iBalloon: Self-Adaptive Virtual Machines Resource Provisioning

Jia Rao, Xiangping Bu, Kun Wang, Cheng-Zhong Xu,
Dept. of Electrical and Computer Engineering
Wayne State University
Detroit, Michigan, USA
{jrao, xpbu, kwang, czxu}@wayne.edu

ABSTRACT

Although cloud computing has gained sufficient popularity in the last two years, there are still some key impediments to enterprise adoption. Cloud management is one of the top challenges. The ability of on-the-fly partitioning hardware resources into virtual machine(VM) instances facilitates the cloud to provide elastic computing environment to users. But the flexibility of resource provisioning in turn poses challenges on effective cloud management. Time-varying user demand, complicated interplay between co-hosted VMs and the arbitrary deployment of multi-tier applications make it difficult for administrators to figure out optimal or even good VM configurations. In this paper, we propose iBalloon, a generic framework that facilitates self-adaptive virtual machines resource provisioning. iBalloon treats cloud resource allocation as a distributed learning task, in which each VM being a highly autonomous agent is allowed to submit resource request at its own benefit. iBalloon evaluates the requests and gives feedbacks to individual VMs. We select reinforcement learning with a highly efficient representation of experiences as the heart of the VM side learning engine. Experiment results on a Xen-based cloud testbed demonstrate the effectiveness of iBalloon. The distributed VM agents are able to agree on near-optimal configurations with an acceptable amount of time. Most importantly, iBalloon shows good scalability on resource allocation by scaling to more than a hundred correlated VMs.

1. INTRODUCTION

One important offering of cloud computing is to deliver computing Infrastructure-as-a-Service (IaaS). In this type of cloud, raw hardware infrastructure, such as CPU, memory and storage, is provided to users as an on-demand virtual server. Aside from client-side reduced total cost of ownership due to a usage-based payment scheme, a key benefit of IaaS for cloud providers is the increased resource utilization in data centers. Due to the high flexibility in adjusting virtual machine (VM) capacity, cloud providers can consolidate traditional web applications into a fewer number of physical servers given the fact that the peak loads of individual applications have few overlaps with each other [3].

In the case of IaaS, the performance of hosted applications relies on effective management of VMs capacity. However, the agile and dynamic cloud infrastructure introduces unique requirements for the management. First, effective cloud management should be able to resize individual VMs in response to the change of application demands. More importantly, besides the objective of satisfying Service Level Agreement (SLA) of individual applications, system-wide (the hosting server) efficiency ought to be optimized. In addition, real-time requirements for VM resource provisioning make the problem even more challenging.

Although server virtualization helps realize performance isolation to some extent, in practice, VMs still have chances to interfere with each other. It is possible that one rogue application could adversely affect the others [17, 7]. In [6], the authors showed that for VM CPU scheduling alone, it is already too complicated to determine the optimal parameter settings. Taking memory, I/O and network bandwidth into provisioning will further complicate the problem. Frequently changed application needs add one more dimension to the configuration task. It is also observed that dynamics in application demands can possibly invalidate prior good VM configurations and result in significant performance loss [21].

Furthermore, practical issues exist in fine-grained VM resource provisioning. By setting the management interval to be 30 seconds, the authors in [21] observed that in some cases a VM needs more several minutes to get its performance stabilize after memory reconfiguration. Similar delayed effect can also be observed in CPU reconfiguration, partially due to the backlog of requests in prior intervals. The difficulty in evaluating the immediate output of management decisions makes the modeling of application performance even harder.

From the standpoint of cloud users, exporting infrastructure as a whole gives them more flexibility to select VM operating systems (OS) and the hosted applications. But this poses new challenges to underlaying VM management as well. Because public IaaS providers assume no knowledge of the hosted applications, VM clusters of different users may overlap on physical servers. The overall VM deployment can show an dependent topology on physical hosts. The bottleneck of multi-tier applications can shift between tiers either due to workload dynamics [22] or mis-configurations on one tier [4]. The mis-configured VMs can possibly become the rogue ones affecting others. Thus, in the worst case, all nodes in the cloud may be correlated with each other and any mistake in the capacity management of one VM may spread in the cloud.

Our previous work [21] demonstrated the efficacy of reinforcement learning (RL)-based resource control in dealing with cloud uncertainties for the objective of long-term optimization. The VM auto-configuration was initiated at the level of host machines based on carefully trained environment models. The models, which capture the relationship between VM settings and summarized host-wide performance, are critical to the efficacy of the VM management. The complexity of training and maintaining these models grows exponentially with the number of VMs types and resources. Although effective, the models are sensitive to workload dynamics. The cloud management needs to maintain different models for possible combinations of typical traffic patterns, which is prohibitively expensive in public clouds.

In this paper, we change the way that cloud management is considered, a centralized approach. We present *iBalloon*, a distributed learning framework that allows self-adaptive virtual machine resource provisioning. More specifically, our contributions are as follows:

(1) Generic distributed learning framework. Unlike [18, 21], in which VM resource provisioning was considered as a centralized optimization problem, we treat VM resource allocation as a distributed learning task. Instead of the resource providers, cloud users initiate VM capacity management in response to application demands. The host evaluates the aggregated resource requests and gives feedback to individual VMs. Based on the feedbacks, each VM learns its capacity management policy accordingly. This framework is generic that any distributed learning algorithm can be incorporated, without affecting the scalability of the management.

(2) Resource efficiency metric. We introduce a resource efficiency metric to measure the VMs' capacity management decisions. The metric syncretizes application performance and resource utilization. When employed as feedback signals, it effectively punishes decisions that violate applications' SLA and gives users incentives to release unused resources.

(3) Self-adaptive capacity management We develop a reinforcement learning-based decision making engine for the adaptive capacity management. The learning agent operates on VM's running status and reconfigures its capacity. We use a highly efficient representation of Q table to record past experiences. With limited history information, the agent is able to find near-optimal capacity configurations with small computation and storage cost.

(4) Design and implementation of iBalloon. Our prototype implementation of iBalloon demonstrated its effectiveness in a Xen [32] based cloud testbed. iBalloon was able to find near optimal configurations for a total number of 128 VMs on a 16-node closely correlated cluster with no more than 5% percent of overhead on application performance. We note that, there were reports in literature about the automatic configuration of multiple VMs in a cluster of machines. This is the first work that scales the auto-configuration of VMs to a cluster of correlated nodes under work conserving mode.

The rest of this paper is organized as follows. Section 2 discusses the challenges in cloud management. Section 3 and Section 4 elaborate the key designs and implementation of iBalloon respectively. Section 5 gives experiments settings and results. Related work is presented in Section 6. We conclude this paper and discuss future works in Section 7.

2. CHALLENGES IN CLOUD MANAGEMENT

In this section, we review the issues of CPU, memory and I/O resource allocations in cloud and discuss the practical issues behind on-the-fly VM resource reconfiguration and large scale VM management.

2.1 Multiple Reconfigurable Resources

In cloud computing, application performance depends on the application's ability to simultaneously access multiple types of resources [18]. In this work, we consider CPU, memory and I/O bandwidth as the building blocks of a VM's capacity. For each of the resources, we discuss the following questions:

1. How is the resource shared between multiple VMs ?
2. How is the utilization of the resource calculated ?
3. How to model the relationship between the resource and application performance ?

