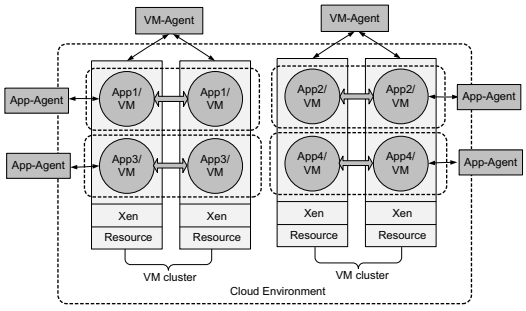
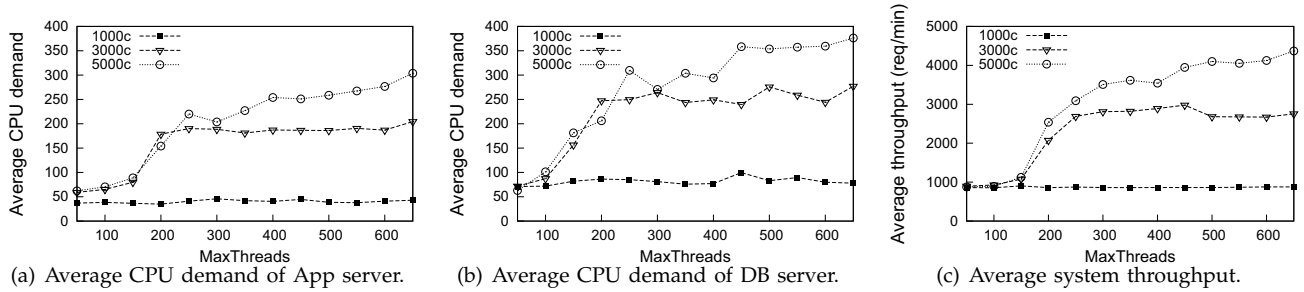


Fig. 3. Architecture of CoTuner framework.

configuration of one tier would greatly affect the resource demand of other tiers. The bottleneck may keep shifting among different tiers if the balanced resource allocation could not be achieved. For example, if we increase the application server's CPU resource from 100 percentage to 300 percentage, system performance improvement was expected. But this may not be the case due to bottleneck shift. From Figure 2(a), we can see that the resource (peak) demand of database also increase from 100 percentage to 250 percentage as well. If the allocated CPU resource to database server was still 100 percentage, the extra workload coming from front tier may drive the database overloaded. Figure 2(b) shows that when allocated CPU was 100 for database, system throughput was 60% lower than that when allocated CPU was 250 percentage. Therefore, performance optimization of individual tiers does not lead to overall system performance improvement. It is necessary to coordinately configure all involved physical servers instead to considering them as independent optimization components.



(a) Average CPU demand of App server. (b) Average CPU demand of DB server. (c) Average system throughput.

Fig. 1. Average resource demand and system performance due to different `MaxThreads` settings.

3 Overview of CoTuner

CoTuner is designed as a coordinated auto-configuration framework for performance optimization in cloud computing. Figure 3 shows its architecture. In such a virtualized environment, VMs may interfere with each other due to resource contention or dependence in multi-tier applications. Such related VMs form a VM cluster, in which VMs should be configured coordinately. The VMs in each cluster may be placed in the same or different physical machines. CoTuner consists of two types of agents: VM-Agent and App-Agent. They are in charge of virtual resource configuration and application configuration, respectively.

Each VM-Agent controls all the VMs within one VM cluster. It keeps monitoring the performance of each VM and adapting their configurations to the environment dynamics online. It takes advantage of the control interface provided by Dom0 (in a Xen environment) to control the configuration of individual VMs. Reconfiguration actions take place periodically based on a predefined time interval. At each iteration, VM-Agent receives performance feedback and uses that information to update the configuration policy. The objective of VM-Agent is to drive the system into an optimal configuration state in terms of throughput, utilization, or any other application-level utility functions.

Each application has its own App-Agent, with different app-specific objectives. It monitors the performance of appliances belonging to the same application and refines their configurations through interactions with the system in a way similar to VM-Agent. During the configuration process, it is trying to tune application parameters to meet the SLA requirement.

All the agents are designed as standalone daemon residing in a dedicated management server. They collect system informations through TCP sockets. Coordinated configuration decisions made by VM-Agent and App-Agent offers an opportunity of tradeoff between the system wide utilization and application-specific performance.

4 Hybrid RL Approach

RL provides a knowledge-free methodology for automatic managment. It requires no explicit system model and could effectively deal with local optimum problem. Meanwhile, there are challenges in applying RL to practice. It suffers from poor scalability and uncertain initial performance. In this section, we present a novel hybrid RL approach. The agents take the advantages of RL to make configuration decisions. They also enhanced by the use of Simplex-based space reduction and guided exploration policy.

4.1 RL-based Configuration Decision Making

Reinforcement learning refers to a trial-and-error learning process whereby an agent can learn to make good decisions through a sequence of interactions with the managed environment [22]. Under the auto-configuration context, the interaction consists of observing current configuration state, selecting valid configuration action and evaluating an immediate reward due to the action in given state. It does not require explicit model of either the managed system or the external environment, like incoming traffic.

A RL problem is often modeled as a finite Markov Decision Process (MDP). An MDP can be formulated with a set of state $\mathcal{S}$, a set of actions $\mathcal{A}$, an immediate reward function $R_a(s,s')= E(r_{t+1}|s_t = s, s_{t+1} = s', a_t = a)$ and a state transition probability function $P_a(s,s')= Pr(s_{t+1} = s'|s_t = s, a_t = a)$. At each time interval t, the learning agent perceives it current state $s_t \in \mathcal{S}$ and selects a valid action $a_t \in \mathcal{A}$. The action selection decision is determined not only by the immediate reward, but also by the future rewards the following states would yield. The "goodness" of an action in a given state is measured by a value function $Q(s,a)$, which estimates the future accumulated rewards by taking this action:

$$Q(s,a) = E(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a),$$

where $0 < \gamma < 1$ is a discount factor helping $Q(s,a)$'s convergence. The optimal value function $Q^*(s,a)$ is unique and can be defined as the solution to the following equation:

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} P_a(s,s')(R_a(s,s') + \gamma \max_{a'} Q^*(s',a')),$$

where s' and a' are the next state and action. Because

configuration decision is based on such long term rewards, RL agent is able to deal with delayed effect and local optimum problems.

