Lightweight automatic resource scaling for multi-tier web applications

Lenar Yazdanov
Faculty of Computer Science
TU-Dresden
Dresden, Germany
lenar@se.inf.tu-dresden.de

Christof Fetzer
Faculty of Computer Science
TU-Dresden
Dresden, Germany
christof.fetzer@inf.tu-dresden.de

***Abstract*—Dynamic resource scaling is a key property of cloud computing. Users can acquire or release required capacity for their applications on-the-fly. The most widely used and practical approach for dynamic scaling based on predefined policies (rules). For example, IaaS providers such as RightScale asks application owners to manually set the scaling rules. This task assumes, that the user has an expertise knowledge about the application being run on the cloud. However, it is not always true. In this paper we propose a lightweight adaptive multi-tier scaling framework VscalerLight, which learns scaling policy online. Our framework performs fine-grained vertical resource scaling of multi-tier web application. We present the design and implementation of VscalerLight. We evaluate the framework against widely used RUBiS benchmark. Results show that the application under control of VscalerLight guarantees 95th percentile response time specified in SLA.**

***Index Terms*—scalability; performance; measurement;**

I. Introduction

It is common today to use public or private clouds. Many users prefer to create or rent virtual machines(VM) in order to host their applications instead of using physical machines. Typically cloud users pay for predefined VM templates on a cloud provider side. However, the assigned VM templates are not well fit for the users. The reason is that the resource demand is not static and it is varying over the time. Therefore users allocate resources according to peak demand to achieve good performance of the application. However, the peak load provisioning leads to under-utilization and the user pays for resources which are not used. Hence, it is desirable for the users to be able to re-size a VM on-the-fly and pay for the resources currently consumed. Some of the providers such as *CloudSigma* [3] already addressed the issue and allow users to specify desired VM template and change it later during the runtime. Now it becomes possible to perform cost-effective scaling. But what is the appropriate policy to control resource allocation without affecting the application performance?

Dynamic application scaling is a non-trivial task. The scaling policy should be able to meet SLA (service level agreement) requirements despite the changing resource requirements and unpredictable interactions between the applications. The following questions arise during the process of the scaling policy implementation. *First, conversion of SLA to resource allocation.* It is difficult to determine the CPU and memory allocation required to provide certain response time of an request. *Second, time-varying resource demand.* Many applications running on the cloud have time varying resource demand. Some demands can be predicted if they have daily, monthly or seasonal patterns, otherwise it is difficult to provide high prediction accuracy. For example, unexpected raise of popularity. Hence, if an application provisioned according to the average load. The performance degrades, if peaks of load occur. Therefore dynamic scaling policy should have predictive and reactive components. *Third, multi-tier applications scaling.* Multi-tier applications require appropriate resource allocation across all nodes to provide the performance specified in SLA. Hence, to build a resource allocation controller it is necessary to create a mathematical model which captures relationships between the amount of provisioned resources and the application performance.

Many existing work [10, 8, 18, 16, 12, 6] addressed the problem of dynamic resource allocation. These works apply different techniques to obtain resource scaling policy. However all of them require offline experiments to design optimal formal model. The drawback of the offline approach is that the changes of the application such as updates require redesign of the model. There are works [21, 19] and [17, 12, 13] which overcome the problem of offline policy learning. The first group applies so called 'sand-box' approach. The idea is to tune the scaling policy online in dedicated sand-box machines without stopping a production system. The approach successfully adapts the policy. However, it creates additional complexity and resource overhead associated with set-up of 'sand-box'. The second group uses reinforcement learning (RL) approach. It is a model free technique, which does not require *a priori* knowledge about the application performance model. It learns the application behavior online, while the application is running. However, algorithms for RL suffer from what is known as the curse of dimensionality: an exponential explosion in the total number of states as a function of the number of state variables. In our previous work [20] we used speed technique to improve the agent learning time. However we did not address the problem of the state-space size reduction.