We believe these questions are essential to the design of any automatic cloud management system. The discussions are based on Xen virtualization platform and should be applicable to other virtualization platforms like VMware and VirtualBox. In the Xen based platform, the driver domain (`dom0`) is a privileged VM residing in the host OS. It manages other guest VMs (`domU`) and performs the resource allocations. [1]

2.1.1 CPU

The CPU(s) can be time-shared by multiple VMs in fine-grain. For example, the Credit Scheduler, which is the default CPU scheduler in Xen, can perform the CPU allocation in a granularity of 30 ms. On boot, each resident VM is assigned a certain number of virtual CPU (VCPU), and the number can be changed on-the-fly. Although the number of VCPUs does not determine the actual allocation of CPU cycles, it decides the maximum concurrency and CPU time the VM can achieve. In general, CPU scheduling works in a *work-conserving (WC)* or *non-work-conserving (NWC)* mode. In WC mode, a VM can use any amount of CPU time as long as there is free CPU cycles. Under NWC mode, in contrast a VM can only use CPU up to a limit (or *cap* in Xen) even there are idle times.

It is convenient to obtain the VMs' CPU utilization. The usage can be reported by `dom0` using `xentop` or by the VM's OS (e.g. the `top` command in linux). However, it is complicated to change the CPU allocation to VMs. In general, there are three ways to alter CPU allocation:

1. Under WC mode, set VMs' VCPU to the number of available physical CPU and change the CPU allocations by altering VMs priorities (or *weight* in Xen).
2. Under WC mode, change CPU allocation by altering the VCPU number. It is equal to setting an upper limit of CPU allocation to the VCPU number. Within the limit, a VM can use CPU for free.

[1]For the rest of this paper, we use `dom0` and the host interchangeably. VMs always refer to the guest VMs or `domUs`.

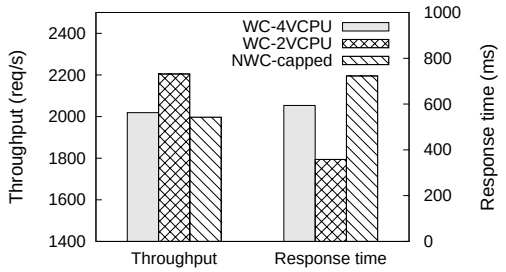


Figure 1: TPC-W under different CPU modes.

3. Under NWC mode, same as the first method, except that the allocations are specified as cap values. All the cap values add up to the total available CPU resource.

To determine the best CPU mode in cloud management, we compared the above three methods on a host machine with two quad-core Intel Xeon CPUs. Two instances of TPC-W database (DB) tier were consolidated on the host. For more details about the TPC-W application, please refer to Section 5. The DB tier is primary CPU-intensive and the VMs were limited to use the first four cores only. We make sure that the aggregated CPU demand is beyond the total capacity of four cores.

Figure 1 plots the aggregated throughput and average response time over of two TPC-W instances, under different CPU modes. WC-4VCPU refers to the first method with equal weight of the two VMs. Although the aggregated CPU demand is beyond four cores, each VM actually needs a little more than two cores. It is equivalent to work-conserving with "over-provisioning" of CPU to individual VMs. WC-2VCPU is similar except that there is a 2-VCPU upper limit for each VM. In NWC-capped, we set the VMs to have 4 VCPU and each of the VM was capped to half of the CPU time. For example, in the case of four cores, a cap of 400 means no limit while 200 refers to half of the capacity.

In the figure, we can see that WC-2VCPU provided the best performance in both throughput and response time. Plausible reasons for the compromised performance in the other two modes can be attributed to possible wasted CPU time. CPU contentions in WC-4VCPU may lower the CPU efficiency in serving client requests. In principle, NWC-capped should have similar performance as WC-2VCPU. It turned out that WC-2VCPU provided a soft limit on CPU allocation rather than the hard limit under NWC-capped. The ability of the VMs to temprally access more CPU resources boosted the application performance considerably.

Under NWC mode, there is usually a simple (and often linear) relationship between CPU resource and application performance. In [18], the authors showed an auto-regressive-moving-average model can represent this relationship well. However, in WC mode, the actual allocated CPU time to a VM is determined by the total CPU demand on the host, which makes the modeling harder. We take the challenges to consider WC mode in the VMs capacity management because it provides better performance and avoids possile waste of CPU resource.

2.1.2 Memory

Unlike CPU, memory is usually shared by dividing the physical address space into non-overlapping regions, each of which is used dedicatedly by one VM [2]. The objective of the cloud memory management is to dynamically balancing "unused" memory from idle VMs to the busy ones. Identification of "unused" memory pages or calculation of the memory utilization of a running VM is not trivial. Different from free pages, "unused" pages refer to those that once touched but not actively being accessed by the system. It can be calculated as the total memory minus the system working set.

System working set size (WSS) can be estimated either by monitoring the disk I/O and major page faults [11], or using miss ratio curve [33]. But these methods are only sensitive to memory pressure and are able to increase VM memory size accordingly. Any decrease of memory usage can not be quickly detected. That is, the memory of a VM may not be shrunk promptly.

In general, the relationship between VM memory size and application-level performance is simple. That is the performance drops dramatically when the memory size is smaller than the application's WSS. The open cloud environment adds one more uncertainty to VM memory management. Modern OSes usually design their write-back policies based on system wide memory statistics. For example, in Linux, by default the write-back is triggered when 10% of the total memory is dirty. A change of VM memory size may trigger background write-backs affecting application performance considerably although the new memory size is well above the WSS.

2.1.3 I/O Bandwidth

All the I/O requests from VMs are serviced by the host's I/O system. If the host's I/O scheduler is selected properly, e.g. the CFQ scheduler in linux, VMs can have differentiated I/O services. Setting a VM to a higher priority leads to higher I/O bandwidth or lower latency. The achieved I/O performance depends heavily on the sequentiality of the co-hosted I/O streams as well as their request sizes. Thus, the I/O usage, e.g. the achieved I/O bandwidth reported by command like `iostat`, does not directly connect to application performance.

There are two key impediments in mapping the memory or I/O resources to application performance. First, it is difficult to accurately measure the utilization of the resources. Second, the actual resource allocation (e.g. achieved I/O bandwidth) is determined by the characteristics of the applications as well as the co-running VMs.

2.2 Issues on Fine-grained VM Reconfiguration

VM capacity management relies on precise operations that reset resources to desired values. The immediate outcome of the management operations then can be used to adjust the policy. However, in fine-grained cloud management, such as in [18, 21], within the management interval the outcome of a reconfiguration can not be correctly perceived. The work in [21] showed up to 10 minutes delayed time before a memory reconfiguration stabilizes. Similar phenomenon was also observed in CPU.

[2]Although it is possible for a VM to give up unused memory through self-ballooning [15], during each management interval we consider the allocated memoy be used exclusively by one VM.

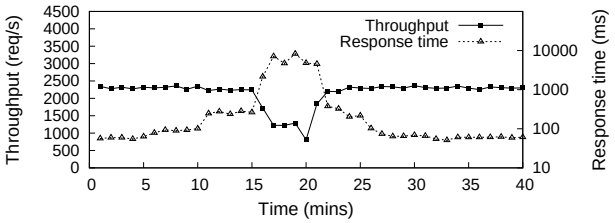


Figure 2: Delayed time in CPU reconfiguration.

We did tests measuring the dead time between a change in VCPU and the stable state. A single TPC-W DB tier was tested by changing its VCPU. Figure 2 plots the application-level performance over time. Starting from 4 VCPUs, the VM was removed one VCPU every 5 minutes until one was left. Then the VCPU was added one by one. At the $20th$ minute, the number of VCPUs varied from 1 to 2. We observed a delay time as long as 3 minutes before the throughput and response time stabilized. The reason for the delay was due to the resource contention caused by the backlogged requests when there were more CPU available. The VM took 3 minutes to digest the congested requests.