The output of RL is a policy π that maps the system states to the best actions. The optimal policy π^* achieves the maximal expected return from any initial state, as defined below:

$$\pi^*(s) = \arg\max_a (R_a(s,s') + \gamma \sum_{s' \in \mathcal{S}} P_a(s,s') Q^*(s',a')).$$

Following the optimal policy, given state s, the best action should be the one that maximizes the sum of the immediate reward and the expected discounted reward of the next state s'. During each iteration, RL agent selects action according to current policy and observes reward feedback. The new reward is used to update $Q(s,a)$. Current policy π is improved based on the updated $Q(s,a)$. Both of the value function and policy would converge to their optimal value after sufficient interactions.

In this work, we formulate coordinated configuration task as a RL problem. For the VM-Agent, the state is defined as the virtualized resource configuration of all the VMs within one VM cluster. For the App-Agent, the state is defined as all the configurable parameter settings of its corresponding application. States defined on system configuration are deterministic in that $P_a(s,s') = 1$, which simplifies the RL problem. They are also fully observable to RL agents. For each configurable parameter, possible operation can be either *increase*, *decrease* or *nop*. The actions for the RL agents are defined as the combinations of the operations on each parameter.

Immediate reward is defined to reflect the overall system performance. Denote G and T for throughput and response time of an application, G_{SLA} and T_{SLA} for the corresponding service level agreements, and C_{thrpt} and C_{rspt} for the penalty of SLA violation. For App-Agent, the immediate reward r is defined based on the performance of the corresponding application as follows:

$$r = \frac{G}{G_{SLA}} + \frac{T_{SLA}}{T} - C_{thrpt} - C_{rspt},$$

where

$$C_{thrpt} = \begin{cases} w_{thrpt} * \frac{G_{SLA}}{G} & \text{if } G < G_{SLA}; \\ 0 & \text{otherwise,} \end{cases}$$

$$C_{rspt} = \begin{cases} w_{rspt} * \frac{T}{T_{SLA}} & \text{if } T > T_{SLA}; \\ 0 & \text{otherwise,} \end{cases}$$

where w_{thrpt} and w_{rspt} are the weights used to control the impact of SLA violations on the reward. In this work, both of w_{thrpt} and w_{rspt} were set to 10. Whenever SLA violation happens, a huge penalty will be added to reward as a feedback. Recall that VM-Agent controls all the VMs within a VM cluster. Its immediate reward of a VM cluster with n applications is defined as a weighted sum of the performance of all the resident applications:

$$r = \sum_{i=1}^{n} w_i * r_i,$$

where $w_i < 1$ and $\sum_1^n w_i = 1$

For RL agent, finding an optimal policy is equivalent to generating the optimal value function $Q^*(s,a)$ for each state. During the learning, $Q(s,a)$ need to keep updating until its value sufficiently approximates $Q^*(s,a)$. In this work, we employed *temporal-difference* (TD) method for value function update:

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

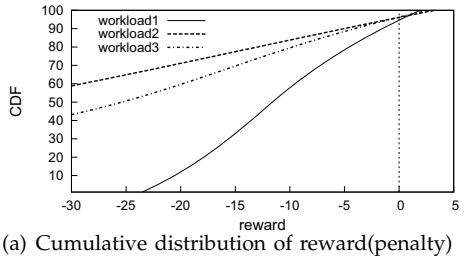
where α is a learning rate parameter that facilitates convergence in the presence of noisy or stochastic transitions. The advantages of TD over other methods (e.g. Monte Carlo or dynamic programming) are that it requires no model of system dynamics and updates $Q(s,a)$ whenever the new reward is observed without waiting for a final outcome. TD is responsive to environment variations and provides a good estimation for long-term performance. The values of $Q(s,a)$ are stored in a look-up table and updated by writing new values to the corresponding entries. The optimal configuration policy is generated by selecting the state with the maximal value after the table becomes stable. Algorithm 1 shows our approach of online learning value function (see Section 4.3 for the detials of guided exploration policy). We set α and γ as 0.1 and 0.9, respectively for value function updating.

Algorithm 1 Online value function learning.

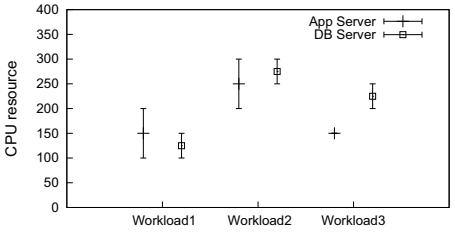
1: **Initialize** value function table.
2: **repeat**
3: $s_t = get\_current\_state()$;
4: $a_t = get\_action(s_t)$ using guided exploration policy;
5: $reconfigure(a_t)$;
6: $r = observe\_reward()$;
7: $s_{t+1} = get\_current\_state()$;
8: $a_{t+1} = get\_action(s_{t+1})$ using guided exploration policy;
9: $Q(s_t,a_t) = Q(s_t,a_t) + \alpha * (r + \gamma * Q(s_{t+1},a_t) - Q(s_t,a_t))$;
10: $s_t = s_{t+1}, a_t = a_{t+1}$;
11: **until** value function converges

4.2 Simplex-based Space Reduction

Our previous studies [21], [5] showed that RL method would suffer from curse of dimensionality in auto-configuration of VMs and applications. In each case, its state space grows exponentially with the number of configurable variables. The poor scalability hinders RL from being applied for the study of the interplay of VM and application configurations. In our hybrid RL approach, the Simplex method is used to reduce the state space to a much smaller "promising" configuration set. RL agent only conduct searching on this small set ignoring other "unpromising" states.



(a) Cumulative distribution of reward(penalty)



(b) Variations of CPU resource under different workloads

Fig. 4. Searching space reduction.

The state space reduction is motivated by an observation that for a give system situation, only a small portion of configurations would yield good performance. Figure 4(a) shows the fraction of the promising configurations in a search space of tens of thousands configurations. The three types of workloads represent various resource demand of a multi-tier application, defined in Table 3. As defined in Section 4.1, a positive reward means no SLA violation happened in any application. A negative reward implies at least one application can not meet its SLA. The result shows that only less than 10% configurations in each case would yield positive reward. For these desired configurations, the value of each parameter is always within a small range compared with the whole configurable scope. Figure 4(b) shows the variations in allocated CPU resource to TPCW application within the positive-reward configuration set. We can see that, for both application and database server, the variations are less than 100 percentage of CPU under all kinds of workloads.