In this paper, we describe our self-adaptable resource scaling framework VscalerLight The core of VscalerLight is RL

approach. Our design is based on analysis of the application behavior under different resource allocation configurations. With the help of the analysis of the results we propose to split memory and CPU controller models instead of tightening them together which is common for works which use RL. The use of an individual controller per each resource allows to reduce the state-space complexity and eliminate well known problem of RL based controllers. Moreover, we apply appropriate initialization policy and add workload dynamics to the state description of each resource controller.

II. Motivation and RL problem

Cloud computing provides a mechanisms for delivery computing resources on-demand over the network. These resources are shared among many users all around the world. So any user can rent resources for a relatively low price in comparison to buying hardware. The user running an application on the cloud expects certain performance from the application such as low response time, high throughput. Many web applications deal with highly fluctuating workloads. If the increase of the load is possible to predict (e.g. promotion campaign, seasonal change) then additional resources can be provisioned to an application in advance. However, for unplanned load spikes auto-scaling mechanism is required. It should be able to automatically tune the amount of resources assigned based on the application needs for given moment of time.

Cloud providers do not have much knowledge about the user application running in the cloud, especially if the application is deployed for the first time. It makes difficult to provide correct resource allocation policy for the application. Therefore most of cloud providers use easy and lightweight scaling policy based on thresholds. The idea of the approach is to assign or release certain resource based on predefined threshold. For example, when CPU utilization reaches 60% a new VM is allocated and the VM is deallocated if CPU utilization drops below 30%. Usually user has to set these thresholds. It means that the user has to have expertise knowledge or evaluate the application offline in order to define the 'right' threshold. But even these conditions are not enough. Different workload patterns may require different thresholds. There are some variations of threshold policy such the one based on voting system [14] and having additional intermediate thresholds [7].

Most of cloud providers support only horizontal scaling of the user application(a number of VMs can be added and removed). Nevertheless, there are cloud providers which allow to scale applications vertically [3]. Vertical scaling means that the user can change a VM capacity on-the fly by adding/removing CPU or memory. In comparison to horizontal scaling, vertical scaling has lower resource allocation latency. Adding virtual CPU to the VM takes less than 1 second, while starting a new VM instance can take about 5 minutes [6]. Moreover, it is not always possible to scale an application horizontally. It is true for state full applications. For example one cannot just add and additional VM instance of MySQL DB on-the-fly, since it requires data transfer to the new instance.

The problem described above inspired us to look for an approach which provides scaling policy learning properties. So we can eliminate offline tuning. RL gives us the necessary properties. Moreover we would like take advantage of using vertical scaling and change CPU and memory allocation to a VM. However, it leads to a grow of state-space size. In the next sections we will describe how we can reduce state-space complexity and improve learning time.

A. RL problem

RL problems can be modeled as a Markov Decision Process (MDP). A MDP can typically be represented as a four tuple consisting of states, actions, transition probabilities and rewards.

- S, represents environmental state space
- A, represents total action space
- $p(. \mid s, a)$, defines a probability distribution governing state transitions $s_{t+1} \sim p(. \mid s_t, a_t)$
- $q(. \mid s, a)$, defines a probability distribution governing the rewards received $R(s_t, a_t) \sim q(. \mid s_t, a_t)$

S, the set of all possible states represents the agents observable world. At each, step t, the agent perceives its current state $s_t \in S$ and the available action set $A(s_t)$. By taking an actions $a_t \in A(s_t)$, the agent moves to the next state s_{t+1} and receives an immediate reward r_{t+1} from the environment.

In cases when environmental model is not known model-free reinforcement learning algorithms are used. Q-learning belongs to the collection of these algorithms. The update rule in Q-learning of taking action a in state s defined as:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma max_a Q(s_{t+1}, a_{t+1}) - Q(s,a)] \tag{1}$$

where α is learning rate and γ is a discount factor. The interactions in RL consist of exploitations and explorations. Exploitation is to follow the policy obtained so far. The policy is to take an action which has highest Q-value. Exploration is the selection of random actions to capture the change of environment to prevent from maximizing the short-term reward. An agent is designed by following ε-greedy policy. With a small probability, the agent picks up a random action, and follows the best policy it has found for the rest of the time.