2.3 Cluster Wide Correlation

In a public cloud, multi-tier applications spanning multiple physical hosts require all tiers to be configured appropriately. In most multi-tier applications, request processing involves several stages at different tiers. These stages are usually synchronous in the sense that one stage is blocked until the completion of other stages on other tiers. Thus, the change of the capacity of one tier may affect the resource requirement on other tiers. In Table 1, we list the resource usage on the frontend application tier of TPC-W as the CPU capacity of the backend tier changed. APP MEM refers to the minimum memory size that prevents the application server from doing significant swapping I/Os; APP CPU% denotes the measured CPU utilization. The table suggests that, as the capacity of the backend tier increases, the demand for memory and CPU in the front tier decreases considerably. A possible explanation is that without prompt completion of requests at the backend tier, the front tier needs to spend resources for unfinished requests. Therefore, any mistake in one VM's capacity management may spread to other hosts. In the worst case, all nodes in the cloud infrastructure could be correlated by multi-tier applications.

In summary, the aforementioned challenges in cloud computing brings unique requirements to VM capacity management. (1) It should guarantee VM's application-level performance in the presence of complicated resource to performance relationships. (2) It should appropriately resize the VMs in response to time-varing resource demands. (3) It should be able to work in an open cloud environment, without any assumptions about VM membership and deployment topology. In next section, we present iBalloon, which is carefully designed to meet these requirements and to achieve the objective of high autonomy, efficiency and good scalability.

3. THE DESIGN OF IBALLOON

In this section, we present iBalloon, a self-adaptive VM capacity management framework.

Table 1: Configuration correlations in multi-tier application.

DB VCPU	1VCPU	2VCPU	3VCPU	4VCPU
APP MEM	790MB	600MB	320MB	290MB
APP CPU%	61%	47%	15%	10%

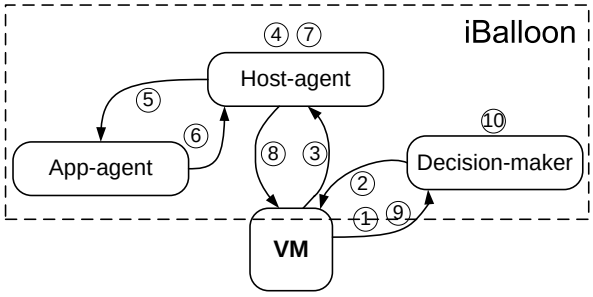


Figure 3: (1) The VM reports running status. (2) `Decision-maker` replies with a capacity suggestion. (3) The VM submits the resource request. (4) `Host-agent` synchronously collects all VMs' requests, reconfigures VM resources and sleeps for a management interval. (5)-(6) `Host-agent` queries `App-agent` for VMs' application-level performance. (7)-(8) `Host-agent` calculates and sends the feedback. (9) The VM wraps the information about this interaction and reports to `Decision-maker`. (10) `Decision-maker` updates the capacity management policy for this VM accordingly.

3.1 Overview

We design iBalloon as a distributed management framework, in which individual VMs initialize the capacity management. iBalloon provides the hosted VMs with capacity directions as well as evaluative feedbacks. Once a VM is registered, iBalloon maintains its application profile and history records that can be analyzed for future capacity management. For better portability and scalability, we decouple the functionality of iBalloon into three components: `Host-agent`, `App-agent` and `Decision-maker`.

Figure 3 illustrates the architecture of iBalloon as well as its interactions with a VM. `Host-agent`, one per physical machine, is responsible for allocating the host's hardware resource to VMs and gives feedback. `App-agent` maintains application SLA profiles and reports run-time application performance. `Decision-maker` hosts a learning agent for each VM for automatic capacity management. We make two assumptions on the self-adaptive VM capacity management. First, capacity decisions are made based on VM running status. Second, a VM relies on the feedback signals, which evaluates previous capacity management decisions, to revise the policy currently employed by its learning agent.

The assumptions together define the VM capacity management task as an autonomous learning process in an interactive environment. The framework is generic in the sense that various learning algorithms can be incorporated. Although the efficacy or the efficiency of the capacity management may be compromised, the complexity of the management task does not grow exponentially with the number of VMs or the number of resources. After a VM submits its SLA profile to `App-agent` and registers with `Host-agent` and

`Decision-maker`, iBalloon works as illustrated in Figure 3. iBalloon considers the VM capacity to be multidimensional, including CPU, memory and I/O bandwidth. As discussed in Section 2, a VM's capacity changes by altering the VCPU number, memory size and I/O bandwidth. The management operation to one VM is defined as the combination of three meta operations on each resource: *increase*, *decrease* and *nop*.

3.2 Key Designs

3.2.1 VM Running Status

VM running status has a direct impact on management decisions. A running status should provide insights into the resource usage of the VM, from which constrained or over-provisioned resource can be inferred. We define the VM running status as a vector of four tuples.

$$(u_{cpu}, u_{io}, u_{mem}, u_{swap}),$$

where u_{cpu}, u_{io}, u_{mem}, u_{swap} refer to the utilization of CPU, I/O, memory and disk swap. As discussed above, memory utilization can not be trivially calculated, we select the guest OS reported metric to calculate u_{mem} [3]. Since disk swapping activities are closely related to memory usage, adding u_{swap} to the running status provides insights on memory idleness and pressure.

3.2.2 Feedback Signal

The feedback signal ought to explicitly punish the resource allocations that lead to degraded application performance, and meanwhile encouraging a free-up of unused capacity. It also acts as an arbiter when resource are contented. We use a real-valued *reward* as the feedback. Whenever there is a conflict in the aggregated resource demand, e.g. the available memory becomes less than the total requested memory, iBalloon set the reward to be -1 (penalty) for the VMs that require an increase in the resource and a reward of 0 (neural) to other VMs. In this way, some of the conflicted VMs may back-off leading to contention resolution. Note that, although conflicted VMs may give up previous requests, `Decision-maker` will suggest the second best plan, which may be the best solution to the resource contention.

When there is no conflict on resources, the reward directly reflects application performance and resource efficiency. We define the reward as a ratio of *yield* to *cost*:

$$reward = \frac{yield}{cost},$$

$$yield = Y(x_1, x_2, \ldots, x_m) = \frac{\sum_{i=1}^{m} y(x_i)}{m},$$

$$y(x_i) = \begin{cases} 1 & \text{if } x_i \text{ satisfies its SLA;} \\ e^{-r*(|\frac{x_i - x_i^{'}}{x_i^{'}}|)} - 1 & \text{otherwise.} \end{cases}$$

$$cost = 1 + \frac{\sum_{i=1}^{n} (1 - u_i^k)^{\frac{1}{k}}}{n}.$$

where *yield* is the summarized gain over m performance metrics $x_1, x_2, \cdots, x_m$. The utility function $y(x_i)$ decays [4] when metric x_i violates its performance objective $x_i^{'}$ in SLA. *cost* is calculated as the summarized utility based on n utilization status $u_1, u_2, \cdots, u_n$. We consider throughput and response time as the performance metrics and u_{cpu}, u_{io}, u_{mem} as the utilization metrics.

The *reward* punishes any capacity plan that violates the SLA and gives incentives to high resource efficiency.

3.2.3 Self-adaptive Learning Engine

At the heart of iBalloon is a self-adaptive learning agent responsible for each VM's capacity management. Reinforcement learning is concerned with how an agent ought to take actions in a dynamic environment so as to maximize a long term reward [25]. It fits naturally within iBalloon's feedback driven, interactive framework. RL offers opportunities for highly autonomous and adaptive capacity management in cloud dynamics. It assumes no priori knowledge about the VM's running environment. It is able to capture the delayed effect of reconfigurations to a large extent.