To find the high-reward configuration set, we first divide the original state space S into multiple exclusive subsets by discretizing the configurable scope of each parameter i into m_i ranges. The states whose parameter settings fall into the same range are classified into the same subset. For each range, its median value is used to represent the whole configurable interval. The original space S is then converted to a range-based search space S', in which each parameter i has m_i configurable steps and each state represents a subset of configuration states in S. The best state s'_b in S' could be considered as the high-reward configuration set in the original search space S.

To determine the s'_b, we enhance the RL learning agents with the downhill Simplex method. It is a high efficient heuristic based method for nonlinear, unconstrained optimization problems. A simplex is a set of $n+1$ points in $\Re^n$. Its objective is to optimize unknown functions $f(x)$ for $x \in \Re^n$ through a sequence of transformations of the working simplex. For each App-Agent or VM-Agent, if there are n configurable parameters in all, the used simplex would contain $n+1$ vertices. The goal is to maximize the system performance in terms of the reward. At each iteration, agents receive performance feedback and select next configuration action based on the heuristic rules.

The Simplex algorithm starts by selecting a random working simplex $\tilde{S}$, whose vertices represent configuration states. Each iteration involves five steps: (1) Ordering—ranking the vertices in $\tilde{S}$ according to a pre-defined utility function; (2) Reflection—replacing worst vertex with a new vertex reflected through the centroid of the remaining n vertices. If the reflected vertex is better than the old vertex, accept the new vertex; (3) Expansion—If reflected vertex is the best one in current $\tilde{S}$, expand the vertex more along the reflected direction. If the expanded vertex better than the reflected one, accept the former; (4) Contraction—If the reflected vertex is worse than the second worst one, contract the worst vertex towards the better one; (5) Reduction—If none of the above steps can achieve a better vertex, shrink the $\tilde{S}$ around the centroid. During online search, $\tilde{S}$ keeps transforming and shrinking, and finally converges to the optimal state. Algorithm 2 shows our Simplex-based space reduction method in the reduction of the configurable state space. Parameters β, ω, ζ and λ are respectively the scale factor of the reflection, expansion, contraction and reduction. They were set to 1, 2, 0.5 and 0.5. The stop threshold τ was set to 0.05 to limit the searching time. More details about downhill Simplex method could be found in [3].

The downhill Simplex method is more time efficient than learning-based RL methods. For the small-scale range-based space S', it is able to quickly locate the best state s'_b. Recall that s'_b represents a set of configurations in orignal state space S. RL searching is only conducted within this best configuration set. The size of the actual search space is reduced to $1/\prod_{i=1}^{n} m_i$ of the orignal state space size. In this work, for each parameter, we set $m = 3$, representing $high$, $median$ and low settings. Because the space reduction process is conducted on each individual parameter, the reduction rate also increases exponentially with the problem scale. The RL algorithm can be always guaranteed a relatively small search space even when the original space grows tremendously.

4.3 System Knowledge Guided Exploration

It is known that there are two types of interactions between RL agents and the managed system: exploitation and exploration. Exploitation is to follow the current optimal policy; in contrast exploration is the selection of random actions to capture the

Algorithm 2 Simplex-based space reduction.

Form a ranged-based search space S';
Select a working smplex $\tilde{S} = \{s_i : s_i \in S', 1 \leq i \leq n+1\}$;
/* State s_i is a vector of configurable paramters.
Following state operations are all vector operations */
Define utility function $f(s_i) = 1/r_i$; /* r_i is the reward of s_i */
Given stop threshold τ;
repeat
  Ordering sample set $\tilde{S}$ according to utility function: $f(s_1) \leq f(s_2) \leq f(s_{n+1})$; /* s_{n+1} is the worst state */
  calculate the centroid $s_0 = \sum_1^n s_i/n$;
  Reflection: $s_r = s_0 + \beta * (s_0 - s_{n+1})$;
  if $f(s_1) \leq f(s_r) \leq f(s_n)$
  then replace s_{n+1} with s_r and **goto** step 8;
  Expansion: if $f(s_r) \leq f(s_1)$
  then $s_e = s_0 + \omega * (s_0 - s_{n+1})$;
    if $f(s_e) \leq f(s_r)$
    then replace s_{n+1} with s_e and **goto** step 8;
    else replace s_{n+1} with s_r and **goto** step 8;
  Contraction: $s_c = s_{n+1} + \zeta * (s_0 - s_{n+1})$;
  if $f(s_c) \leq f(s_{n+1})$
  then replace s_{n+1} with s_c and **goto** step 8;
  Reduction: replace all the points except s_1 with $s_i = s_1 + \lambda * (s_i - s_1)$ for $i, 2 \leq i \leq n+1$;
  goto step 8;
until $stdev(\tilde{S}) \leq \tau$

system dynamics so as to refine the existing policy. RL agent requires a certain amount of exploration operations to accumulate sufficient system knowledge for generating a stable policy. Basic RL algorithm adopts $\epsilon - greedy$ policy for action selection, under which the agent mostly conducts exploitation expect for random selections with a probability ϵ. How to balance exploitation and exploration is a fundamental problem for RL algorithms. A too small ϵ would slow down the learning process due to limited observations. In contrast, a too large ϵ would cause performance fluctuations because of frequently visiting suboptimal states.

Notice that performance fluctuations are mostly caused by selecting unwise actions during exploration. For example, removing resource from a busy VM would lead to more severe contention. Increasing the `MaxThreads` value in a busy web server would cause overload. To help RL agent make a wise decision, we employ a system knowledge-guided exploration policy. We consider two most performance relevant system metrics, CPU and memory utilization. Associated with each metric, the upper bound and lower bound are defined to regulate resource utlization. We denote U_{cpu}^{ub}, U_{cpu}^{lb}, U_{mem}^{ub}, and U_{mem}^{lb} as the upper bound and lower bound for CPU and memory, respectively. Resource utilizations should be kept within the ranges.

During the configuration process, the RL agents would keep a set of valid actions, $\mathcal{A}$, for each state. It is divided into three exclusive subsets: $\mathcal{A}_{inc}$, $\mathcal{A}_{dec}$, and $\mathcal{A}_{nop}$, which represent the set of actions to increase resources, decrease resources, and keep unchanged, respectively. If current resource utilization U goes beyond the corresponding U^{ub}, actions in $\mathcal{A}_{dec}$ would be removed from $\mathcal{A}$ to avoid further resource reduction. In contrast, if U becomes below the U^{lb}, actions in $\mathcal{A}_{inc}$ would be removed to prevent resource waste, see Algorithm 3 for the details. The new policy is expected to guide the exploration to select wise actions. Under this protection, a higher exploration rate could be used to collect more system information quickly. In this work, we set the exploration rate to 0.3 to accelerate the learning process. The resource upper bound and lower bound were set in Table 1, based on our experiences. We implemented system knowledge-guided exploration policy in VM-Agent and TPCW App-Agent.