To facilitate agent learning for a cloud resource allocation problem, one must define an proper state-action space formalism. In cloud resource management the state s corresponds to the amount of resources allocated. The state can be represented as number of running VMs [2, 5] and action a is amount of VMs which can be added, removed or maintained. If we want to scale vertically, then the state s represents VM's running status. It can be defined as a number of CPUs and memory assigned to a VM [12, 13, 20]. The action a indicates the change of the VM capacity, which can be shown in form of (CPU, RAM). For example $a = (10, 16)$ means an increase in CPU capacity by value of 10 and increase in memory size by value of 16. The state-space size depends on the

number of variables which define the state and number of actions available in the state. For example, if the state contains two variables and each variable has n different values and m actions available for each state. Then the state-space size is $n * n * m$. The increase of n and m would improve the granularity of resource allocation, but it requires more time to collect the reward samples for all state-action pairs.

We showed that RL approaches generally suffer from so called curse of dimensionality problems, as the size of the state space grows exponentially with each new state variable added. As result the learning time dramatically increases. Even if speedup techniques are used[20], the size of Q-table is still large. The goal of our work is to perform dynamic resource scaling of applications with fluctuated workload. Hence we have to increase the granularity of resource allocation. In order to reduce state-space complexity and improve learning time we propose to split the CPU and memory controller. In the next section we provide the explanation why the split of CPU and memory controllers is possible.

III. System identification

Designing the formal system model is complicated and time consuming process. It has to be repeated each time when the workload pattern changed or and application is updated. The goal of our identification experiments is to understand how different control knobs affect the application performance. The result we are going to use to design resource scaling controller. For the experiments we created a Xen-based test-bed which consists of multi-tier web application benchmark - RUBiS[15] (PHP version). RUBiS is a free, open source auction site prototype which simulates real users behavior of a popular auction *eBay.com*. The front-end tier is a Apache http web-server(WS), the back-end is MySQL database(DB). Each tier runs on VM with centOS6 on board. Client requests are issued by dedicated group of machines running RUBiS client emulator.

A. CPU usage and performance

We periodically changed the CPU entitlement and evaluate application performance. We use Xen credit scheduler[4] to assign virtual CPU capacity to the VM. CPU allocation cap value varied from 8 to 100. The application was tested under different workloads. We varied clients number from 400 to 1600 by step 400. The CPU allocation experiment is implemented with each tier.

Figure 1 shows the mean response time(MRT) as function of front-end tier CPU entitlement. The graph 2 represents relationship between CPU utilization and MRT. Each data point is an average from 20 samples. The presented results allow us to make an observation that MRT is under smooth control of CPU entitlement when CPU utilization above 80%. Moving below the threshold takes the response time out of CPU entitlement control.

The change of CPU entitlement affects also memory. It is shown on figure 3. Memory utilization increases with increase of CPU utilization. The reason is that higher CPU utilization leads to increase of MRT. Therefore incoming requests instead of being served immediately go to the application queue, which requires additional memory space.