A RL problem is usually modeled as a *Markov Decision Process* (MDP). Formally, for a set of environment states $\mathcal{S}$ and a set of actions $\mathcal{A}$, the MDP is defined by the transition probability $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ and an immediate reward function $R = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s']$. At each step t, the agent perceives its current state $s_t \in \mathcal{S}$ and the available action set $\mathcal{A}(s_t)$. By taking action $a_t \in \mathcal{A}(s_t)$, the agent transits to the next state s_{t+1} and receives an immediate reward r_{t+1} from the environment. The value function of taking action a in state s can be defined as:

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

where $0 \leq \gamma < 1$ is a discount factor helping $Q(s, a)$'s convergence.

The optimal policy is as simple as: always select the action a that maximizes the value funtion $Q(s, a)$ at state s. Finding the optimal policy is equivalent to obtain an estimation of $Q(s, a)$ which approximates its actual value. The estimate of $Q(s, a)$ can be updated each time an interaction (s_t, a_t, r_{t+1}) is finished:

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

where α is the learning rate. The interactions consist of exploitations and explorations. Exploitation is to follow the policy obtained so far; in contrast exploration is the selection of random actions to capture the change of environment so as to refine the existing policy. We follow the ϵ-greedy policy to design the RL agent. With a small probability ϵ, the agent picks up a random action, and follows the best policy it has found for the rest of the time. See [21] for more details on RL for automatic cloud management.

In VM capacity management, the state s corresponds to the VM's running status and action a is the management operation. For example, the action a can show in the form of $(nop, increase, decrease)$, which indicates an increase in the VM's memory size and a decrease in I/O priority. Actions in continuous space remains an open research problem in the RL field, we limit the RL agent to discrete actions. The actions are dicretized by setting steps on each resource

[3] See Section 4 for details

[4] r controls how fast the *yield* decays. k controls how fast cost decays with utilization.

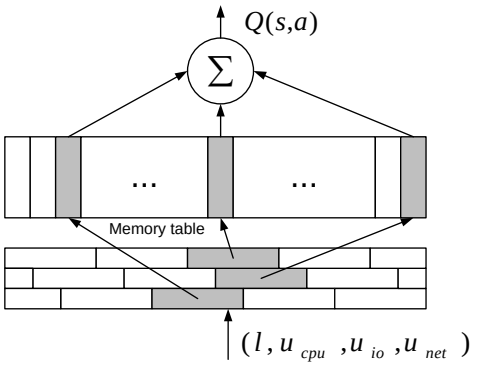


Figure 4: CMAC-based Q table.

instead. VCPU is incremented or decremented by one at a time and memory is reconfigured in a step of 256MB. I/O bandwidth is changed by a step of 0.5MB.

The autonomous VM capacity management in a cloud environment poses two key questions on the design of the RL engine. First, how to overcome the scalability and adaptability problems in RL? Second, how would the multiple RL agents, each of which represents a VM, coordinate and optimize sytem-wide performance. We answer the questions by designing the VM capacity management agent as a distributed RL agent with a highly efficient representation of the Q table. Unlike, multi-agent RL, in which each agent needs to maintain other competing agents' information, distributed RL does not have explicit coordination scheme. Instead, it relies on the feedback signals for coordination. An immediate benefit is that the complexity of the learning problem does not grow exponentially with the number of VMs.

The VM running status is essentially defined in a multi-dimensional continuous space. Although we limit the actions to be discrete operations, the state itself can render the Q value function intractable. We select the Cerebellar Model Articulation Controller (CMAC) [2] to represent the Q function. It maintains multiple coarse-grained Q tables or so-called tiles, each of which is shifted by a random offset with respect to each other. With CMAC, we can achieve higher resolution in the Q table with less cost. For example, if each status input (an element in the status vector) is discretized to four intervals (a resolution of 25%), 32 tiles will give a resolution less than 1% (25%/32). The total size of the Q tables is reduced to $32*4^4$ compared to the size of 100^4 if plain look-up table is used. In CMAC, the actual Q table is stored in a large one-dimensional memory table. Figure 4 illustrates the architecture of a one-dimensional CMAC.[5] Given a state s, CMAC uses a hash function, which takes a pair of state and action as input, to generate indexes for the (s, a) pair. CMAC uses the indexes to access the memory table and calculates the $Q(s, a)$ as the sum of all the weights accessed.

One advantage of CMAC is its efficiency in handling limited data. Similar VM states will generate CMAC indexes with a large overlap. Thus, updates to one state can generalize to the others, leading to accelerated RL learning process. Tthe update of the CMAC-based Q table only needs

[5]The VM running status listed in Figure 4 is only for illustration purpose. The state needs to work with a four-dimensional CMAC.

Algorithm 1 Update the CMAC-based Q value function

1: **Input** s_t, a_t, s_{t+1}, r_t;
2: **Initialize** $\delta = 0$;
3: $I[a_t][0] = get\_index(s_t)$;
4: $Q(s_t, a_t) = \sum_{j=1}^{j \le num\_tilings} Q[I[a_t][j]]$;
5: $a_{t+1} = get\_next\_action(s_{t+1})$;
6: $I[a_{t+1}][0] = get\_index(s_{t+1})$;
7: $Q(s_{t+1}, a_{t+1}) = \sum_{j=1}^{j \le num\_tilings} Q[I[a_{t+1}][j]]$;
8: $\delta = r_t - Q(s_t, a_t + \gamma * Q(s_{t+1}, a_{t+1}))$;
9: **for** $i = 0; i < num\_tilings; i++$ **do**
10: /*If SLO violated, enable fast adaptation*/
11: **if** $r_t < 0$ **then**
12: $\theta[I[a_t][i]] += (1.0/num\_tilings) * \delta$;
13: **else**
14: $\theta[I[a_t][i]] += (\alpha/num\_tilings) * \delta$;
15: **end if**
16: **end for**

6.5 milliseconds in our testbed, in comparison with the 50-second update in neural network based approximator [21]. Once a VM finishes an iteration, it submits the four-tuple (s_t, a_t, s_{t+1}, r_t) to the `Decision-maker`. Then the corresponding RL agent updates the VM's Q table using Algorithm 1. In the algorithm, we further enhanced the CMAC-based Q table with fast adaptation when SLA violated. We set the learning rate α to 1 whenever receives a negative penalty. This ensures that "bad" news travels faster than good news allowing the learning agent quickly response to the performance violation.

4. IMPLEMENTATION

iBalloon has been implemented as a set of user-level daemons in guest and host OSes. The communication between the host and guest VMs is carried out through an interdomain channel. In our Xen-based testbed, we used `Xenstore` for the host and guest information exchange. Xenstore is a centralized configuration database that is accessible by all domains on the same host. The domains who are involved in the communication place "watches" on a group of predefined keys in the database. Whenever sender initializes a communicatin by writing to the key, the receiver is notified and possibly trigging a callback function. Upon a new VM joining a host, the `Host-agent` creates a new key under the VM's path in Xenstore. `Host-agent` launches a worker thread for the VM and the worker "watches" any change of the key. Whenever a VM submits a resource request via the key, the worker thread retrieves the request details and activates the corresponding handler in the `dom0` to handle the request. The VM receives the feedback from `Host-agent` in a similar way.

We implemented resource allocation in `dom0` in a synchronous way. VMs send out resource requests in a fixed interval (30 second in our experiments) and the `Host-agent` waits for all the VMs before satisfying any request. It is often desirable to allow users to submit requests with different management intervals for flexibility and reliability in resource allocation. We leave the extension of iBalloon to asynchronous resource allocation in the future work. After VMs and the `Host-agent` agree on the resource allocations, the `Host-agent` modifies individual VMs' configurations accordingly. We change the memory size of the VM by writing the new size to the domain's `memory/target` key in Xenstore. VCPU number is altered by turning on/off individual CPUs via `cpu/CPUID/availability`. For I/O priorities, we use command `lsof` to correlate VMs' virtual disks to processes

and change the corresponding processes' priorities via Linux command `ionice`.