Algorithm 3 System knowledge-guided exploration policy.

Given exploration rate ϵ;
Given U_{cpu}^{ub}, U_{cpu}^{lb}, U_{mem}^{ub}, U_{mem}^{lb};
Generate random number ran in [0,1];
if $ran \geq \epsilon$ **then select** the best observed action $a \in \mathcal{A}$;
else
for all monitored system resources **do**
  Check U;
  if $U \geq U^{ub}$ **then** $\mathcal{A} = \mathcal{A} - \mathcal{A}_{dec}$;
  else
  if $U \leq U^{lb}$ **then** $\mathcal{A} = \mathcal{A} - \mathcal{A}_{inc}$;
end for
Random Select action $a \in \mathcal{A}$;

4.4 Coordinated Auto-configuration

Recall that the processes of VM and application configurations interfere with each other. In some cases, system performance heavily depends on the order of the configuration procedures. For example, if we tune VM configuration first, an application parameter setting error would mislead the VM configuration decision and cause performance degradation and vice versa. Instead of specifying the configuration order for each situation, we employ a strategy that repeats the two levels of configuration alternatively until the system performance is stabilized. The coordinated configuration algorithm within individual VM cluster is shown in Algorithm 4. Different VM clusters should conduct configuration task in parallel. The algorithm starts the VM configuration at the beginning of the first loop because it helps involving all VMs as soon as possible. To quickly erase the side effect of unwise configurations, VM-Agent and App-Agent take Simplex searching alternatively at first. This makes agents putting more focus on promising states and reduces the algorithm's convergence time.

5 Evaluation Design

To evaluate the efficacy of CoTuner, we designed experiments to test the following capabilities of the

Algorithm 4 Coordinated auto-configuration algorithm.

1: **Form** search space S' for VM-Agent;
2: **Form** search space S' for each App-Agent;
3: **repeat**
4: **All the VM-Agent Perform** Simplex searching on the corresponding S';
5: **All the App-Agents Perform** Simplex searching in parallel on the corresponding S';
6: **Select** high-reward configuration set for VM-Agent;
7: **Select** high-reward configuration sets for App-Agents;
8: **VM-Agents Perform** RL based Auto-configuration using guided exploration policy;
9: **App-Agents Perform** RL based Auto-configuration in parallel using guided exploration policy;
10: **until** performance gets stable

TABLE 1
Resource thresholds setting.

Apps	U_{cpu}^{ub}	U_{cpu}^{lb}	U_{mem}^{ub}	U_{mem}^{lb}
TPCW	60%	40%	80%	50%
TPCC	80%	50%	80%	50%

approach: (1) Automatically adjust VM and application configurations according to system dynamics (Section 6.1); (2) Optimize system wide performance as well as reducing SLA violations for each application (Section 6.1); (3) Deal with the interplay between the two level of configurations and improve system performance over independent tuning (Section 6.2); (4) Improve configuration efficiency and regulate resource utilizations (Section 6.3); (5) Scale to a large virtual cluster (Section 6.4).

We developed a prototypes of the CoTuner framework and tested it on a DELL server cluster, connected by a gigabit Ethernet. Each server was configured with 2 quad-core Xeon CPUs and 8GB memory, and virtualized through Xen Vesion 3.1. Both the driver domain (Dom0) and the VMs were running CentOS 5.0 with Linux kernel 2.6.18.

We selected two benchmark applications, TPCW and TPCC. TPCW [1] is an online book store application, consisting of a tier of application in the front and a tier of database in the back. TPCC [2] is an online transaction processing (OLTP) workload that represents a wholesale parts supplier operating out of a number of warehouses. Unlike TPCW which is largely CPU-intensive, TPCC contains a large amount of lightweight disk reads and sporadic heavy writes and its performance is more sensitive to memory size. The TPCW application was run Tomcat/MySQL servers. MySQL was used as the database server in TPCC applications.

To characterize the behavior of each application, we first conducted experiments in two physical servers, each hosting two VMs. We deployed TPCW across the two servers with application server on one server and database on the other. Each physical server also hosted one TPCC instance. Such configuration represents a typical scenario of VM deployment: a multi-tier application is deployed across different servers, and heterogeneous applications may be consolidated in the same physical server. We pinned the two VMs on the same physical server onto 4 physical cores and 4 GB memory to cap the VM resource in order to demonstrate the effect of configuration in a resource-contented system. We also evaluated the scalability of our approach on a virtualized environment with 16 physical servers and 100 VMs.

TABLE 2
Tunable performance critical parameters.

	Parameters	Default Setting
Virtual Machine	cpu time	equally allocated
	memory	equally allocated
Tomcat	MaxThreads	150
	Session timeout	30
MySQL	keyBufferSize	32MB
	maxHeapSize	32MB

We used the metrics of throughput and response time to evaluate the applications performance. We define system-wide throughput to be accumulated throughput of all running applications within the system. We assumed response time SLA to be 5 seconds for both TPCW and TPCC applications. To evaluate and compare application throughput under different workloads, we define the throughput when 95% of incoming requests are finished in time (5 seconds in our experiments) as a reference value (*i.e.* throughput SLA) for normalization.

We selected virtual CPU time and virtual memory as VM configuration parameters. The configuration actions were issued through Dom0's privileged control interface `xm`. For the TPCW and TPCC applications, four most performance-critical parameters are `MaxThreads`, `Session Timeout`, `keyBufferSize`, and `maxHeapSize`. Their default settings are shown in Table 2.

6 Experimental Results

6.1 System Wide Performance Optimization

In this section, we evaluate CoTuner's capabilities in configuration adaptation, performance optimization and SLA guarantee during workload changing. The workload intensity of TPCW application is determined by the number of concurrent clients and that of TPCC application is controlled by the number of warehouses. We defined three types of workload mixes in Table 3, representing various resource demands of the applications. We ran the experiment for 240 configuration iterations; each iteration lasted 1 minute. That is, the cloud system performed a reconfiguration operation every 1 minute. To demonstrate the adaptivity of the reconfiguration process, we divided the 240 iterations into 3 phases and changed the

TABLE 3
Workloads settings.