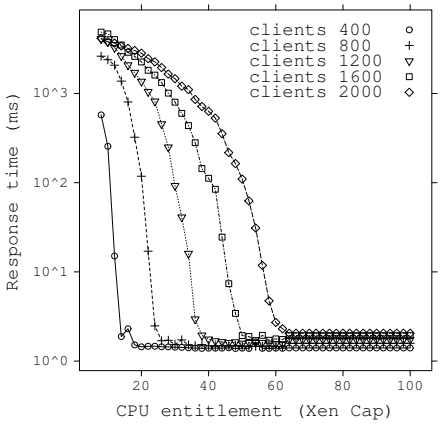

Fig. 1. MRT vs CPU entitlement

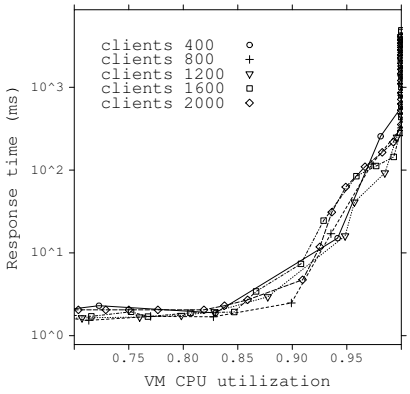

Fig. 2. MRT vs CPU utilization

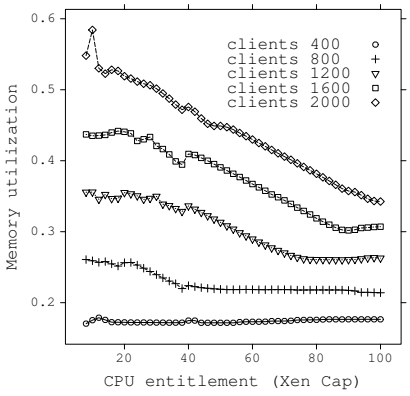

Fig. 3. Memory utilization vs CPU entitlement

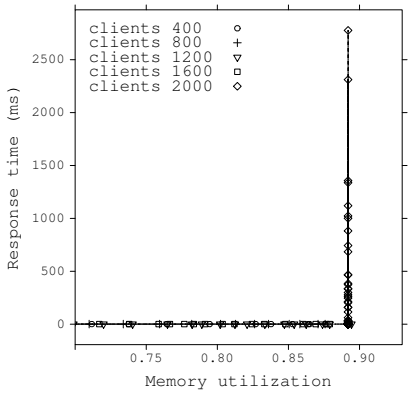

Fig. 4. MRT vs memory utilization

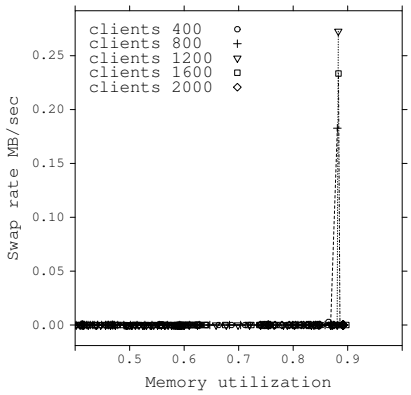

Fig. 5. Swap rate vs memory utilization

B. Memory usage and performance

In the second group of experiments we evaluated the effect of VM memory capacity on the application performance. The memory allocation varied from 240 MB to 896 MB. The experiment was repeated for different workload intensity of 400, 800, 1200, 1600 clients.

Figure 4 presents the relationship between response time and memory utilization. One can notice that there is no possibility to perform smooth MRT control. MRT sharply increases when memory utilization reaches 90% level. The reason is swapping activity which is shown on figure 5. During swapping process OS tries to free memory by saving pages to a disk. The speed of the disk order of magnitude slower than memory, therefore the application performance is affected dramatically.

C. Cluster wide correlation

Resource provisioning of multi-tier applications is a challenging task for auto-scaling techniques. It is important to understand how the change of resource allocation on one of tiers affects the resource consumption of another tier. On figure 6 presented the correlation between DB CPU entitlement and

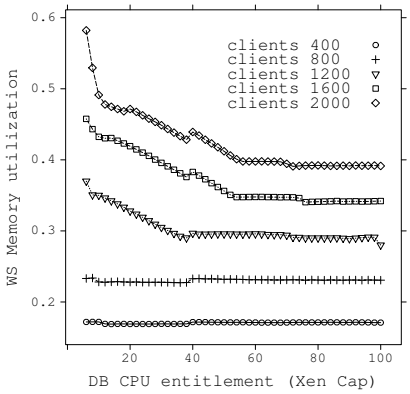

Fig. 6. Effect DB CPU entitlement on WS memory usage

WS memory utilization. The graph shows that the memory usage increases when the CPU entitlement is reduced. The reason is that highly utilized CPU of the DB tier requires more time to serve a request. Hence WS stores incoming requests in the queue instead of sending them down to the DB tier. This is one of the examples of cluster wide correlation. Our goal is to learn scaling policy online therefore we need to ensure that during learning phase the bottleneck does not shift from one tier to another. To avoid described scenario we can use simple approach: increase allocation of all tiers that close to saturation.