The `App-agent`, one per host, maintains the hosted application SLA profiles. In our experiments, it periodically queries participant machines through standard socket communication and reports application performance, such as throughput and response time, to the `Host-agent`. In a more practical scenario, the application performance should be reported by a third-party application monitoring tool instead of the clients. iBalloon can be easily modified to integrate such tools.

We also consider two possible the implementations of the `Decision-maker`.

1. **Centralized decision center.** In this approach, a designated server maintains all the Q learning tables of the VMs. Although centralized in maintainence of the learning trace, VMs' capacity management decisions are independent with each other. The advantages include the simplicity of management: learning algorithms can be modified or reused across a group of VMs; avoidance of performance overhead: the possible overhead incurred by the learning is removed from individual VMs. However, the centralized server can become a single point of failure as well as performance bottleneck as the number of VMs increases. We use asynchronous socket and multi-threads to improve concurrency in the server.

2. **Distributed decision agent.** In this approach, learning is local to individual VMs and the `Decision-maker` is a process residing in the guest OS. The scalability of iBalloon is not limited by the processing power in the centralized decision server, but at a cost of CPU and memory overhead in each VM.

Trade-offs between the two approaches are studied in Section 5.8.

We use `xentop` utility to report VM CPU utilization. `xentop` is instrumented to redirect the utilization of each VM to separate log files in the `tmpfs` folder `/dev/shm` every second. A small utility program parses the logs and calculates the average CPU utilization for every management interval. The disk I/O utilization is calculated as a ratio of achieved bandwidth to allocated bandwidth. The achieved the bandwidth can be obtained by monitoring the disk activities in `/proc/PID/io`. PID is the process number of a VM's virtual disk in `dom0`. The swap rate can also be collected in a similar way. We consider memory utilization to be the guest OS metric `Active` over memory size. The `Active` metric in `/proc/meminfo` is a coarse estimate of actively used memory size. However, it is lazily updated by guest kernel especially during memory idle periods. We combine the guest reported metric and swap rate for a better estimate of memory usage. With explorations from the learning engine, iBalloon has a better chance to reclaim idle memory without causing significant swappings. The CMAC-based RL was implemented by integrating Rutton's CMAC library [1].

5. EXPERIMENT RESULTS

5.1 Methodology

To evaluate the efficacy of iBalloon, we attempt to answer the following questions: (1) How well does iBalloon perform in the case of single VM capacity management? (2) Can the learned capacity management policy be re-used to control a similar application or on a differnt platform? (3) When there is resource contention, can iBalloon properly distribute the constrained resource and optimize overall system performance? (4) How is iBalloon's scalability and overhead? We selected two representative server workloads as the hosted applications. TPC-W [28] is an E-Commerce benchmark that models after an online book store, which is primary CPU-intensive. It consists of two tiers, i.e. the frontend application (APP) tier and the backend database (DB) tier. TPC-C [29] is an online transaction processing benchmark that contains a large amount of lightweight disk reads and sporadic heavy writes. Its performance is sensitive to memory size and I/O bandwidth.

To create dynamic variations in resource demand, we instrumented the workload generators of TPC-W and TPC-C to change client traffic level at run-time. The workload generator reads the traffic level from a trace file, which models after the real Internet traffic pattern [27]. We scaled down the Internet traces to match the capacity of our platform.

5.2 Experiment Settings

Two clusters of nodes were used for the experiments. The first cluster (CIC100) is a shared research environment, which consists of a total number of 22 DELL and SUN machines. Each machine in CIC100 is equipped with 8 CPU cores and 8GB memory. The CPU and memory configuration limit the number of VMs that can be consolidated on each machine. Thus, we use CIC100 as a resource constrained cloud testbed to validate iBalloon's effectiveness for small scale capacity management with resource contention. Once iBalloon gains enough experiences to make decisions, we applied the learned policies to manage a large number of VMs. CIC200 is a cluster of 16 DELL machines dedicated to the cloud management project. Each node features a configuration of 12 CPU cores (with hyperthreading enabled) and 32 GB memory. In the scale-out testing, we deployed 64 TPC-W instances, i.e. a total number of 128 VMs on CIC200. To generate sufficient client traffic to these VMs, all the nodes in CIC100 were used to run client generators, with 3 copies running on each node. The experiments were limited to a LAN environment which is interconnected by Gigabyte Ethernet network.

We used Xen version 4.0 as our virtualization environment. `dom0` and guest VMs were running Linux kernel 2.6.32 and 2.6.18, respectively. To enable real-time reconfiguration of CPU and memory, all the VMs were para-virtualized. The VM disk images were stored locally on a second hard drive on each host. We created the `dm-ioband` device mapper on the partition containing the images to control the disk bandwidth to each VM. For the benchmark applications, MySQL, Tomcat and Apache were used for database, application and web servers.

5.3 Evaluation of the Reward Metric

The *reward* metric syncretizes multi-dimensional application performance and resource utilizations. We are interested in how the *reward* signal directs the capacity management. The decay rates r and k reflect how important it is for an application to meet the performance objectives in its SLA and how aggressive of the user to increase resource utilization even at the risk of overload. Figure 5 plots the application *yield* with different decay rate r. The *reward*

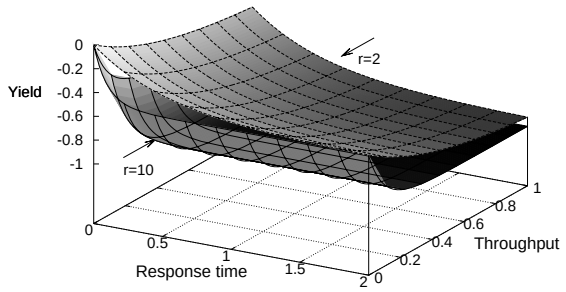


Figure 5: Application yield with different decay rates.

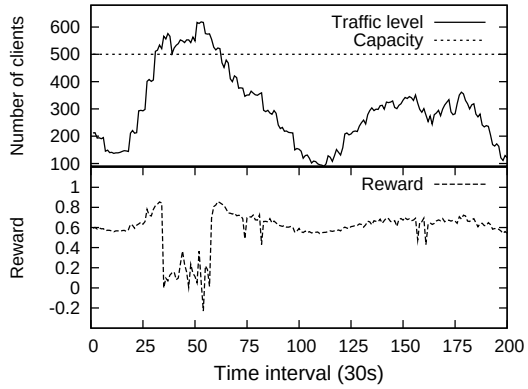


Figure 6: Reward with different traffic levels.

data was from a 2-hour test run of TPC-W with limited resources. During the experiment, there were considerable SLA violations. The x and y axes show the difference (in percentage) between the achieved performance and the objective when SLA is violated. The larger the difference, the more deviation from the target. From the figure, we can see that a larger decay rate poses more strict requirement on meeting SLA. A small deviation from the target will effectively generate a large penalty. Similarly, k controls how the *cost* changes with resource usage. A larger value of k encourages the user to use the resource aggressively. Finally, we decided to guarantee user satisfaction and assume risk neutral users, and set $r = 10$ and $k = 1$.

Recall that the *reward* is defined as the ratio of *yield* and *cost*, Figure 6 shows how the *reward* can reflect the status of VM capacity. In this experiment, we varied the client traffic to occasionally exceed the VM's capacity. *reward* is calculated from the DB tier of a TPC-W instance, with a configuration of 3 VCPU, 2GB memory and 2 MB/s disk bandwidth. As shown in the Figure 6, when there is light load, performance objectives are met. During this period, *yield* is set to be 1 and *cost* dominates the change of *reward*. As traffic increases, resource utilization goes up incurring smaller *cost*. Similarly, *reward* drops when traffic leaves because *cost* goes up again. In contrast, when the VM is overloaded with SLA violations, *yield* takes over *reward* by imposing a large penalty. In conclusion, *reward* effectively punishes performance violations and gives users incentives to release unused resources.