Workloads	TPCW	TPCC-1	TPCC-2
workload-1	1000 clients	30 WHs	10 WHs
workload-2	3000 clients	5 WHs	5 WHs
workload-3	2000 clients	10 WHs	20 WHs

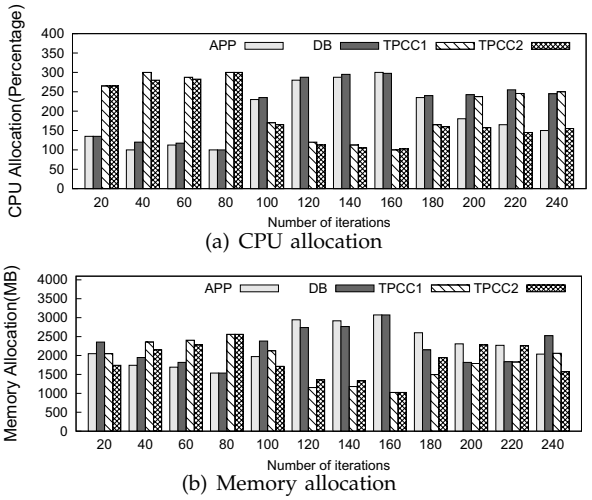
TABLE 4
Application parameter configuration.

Parameter	wkload1	wkload2	wkload3
MaxThreads	100	350	250
Session timeout	5 sec	1 sec	3 sec
KeyBufferSize(TPCC1)	64MB	16MB	32MB
maxHeapSize(TPCC1)	64MB	16MB	64MB
KeyBufferSize(TPCC2)	64MB	32MB	64MB
maxHeapSize(TPCC2)	128MB	16MB	64MB

workload mix from workload-1, to workload-2, and to workload-3 every 80 iterations. We assumed virtual resources were initially evenly distributed among the VMs and that the virtual appliances were run in their default parameter settings in the beginning.

Figure 5 shows the CPU and memory allocations to the applications on both servers. Each data point is an average of a moving window of 20 iterations. From the figure, we can see that in the first phase, CoTuner gave more portions of CPU and memory resources to TPCC applications to meet their SLAs. When workload mix changed from workload-1 to workload-2 at iteration 80, TPCW workload became heavy and TPCC workload applications was reduced. In response to the workload change, CoTuner reclaimed idle resources from TPCC applications and allocated them to the TPCW. In the last phase, all the applications were under medium workloads. They all obtained sufficient resources proportionally to their demands. Table 4 shows the parameter settings of the three applications due to CoTuner under three workloads. These key parameters were reconfigured according to incoming workloads and allocated resources. The result demonstrates that CoTuner was able to detect the system variations and adapt both VM and application configurations to dynamic traffics.

Figure 6 shows the accumulated throughput of all three applications due to our hybrid RL algorithm in the CoTuner framework. For comparison, we implemented three other algorithms for coordinated configuration under the framework: Simplex algorithm, basic RL (BRL) and Model-based RL (MRL). They followed the same configuration strategy only different in decision algorithms. Simplex and BRL framework used standard Nelder-Mead and Reinforcement algorithm respectively as their decision making method. To avoid performance fluctuations, we set exploration rate as low as 0.05 for BRL approach and a relatively relaxed stop threshold 0.1 for Simplex algorithm.



(a) CPU allocation
(b) Memory allocation

Fig. 5. Resource allocations to different VMs.

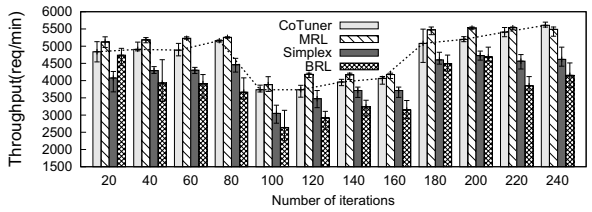


Fig. 6. System wide performance due to different configuration strategies.

Simplex method was used in previous study [33] for automatic parameter tuning. MRL adopt model-based reinforcement learning method used in our previous work [21]. We built the performance prediction model for each workload elaborately. Because MRL assumed the perfect future knowledge of incoming workload and the accurate performance model, its results set an upper bound of the performance.

From the figure, we can see that CoTuner with hybrid RL algorithm was able to consistently optimize system wide throughput and drive the system into a high-productive configuration state rapidly whenever there is an abrupt workload change. Although performance fluctuations (represented as error bar) happened at the initial adaptation stage, the variations in throughput were significantly reduced after hybrid RL agent had observed sufficient configuration states (usually after 40 iterations for each workload). It is expected that Simplex and BRL strategies led to much more performance fluctuations and degradation. CoTuner gained more than 30% performance improvement over the Simplex strategy and as much as 40% improvement over the BRL strategy. Due to the use of prediction models, MRL strategy could drive system to a high-productive configuration faster than CoTuner. But the hybrid RL was able to achieve more than 95% of the optimal performance due to MRL after accumulating a certain amount of system information.

From the figure, we can also know that the Simplex and BRL frameworks had the capability of adapting

configurations to system dynamics and improving system performance. But their limitations restricted their applications in online auto-configuration. Simplex method is based on a series of heuristic trail-and-error tests. It tends to be trapped in local optimal state. In contrast, CoTuner performs a RL-based search following Simplex search. The objective of maximizing accumulated long term reward in RL inherently solve this problem. BRL strategy requires a long exploration time. It can not generate an optimal policy for such a large state space in tens of iterations due to too few observations. The hybrid RL approach in CoTuner framework employs Simplex method to reduce the huge search space and execute RL searching only within the high-reward configuration set. It takes the advantages of both approaches and appropriately addresses their limitations.

We note that Figure 6 shows the accumulated throughput of all three applications. To reflect the service quality of individual applications with respect to their respective SLA requirements, we present their normalized throughput due to different configuration in Figure 7. The figures show that CoTuner framework was able to improve the performance of each individual application and guarantee their SLAs after the system became stable. In contrast, Simplex and BRL framework may treat applications unfairly due to unbalanced resource configurationa. For example, controlled by Simplex framework, TPCW application yield a high throughput under workload2. The normalized values reached 1.2 after it became stable. However, TPCC1 and TPCC2 applications were unfairly treated under workload2 with a delivery of about 80% SLA throughputs. In cloud computing environment, different applications maybe belong to different users. Their performance should be equally guaranteed. It is not reasonable to sacrifice any application even if for system wide performance improvement. In CoTuner, SLA is guaranteed by adding a huge penalty to the reward if any SLA violation happened.

Figure 8 shows cumulative distribution of SLA violations with respect to throughput and response time over the 240 configuration iterations. From Figure 8(a), we can observe that CoTuner was able to keep system wide throughput SLA violation rate less than 15% during the configuration process. In contrast, the SLA violation rates due to Simplex and BRL strategies could reach as high as 85% due to their low efficiency of finding good configurations. Figure 8(b) shows the CoTuner's SLA violations distribution, in terms of response time, for each application. From the figure, less than 15% violation rates could be observed.

6.2 Benefits of Coordinated Configuration

As discussed in Section 2, VM and application configurations are related. Independent tuning either of them can hardly lead to optimal system-wide performance. In this section, we demonstrate how well CoTuner deal with the interplay and how much performance improvement could be resulted from coordinated configuration.

We considered one TPCW application with a workload of 3000 clients in this experiment. In the first phase of 80 iterations, App-agent was disabled and the application parameter, `MaxThreads`, was set to a small value of 50. We enabled the VM-agent and

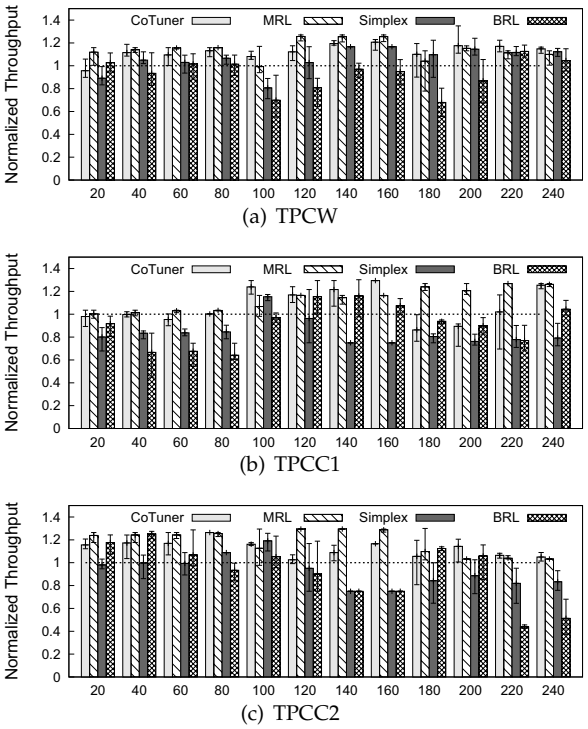


Fig. 7. Application performance due to different configuration frameworks.

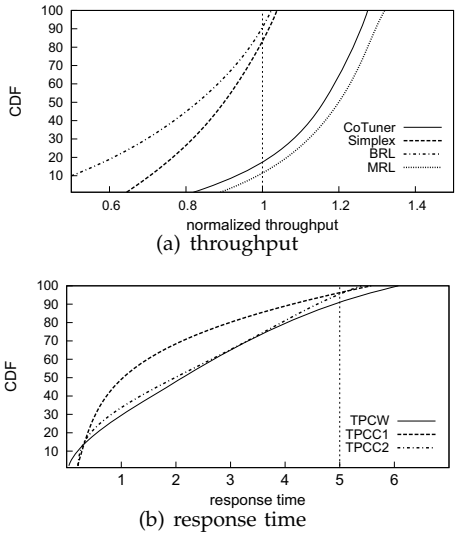


Fig. 8. Cumulative distribution of SLA violations.

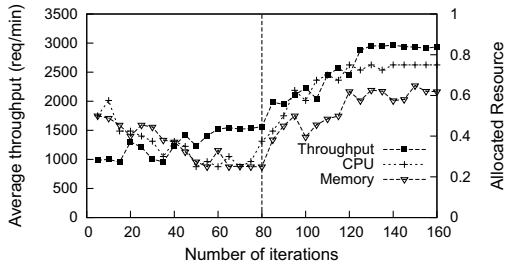


Fig. 9. Interaction between VM and Application configurations.

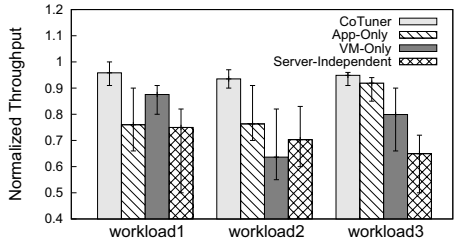


Fig. 10. System performance due to different configuration strategies.

App-agents at iteration 81, which configured the VM and application parameters in a coordinated way. Figure 9 shows that the throughput saturated at 1500 requests/minute in the first phase due to the low setting of `MaxThreads`. The App-agent enabled at iteration 81 increased `MaxThreads` to 350 automatically and brought the system performance up to 2800 requests/minute in the second phase. Figure 9 also shows the CPU and memory resource allocation over the time, normalized to their caps: 400 CPU percentage and 4GB memory. Since extra resource could not improve the throughput, VM-Agent just allocated 20% of them to TPCW and reserved the remaining for other applications. After the application misconfiguration was corrected, CoTuner increased the resource allocation to more than 60% for both CPU and memory within 40 iterations. The results suggest that CoTuner was able to deal with the interactions between VM and application configurations efficiently.

We compared CoTuner with three independent tuning strategies: VM-Only, App-Only and Server-independent. VM-Only and App-Only agents consider single aspect of the two level configurations. For VM-Only agent, we assumed the default settings for application parameters. For App-Only agent, virtual resources were evenly distributed to the VMs. In Server-independent strategy, each physical server pursues its performance optimization individually. For fairness in comparison, all agents are run with a hybrid RL algorithm and a system knowledge-guided exploration policy. Recall that the system performance resulted from MRL was optimal for application specific workloads. We normalize the throughputs due to different tuning strategies to the maximum throughput in Figure 10. From the figure, we can see that CoTuner could deliver as much as 95% of the upper bound for all the workloads. It was able to adapt both VM and application configurations to the traffic dynamics. In contrast, Server-independent agent could never reach 80% of the maximum throughput. Performance of VM-Only and App-Only agents fluctuated to a large extent due to their limited adaptability. Coordinated tuning achieved an improvement of system wide performance by 20%-33% over independent tuning. We note that independent tuning assumed default application parameter settings or even VM resource distribution. Any misconfiguration in either aspect would lead to significant performance degradation, as shown in Figure 9. Coordinated tuning is able to correct such misconfiguration and improve performance.