5.4 Exploitations vs. Explorations

Reinforcement learning is a direct adaptive optimal control approach which relies on the interactions with the environment. Therefore, the performance of the learning algorithm depends critically on how the interactions are defined. Explorations are often considered as sub-optimal actions that lead to degraded performance. However, without enough explorations, the RL agent tends to be trapped in local optimal policies, failing to adapt to the change of the environment. On the other hand, too much exploration would certainly result in unacceptable application performance. Before iBalloon is actually deployed, we need to determine the value of ϵ, i.e. the exploration rate, that best fits our platform.

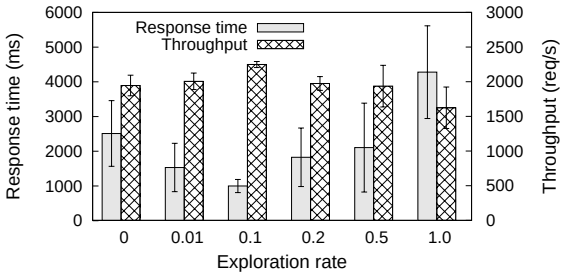


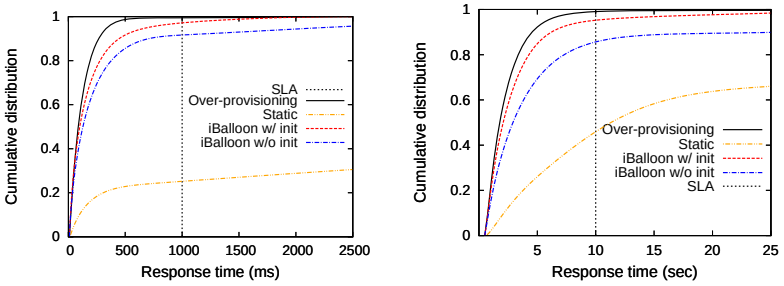
Figure 7: Application performance with different exploration rates.

In this experiment, we dedicated a physical host to one application and initialized the VM's Q table to all zeros. We varied the exploration rate of the learning algorithm and plot the application performance of TPC-W in Figure 7. In Figure 7, the bars represent the average of 5 one-hour runs with the same exploration rate. The error bars plot the variations. From the figure, we can see that the response time of TPC-W is convex with respect to the exploration rate with $\epsilon = 0.1$ being the optimal. The same exploration rate also gives the best throughput as well as the smallest variations. Experiments with TPC-C suggested the similar ϵ value. We also empirically determined the learning rate and discount factor. For the rest of this paper, we set the RL parameters to the following values: $\epsilon = 0.1, \alpha = 0.1, \gamma = 0.9$

5.5 Single Application Capacity Management

In its simplest form, iBalloon manages a single VM or application's capacity. In this subsection, we study its effectiveness in managing different types of application with distinct resource demands. As discussed in [21], the RL-based auto-configuration can suffer from initially poor performance due to explorations with the environment. To have a better understanding of the efficiency of RL-based capacity management, we tested two variations of iBalloon, one with an initialization of the management policy and one without. We denote them as *iBalloon w/ init* and *iBalloon w/o init*, respectively. The initial policy was obtained by running the application workload for 10 hours, during which iBalloon interacted with the environment with only exploration actions.

Figure 8(a) and Figure 8(b) plot the performance of iBalloon and its variations in a 5-hour run of the TPC-W and TPC-C workloads. Note that during each experiment, the host was dedicated to the TPC-W or TPC-C, thus no resource contention existed. In this simple setting, we can obtain the upper bound and lower bound of iBalloon's performance. The upper bound is due to resource *over-provisioning*, which allocates more than enough resource for the applications. The lower bound performance was derived from a



(a) TPC-W (b) TPC-C

Figure 8: The CDF of response time.

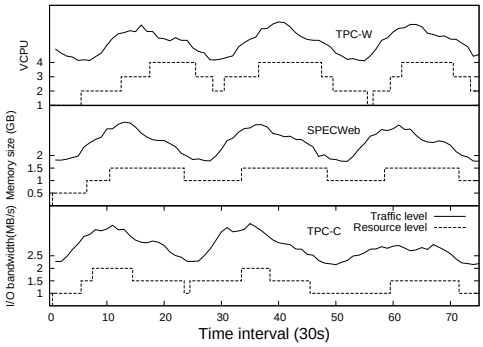
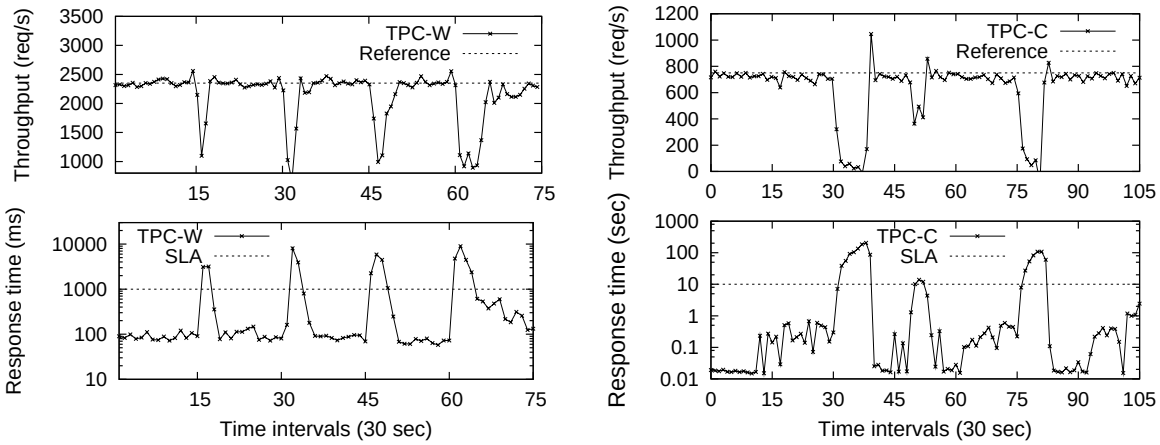


Figure 9: The effect of incentives.



(a) TPC-W (b) TPC-C

Figure 10: User-perceived performance under iBalloon.

VM template whose capacity is not changed during the test. We refer it as *static*. We configured the VM template with 1 VCPU and 512 MB memory in the experiment. If not otherwise specified, we used the same template for all VM default configuration in the remaining of this paper.

From Figure 8(a), we can see that, iBalloon achieved close performance compared to *over-provisioning*. Interestingly, *iBalloon w/o init* managed to keep almost 90% of the request below the SLA response time threshold except that a few percent of requests had wild response times. It suggests that, although started with poor policies, iBalloon was able to quickly adapt to good policies and maintained the performance at a stable level. We attribute the good performance to the highly efficient representation of the Q table. The CMAC-enhanced Q table was able to generize to the continuous state space with a limited number of interactions. Not surprisingly, *static*'s poor result again calls for appropriate VM capacity management.

As shown in Figure 8(b). *iBalloon w/ init* showed almost optimal performance for TPC-C workload too. But without policy initialization, iBalloon can only prevent around 80% of the requests away from SLA violations; more than 15% requests would have response times larger than 30 second. This barely acceptable performance stresses the importance of a good policy in more complicated environments. Unlike CPU, memory sometimes shows unpredictable impact on performance. Its delayed time is much longer than CPU (10 minutes compared to 3 minutes in our experiments). In this case, iBalloon needs a longer time to obtain a good policy. Fortunately, the derived policy, which is embedded in the Q table, is only based on generic system statistics. It can possibly be re-used to manage similar applications.

Table 2: Application speedup with learned policy.