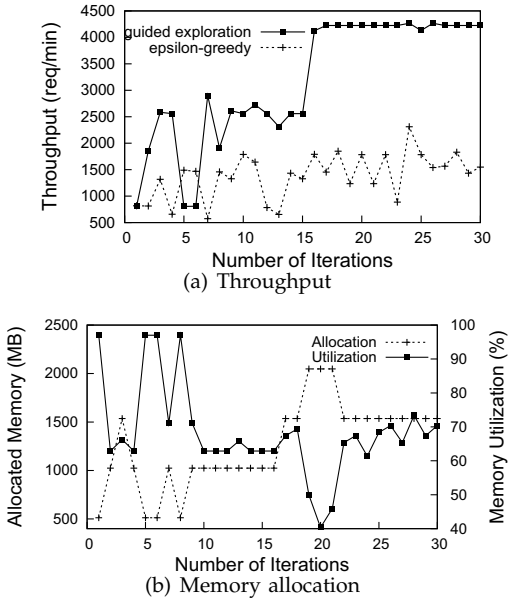
6.3 Effect of Guided Exploration

In this experiment, we examined the effect of the system knowledge-guided exploration policy, in comparison with the standard $\epsilon - greedy$ policy. We considered a single TPCW application with VM-Agent in operation. Application parameters were carefully tuned to adapt to the workload.

Figure 11(a) shows the guided exploration policy gained throughput improvement as much as 200% over the $\epsilon - greedy$ policy. It was able to lead the system to a good configuration state within 15 iterations. The result also suggests that system performance obtained consistent improvement during configuration process. The large variation in the beginning was due to lack of memory resource. However the guided policy could quickly direct the VM-Agent to correct this misconfiguration and bring the throughput back to a normal level.

Figure 11(b) also shows the change of memory allocation and utilization of the Tomcat application server. We can see that at iteration 1, the memory utilization went beyond U^{ub}_{mem} (80%) and approached nearly 100%. Accordingly, the guided policy selected an action of "memory increase" in the next two iterations, leading to significant performance improvement at iteration 2 and 3. Similarly, at iteration 20, memory utilization dropped down to 40% below U^{lb}_{mem} (50%). An action of "memory decrease" helped reclaim idle resource. After that, memory utilization was brought back to 70% without compromising system throughput.

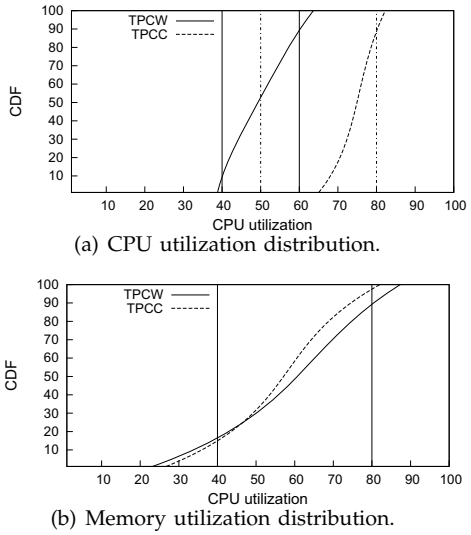
Figure 12 shows the cummulative distributions of resource utilization for multiple applications we experimented with in Section 6.1. We considered the two TPCW VMs as a group (TPCW) and the other two TPCC VMs as another group (TPCC). The vertical lines represent the bounds for CPU or memory utilizations defined in Table 1. We can see that, for both CPU and memory resources, the outlier points are below 20%. This suggests that guided exploration



(a) Throughput

(b) Memory allocation

Fig. 11. Effect of guided exploration in comparison with $\epsilon - greedy$.



(a) CPU utilization distribution.

(b) Memory utilization distribution.

Fig. 12. Bounded resource utilization.

policy could effectively bound the resource utilization and lead to a higher learning efficiency.

6.4 Scalability Analysis

In this section, we evaluate the scalability of CoTuner using testbeds with 16 physical servers and 100 VMs. 40 instances of the multi-tier TPCW application and 20 instances of the TPCC application were deployed, which formed several VM clusters in different sizes. All VM clusters conducted configuration tasks in parallel. We defined the largest VM cluster as the dominant cluster. We ran the experiments five times with different number of VMs from 20 to 100. Each experiment lasted for 720 iterations. The applications changed their workloads very 240 iterations.

Figure 13 shows the settling time of configuration agents in testbeds of different sizes. We defined the settling time as the duration (number of iterations) required for the performance to reach and stay within a specified error band (5% in this work) around the final value. From the figure, we can see that the settling time increased slightly with the scale of the testbed. Because the configuration processes run in parallel, the system wide performance is usually determined by the dominant cluster. The performance deviations were due to existing multiple dominant clusters. For example, there were 2 dominant clusters with size 16 in the third testbed and 5 dominant clusters with size 16 in the fifth testbed. The result shows that CoTuner was able to keep average settling time under 85 iterations for all the testbeds. It retained the high efficiency when the controlled system scaled up.

Figure 13 also shows the average SLA violation rates with respect to throughput and response time among all the running applications. The SLA violation rate was defined as the number of configuration steps leading to SLA violations over the total number of running iterations. It involved transient configuration states over the online learning time and steady states reached after the policy becomes stable. CoTuner was able to keep the overall SLA violation rate under 12%. When the system in steady state, the average SLA violation rate was no more than 4.1%.

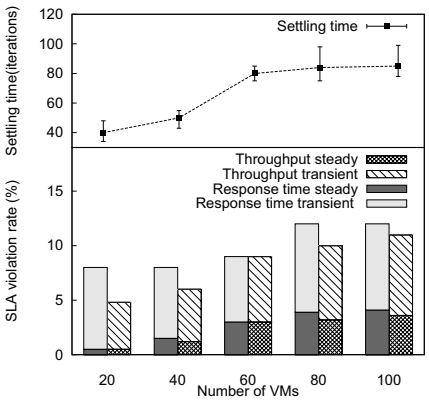


Fig. 13. Scalability analysis.

7 Related Work

In [13], Kephart and Chess defined autonomic computing as a general methodology in which computing systems can manage themselves given high level objectives from administrators. A recent comprehensive survey of autonomic computing works can been seen in [11]. Cloud dynamics requires an autonomic management strategy. Many researches have devoted to adaptive configuring virtual machines and resident application parameters for optimizing system performance.

In [30], [6], [33], classical combinatorial optimization approaches were applied to automate the tuning

process. In their works, automated configuration is to find a combination of parameters setting that maximizes the performance function. Such approaches were applied to configuration of software systems like Apache server [29], application server [30], database server [14] and online transaction services [6], [33]. Xi, et al. [30] and Zhang, et al. [33] used hill-climbing and Simplex algorithms to search optimal server configurations for performance optimization. These heuristic approaches tend to trap the system into local optimums when the complexity of configuration problem is increased due to cloud dynamics. By maximizing the long term reward, RL could effectively avoid local optimum problem. Moreover, these works assumed the hardware resources allocated to the applications were unchanged. In contrast, our work consider application parameter tuning in the presence of virtual resource variation.

There were other recent approaches based on feedback control for online configuration. In [19], Padala, et al. attempted to apply adaptive control to automate the configuration of virtual machine resources. Xu, et al. proposed an adaptive controller to modulate CPU share between different classes of traffic for page-view response time guarantees [28], [31]. In [8], Gong, et al. developed a model-predictive feedback controller for responsive power capping. They further proposed an adaptive control method to coordinate power and performance in virtual machines [9]. For controllability, most of them restricted themselves to the tuning of limited number of parameters of target sytems. In [7], a distributed controller was proposed in a two-tier website with an actuator in each tier. Padala et al. [17] and Wang et al. [27] successfully applied multi-input multi-out controllers to adaptively configure resource in virtual data centers. Their approaches rely on predefined explicit system models, building which can be highly knowledge-intensive and labor-intensive for complex system. Different from them, our hybrid RL approach does not require explicit model of either the managed system or the external process.

RL-based approaches were initially applied to automatic configuration of application parameters in web systems [5] and VM resource in virtual data centers [21] independently. Both works designed the configuration agents to be run with model-based performance approximators or pre-learned initial Q-table, for the purpose of addressing the scalability and poor initial performance issues. In [20], we proposed a scalable distributed RL algorithm for elastic virtual resource provisioning in a large scale data center. The configuration agents were enhanced with a highly efficient representation of experiences (CMAC) and intelligent initial policies to improve system performance. However, the interplay between the two layers of configurations significantly increases the complexity of coordinated configuration task. Building an accurate performance approximator or initial policies requires much human effort and knowledge in such context. In contrast, our hybrid RL algorithm employs Simplex method to reduce state space and uses system knowledge-guided exploration policy to guide configuration process. It does not assume any performance model or initial policy. The Simplex method and RL method have found their applications in other aspects of computer systems. Zheng, et al. [33] applied Simplex method to reduce the number of trails and error testings in automatic configuration of web services. Recent studies showed the feasibility of RL approaches in resource allocation [21], [23], [25], power management [24], job scheduling in grid [4] and memory controller [12].

Previous studies on virtual resource management or application parameter tuning considered these two layers of configurations independently. To our best knowledge, CoTuner framework should be the first approach towards coordinated auto-configurations of both VM resource and running appliances in clouds, dealing with interference between them.

8 Conclusions

In this paper, we have presented a model-free coordinated auto-configuration framework CoTuner to automatically tune both virtual resource allocations and application parameters for system performance optimization. At the heart of the framework is an efficient approach based on Simplex optimization and reinforcement learning methods. It is enhanced by system knowledge-guided exploration policy to accelerate the learning process. Experimental results on Xen VMs with TPCW and TPCC benchmarks showed the CoTuner was able to adapt VM and appliance configurations to cloud dynamics and drive system to an optimal or near optimal state eventually.

Current implementation of CoTuner framework was limited to web appliances, focusing on the resources of CPU and memory. Other important resources such as network I/O, disk I/O, especially L2 cache for multi-core CPUs should be considered in future work. Moreover, the newly designed guided exploration policy just worked well for VM-Agent and TPC-W App-Agent. More application specific domain knowledge was needed for other App-Agents. A general guided exploration policy deserves further study. The CoTuner framework required no prebuilt models through machining learning or system identification. Due to the lack of system dynamic knowledge, it may take a little longer time to find an optimal configuration compared with other model-based approaches. This hinders the CoTuner from being applied for short-lived appliances or virtual machines. Moreover, CoTuner could not generate a global policy for all states. It just focus on more promising configuration set.

REFERENCES

[1] http://www.tpc.org/tpcw.

[2] http://www.tpc.org/tpcc.

[3] M. Avriel. *Nonlinear Programming: Analysis and Methods*. Dover Publishing, 2003.

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

[5] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online web systems auto-configuration. In *ICDCS*, 2009.

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

[7] Y. Diao, J. L. H. andTesauro Sujay S. Parekh, H. Shaikh, and M. Surendra. Controlling quality of service in multi-tier web applications. In *ICDCS*, page 25, 2006.

[8] J. Gong and C.-Z. Xu. A gray-box feedback control approach for system-level peak power management. In *ICPP*, 2010.

[9] J. Gong and C.-Z. Xu. vpnp: Automated coordination of power and performance in virtualized datacenters. In *IWQoS*, 2010.

[10] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware*, 2006.

[11] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

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

[13] J.O.Kephart and D.M.Chess. The vision of autonomic computing. In *IEEE Computer*, 2003.

[14] E. Kwan, S. Lightstone, K. B. Schiefer, A. J. Storm, and L. Wu. Automatic database configuration for db2 universal database: Compressing years of performance expertise into seconds of execution. In *BTW*, pages 620–629, 2003.

[15] 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.

[16] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling i/o in virtual machine monitors. In *VEE*, 2008.

[17] 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 *EuroSys*, pages 13–26, 2009.

[18] 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.

[19] 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.

[20] J. Rao, X. Bu, and C.-Z. Xu. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *MASCOTS*, 2011.

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

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

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

[24] 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.

[25] 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.

[26] Y. Wang, R. Deaver, and X. Wang. Virtual batching: Request batching for energy conservation in virtualized servers. In *IWQoS*, 2010.

[27] Y. Wang, X. Wang, M. Chen, and X. Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *IEEE Real-Time Systems Symposium*, pages 303–312, 2008.

[28] J. Wei and C.-Z. Xu. eqos: Provisioning of client-perceived end-to-end qos guarantees in web servers. *IEEE Trans. Computers*, 55(12):1543–1556, 2006.

[29] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *OSDI*, 2004.

[30] 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.

[31] C.-Z. Xu, J. Wei, and F. Liu. Model predictive feedback control for end-to-end qos guarantees in web servers. In *IEEE Computer*, 2008.

[32] Y. Zhang, W. Qu, and A. Liu. Automatic performance tuning for j2ee application server systems. In *WISE*, 2005.

[33] W. Zheng, R. Bianchini, and T. D. Nguyen. Automatic configuration of internet services. In *EuroSys*, 2007.