	Throughput	Response time
SPECweb	1.4	1.8
CIC200	1.2	1.3

Table 2 lists the application speedup if the learned management policies are applied to a different application or to a different platform. The speedup is calculated against the performance of iBalloon without an initial policy. SPECweb [24] is a web server benchmark suite that contains representative web workloads. The E-Commerce workload in SPECweb is similar to TPC-W (CPU-intensive) except that its performance is also sensitive to memory size. Results in Table 2 suggest that the Q-table learned for TPC-W also worked for SPECweb. An examination of iBalloon's log revealed that the learned policy was able to successfully match CPU allocation to incoming traffic. A policy learned on cluster CIC100 can also give more than 20% performance speedup to the same TPC-W application on cluster CIC200. Given

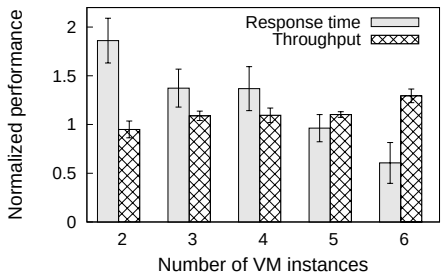


Figure 11: iBalloon with multiple applications.

the fact that the nodes in CIC100 and CIC200 have more than 30% difference on CPU speed and disk bandwidth, we conclude that iBalloon policies are applicable to heterogeneous platforms across cloud systems.

The *reward* signal provides the VMs with strong incentives to give up unnecessary resources. In Figure 9, we plot the configuration of VCPU, memory and I/O bandwidth of TPC-W, SPECweb and TPC-C as client workload varied. Recall that we do not have an accurate estimation of memory utilization. We rely on the `Active` metric in `meminfo` and the swap rate to infer memory idleness. The Apache web server used in SPECweb periodically free unused `httpd` process thus memory usage information in `meminfo` is more accurate. As shown in Figure 9, with a 10-hour trained policy, iBalloon successfully expanded and shrunk CPU and I/O bandwidth resources as workload varied. As for the memory, iBalloon was able to quickly response to memory pressure and release part of the unused memory. For example, when traffic dropped, iBalloon stopped at 1GB instead of the optimal 512MB, waiting for the

We note that the above results only show the performance of iBalloon statistically. In practice, service providers concern more about user-perceived performance, because in production systems, mistakes made by autonomous capacity management can be prohibitively expensive. To test iBalloon's ability of determing the appropriate capacity online, we ran the workload generators at full speed and randomly changed the VM's capacity every 15 management intervals. Figure 10(a) and Figure 10(b) plot the client-perceived throughput and response time in TPC-W and TPC-C respectively. In both experiments, iBalloon was configured with initial policies. Each point in the figures represents the a 30-second management interval. As shown in Figure 10(a), iBalloon is able to promptly detect the misconfigurations and reconfig the VM to appropriate capacity. On average, the throughput and response time can be recovered within 7 management intervals. Similar results can also be observed in Figure 10(b) except that the client-perceived response times have larger fluctuations in TPC-C workload.

5.6 Coordination in Multiple Applications

iBalloon is designed as a distributed management framework that handles multiple applications simultaneously. The VMs rely on the feedback signals to form their capacity management policy. Different from the case of a single application, in which the feedback signal only depends on the resource allocated to the hosting VM, in multiple application hosting, the feedback signals also reflect possible performance interferences between VMs.

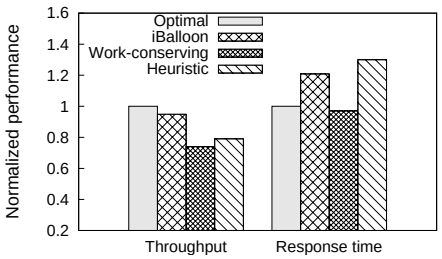


Figure 12: iBalloon scales to 128 correlated VMs.

We designed experiments to study iBalloon's performance in coordinating multiple applications. Same as above, iBalloon was configured to manage only the DB tiers of TPC-W workload. All the DB VMs were homogeneously hosted in one physical host while the APP VMs were over-provisioned on another node. The baseline VM capacity strategy is to statically assign 4VCPU and 1GB memory to all the DB VMs, which is considered to be over-provisioning for one VM. iBalloon starts with a VM template, which has 1VCPU and 512MB memory. Figure 11 plots the performance of iBalloon compared to the baseline capacity strategy in a 5-hour test. The plotted throughput and response time are normalized to the baseline strategy. The workload to each TPC-W instances varied dynamically, but the aggregated resource demand is beyond the capacity of the machine that hosts the DB VMs. Figure 11 shows that, as the number of the DB VMs increases, iBalloon gradually beats the baseline in both throughput and response time.

An examination of the iBalloon logs revealed that iBalloon suggested a smaller number of VCPUs for the DB VMs, which possibly alleviated the contention for CPU. As discussed in Section 2, the baseline strategy encouraged resource contention and resulted in wasted work. In summary, iBalloon, driven by the feedback, successfully coordinated the competing VMs to use the resource more efficiently.

5.7 Scalability

In this subsection, we scale iBalloon out to multiple hosts and do not assume the homogeneous VM deployment as above. 64 TPC-W instances, each has two tiers, were consolidated on the CIC200 cluster. We randomly deployed the 128 VMs to the 16 nodes. To avoid possible hotspot and load unbalancing, we make sure that each node hosts 8 VMs, 4 APP and 4 DB tiers. The deployment is challenge to autonomous capacity management for two reasons. First, iBalloon ought to coordinate VMs on different hosts, each of which runs its own resource allocation policy. The dependent relationships makes it harder to orchestrate all the VMs. Second, consolidating APP (network-intensive) tiers with DB (CPU-intensive) tiers onto the same host poses challenges in finding the balanced configuration.

Figure 12 plots TPC-W performance for a 10-hour test. In addition to iBalloon, we also include three other strategies. The *optimal* strategy was obtained by tweaking the cluster manually. It turned out that the setting: DB VM with 3VCPU,1GB memory and APP VM with 1VCPU, 1GB memory delivered the best performance. *work-conserving* scheme is similar to the baseline in last subsection; it sets the VMs with fixed 4VCPU and 1GB memory. The *heuristic* is a widely used simple policy that requests more resource when utilization is high and release it once utilization is low. The high and low threshold were set to 80% and 25%. The

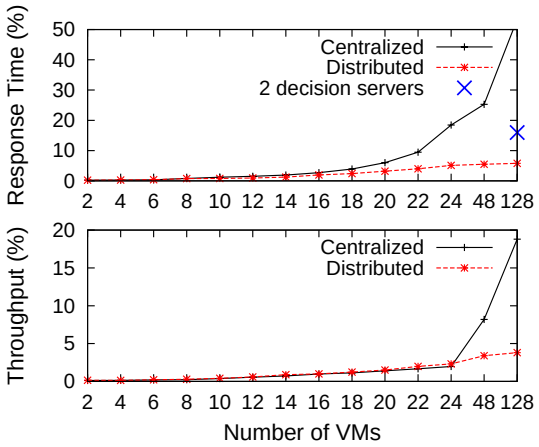


Figure 13: Runtime overhead of iBalloon.

performance is normalized to *optimal* scheme. For throughput, the high the better; for response time, lower is better.

From the figure, iBalloon achieved close throughput as the *optimal* while incurred 20% degradation on request latency. This is understandable because any change in a VM's capacity, especially memory reconfigurations, brings in unstable periods. iBalloon outperformed the *work-conserving* scheme by more than 20% in throughput. Although *work-conserving* had compromised throughput, it achieved similar response time as *optimal* because it did not perform any reconfigurations. *Heuristic* achieved slightly better thoughput than *work-conserving* but with almost 30% overhead on response time. In conclusion, iBalloon scales to 128 VMs on a correlated cluster with near-optimal application performance. In the next, we perform tests to narrow down the overhead incurred on request latency.

5.8 Overhead

In previous experiments, iBalloon incurs nonnegligible cost in response time. The cost is due to the real overhead of iBalloon as well as the performance degradation caused by the reconfiguration. To study the overhead incurred by iBalloon, we did the same experiment as in Section 5.7 except that iBalloon operated on the VMs with optimal configurations and the reconfiguration was disabled in the `Host-agent`. In this setting, the overhead only comes from the interactions between VMs and iBalloon. Figure 13 shows the run-time overhead of iBalloon with two different implementations of the `Decision-maker`, namely the centralized and the distributed implementations. Again, the overhead is normalized to the performance in the *optimal* scheme.

Figure 13 suggests that the centralized decision server becomes the bottleneck with as much as 50% overhead on request latency and 20% on throughput as the number of VMs increases. In contrast, the distributed decision agent, which computes capacity decisions on local VMs, incurred less than 5% overhead on both response time and throughput. To further confirm that the centralized decision server was the bottleneck, we split the centralized decision work onto two separate machines in the case of 128 VMs. As shown in Figure 13, the overhead on request latency reduces by more than a half. Additional experiments revealed that computing the capacity management decisions locally in VMs requires no more than 3% CPU resources for Q computation and approximately 18MB of memory for Q table storage. The resource overhead is insignificant compared to the capacity of the VM template (1VCPU, 512MB). These results conclude that if properly implemented, iBalloon adds no more than 5% overhead to the application performance with a manageable resource cost.

6. RELATED WORK

Cloud computing allows cost-efficient server consolidation to increase system utilization and reduce cost. Resource management of virtualized servers is an important and challenging task, especially when dealing with fluctuating workloads and performance interference. Recent work demonstrated the feasibility of statistical analysis, control theory and reinforcement learning to automatic virtual server resource allocation to some extent.

Early work [19, 23] focused on the tuning of the CPU resource only. Padala, et al. employed a proportional controller to allocate CPU shares to VM-based multi-tier applications [19]. This approach assumes non-work-conserving CPU mode and no interferece between co-hosted VMs, which can lead to resource under-provisioning. Recent work [12] enhanced traditional control theory with Kalman filters for stability and adaptability. But the work remains under the assumption of CPU allocation. The authors in [23] applied domain knowledge guided regression analysis for CPU allocation in database servers. The method is hardly applicable to other applications in which domain knowledge is not available.

The allocation of memory is more challenging. The work in [9] dynamically controlled the VM's memory allocation based on memory utilization. Their approach is application specific, in which the Apache web server optimizes its memory usage by freeing unused `httpd` processes. For other applications like MySQL database, the program tends to cache data aggressively. The calculation of the memory utilization for VMs hosting these applications is much more difficult. Xen employs Self-Ballooning [15] to do dynamic memory allocation. It estimates the VM's memory requirement based on OS-reported metric: `Commited_AS`. It is effective expanding a VM under memory pressures, but not being able to shrink the memory appropriately. More accurate estimation of the actively used memory (i.e. the working set size) can be obtained by either monitoring the disk I/O [11] or tracking the memory miss curves [33]. However, these event-driven updates of memory information can not promptly shrink the memory size during memory idleness. Although, we have not found a good way to estimate the VM's working size, iBalloon relies on a combination of performance metrics to decide memory allocation. With more information on VMs' business and past trial-and-error experiences, iBalloon has the potential in more accurate memory allocation.

Automatic allocation of multiple resources [18] or for multiple objectives [13] poses challenges in the design of the management scheme. Complicated relationship between resource and performance and often contradicted objectives prevent many work from being automatic but heuristic. Padala, et al. applied an auto-regressive-moving-average (ARMA) model with success to represent the allocation to application performance relationship. They used a MIMO controller to automatically allocate CPU share and I/O bandwidth to multiple VMs. However, the ARMA model may not be effective under steady workload and can become invalid if VM inference exists.

Different from the above approaches in designing a self-managed system, RL offers tremendous potential benefits in autonomic computing. Recently, RL has been successfully applied to automatic application parameter tuning [4, 5], optimal server allocation [26] and self-optimizing memory controller design [10]. Autonomous resource management in cloud systems introduces unique requirements and challenges in RL-based automation, due to dynamic resource demand, changing topology and frequent VM interference. More importantly, user-perceived quality of service should also be guaranteed. The RL-based methods should be scalable and highly adaptive. We attempted to apply RL in host-wide VM resource management [21]. We addressed the scalability and adaptability issues using model-based RL. However, the complexity of training and maintaining the models for the systems under different scenarios becomes prohibitively expensive when the number of VMs increases. In contrast, we fundamentally change the way resource allocation was perceived in iBalloon. In a distributed learning process, iBalloon demonstrated a scalability up to 24 correlated VMs on four nodes under work-conserving mode.

7. CONCLUSION

In this work, we present *iBalloon*, a generic framework that allows self-adaptive virtual machine resource provisioning. The heart of iBalloon is the distributed reinforcement learning agents that coordinate in dynamic environment. Our prototype implementation of iBalloon, which uses a highly efficient reinforcement learning algorithm as the learning, was able to find the near optimal configurations for a total number of 128 VMs on a closely correlated cluster with no more than 5% overhead on application throughput and response time.

Nevertheless, there are several limitations of this work. First, the management operations are discrete and are in a relatively coarse granularity. Second, the RL-based capacity management still suffers from initial performance considerably. Future work can extend iBalloon by combining control theory with reinforcement learning. This framework is very similar to the actor-critic learning in RL. It offers opportunities for the control theory to provide fine grained operations and stable initial performance.

8. REFERENCES

[1] http://webdocs.cs.ualberta.ca/šutton/tiles.html.

[2] J. S. Albus. A new approach to manipulator control: the cerebellar model articulation controller (cmac. *Journal of Dynamic Systems, Measurement, and Control*, 1975.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, Feb 2009.

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

[5] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira. Boosting the performance of computing systems through adaptive configuration tuning. In *SAC*, 2009.

[6] L. Cherkasova, D. Gupta, and A. Vahdat. When virtual is harder than real: Resource allocation challenges in virtual machine based it environments. Technical report, HP Labs, Feb 2007.

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

[8] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *OSDI*, 2008.

[9] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IM*, 2009.

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

[11] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS*, 2006.

[12] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC*, 2009.

[13] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vmanage: loosely coupled platform and virtualization management in data centers. In *ICAC*, 2009.

[14] H. A. Lagar-Cavilla, J. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Eurosys*, 2009.

[15] D. Magenheimer. Memory overcommit...without the commitment. Technical report, Xen Summit, June 2009.

[16] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: enlightened page sharing. In *Usenix Annual Technical conference*, 2009.

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

[18] 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*, 2009.

[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] P. Padala, X. Zhu, Z. Wang, S. Singhal, and S. K. G. Performance evaluation of virtualization technologies for server consolidation. Technical report, HP Labs, Sep 2008.

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

[22] J. Rao and C.-Z. Xu. Online measurement the capacity of multi-tier websites using hardware performance counters. In *ICDCS*, 2008.

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

[24] The SPECweb benchmark. http://www.spec.org/web2005.

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

[26] 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.

[27] The ClarkNet Internet traffic trace. http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html.

[28] The Transaction Processing Council (TPC). http://www.tpc.org/tpcw.

[29] The Transaction Processing Council (TPC). http://www.tpc.org/tpcc.

[30] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI*, 2002.

[31] K. Wang and C.-Z. Xu. Virtual machine substrate for rapid deployment of virtualized appliance. Technical report, Wayne State University, Nov 2009.

[32] Xen. http://www.xen.org/.

[33] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE*, 2009.