The following can be concluded as result of the experiments. CPU is right control knob for fine granular response time regulation. The response time is possible to regulate if CPU utilization above 80%. The change of the CPU entitlement below the value will not take an effect on MRT. In comparison with CPU the change of memory entitlement does not allow to smoothly regulate MRT. Moreover, it brings instability when the memory utilization goes above 90% level. The reason is swapping activity, which requires interaction with disk. As result it dramatically degrades response time. Hence it is important to keep memory utilization below the value. The objective of our work is to keep the application MRT below the value specified in SLA. Hence, we can eliminate memory from MRT control. However we have keep memory utilization below 90% level.

IV. VScaler design

A. Overview

We designed and implemented online resource scaling framework for multi-ter application. The framework does not require a priory model of the application running inside a VM. With the help of reinforcement learning approach VscalerLight learns scaling policy online on a production system. Our framework has predictive and reactive mechanisms. Reactive mechanism allows to quickly scale up the VM resource assignment in response to unexpected load increase and provide desired performance.

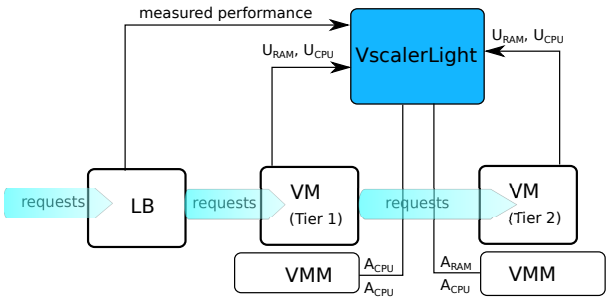

Fig. 7. VscalerLight implementation

Our scaling framework runs on top of Xen hypervisor. It consists of five main components: monitor, predictor, CPU and memory controllers, capacity manager. The monitor collects the application performance and resource usage statistics. VscalerLight has dedicated CPU and memory controllers for each VM. Controllers contain RL models and output the resource allocation scheme. The capacity manager performs resource allocations by communicating with underlying hypervisor. Predictor tracks incoming request rate and issues the value of the workload for the next reconfiguration interval.

On figure 7 presented the implementation of VscalerLight. The resource management is organized in a following way. The monitor periodically tracks the application performance and resource consumption metrics U_{CPU}, U_{RAM}. The load balancer(LB) presented on the picture provides the monitor with performance metrics. Resource consumption is collected from the hypervisor. The predictor forecasts the request rate value for next reconfiguration interval. Each resource controller takes predicted value and outputs resources entitlement values A_{CPU}, A_{RAM} for each tier. Then capacity manager performs VMs reconfiguration. After the fixed time interval the summary statistics are collected and the models of CPU and memory controllers are updated. If performance of the application violates the value specified in SLA, then VscalerLight immediately recalculates resource allocation values and reconfigures VMs.

VscalerLight prediction module uses auto-regressive model to perform a prediction. The prediction value for the next reconfiguration interval is calculated based on previous 100 samples. VscalerLight has also an option to react quickly before the end of the reconfiguration interval if the unpredicted load occurs and response time goes above the value written in SLA. In this case VscalerLight takes the current request rate value and asks CPU and memory modules for appropriate resources entitlement. If the entitlement found then VscalerLight performs resource allocation. If the 'right' resource entitlement value is not found, then VscalerLight shifts to the exploration phase.

B. MDP design solutions

Figures in the section III show that resource consumption has a positive correlation with incoming workload. Both CPU and memory usage increases with the increase of the load. Therefore we include workload dynamics component to the state space description of each resource controller.

1) CPU controller: We define MDP which models our approach to VM CPU allocation problem as $S = \{(c, w)|0 \leq c \leq C_{max}\}$, where:

- $c \in \mathbb{N}$ is CPU allocated to the VM; this value is expressed in Xen credit scheduler cap value. $C_{max} = 100$
- $w \in \mathbb{N}$ is observed request rate which was served without violating SLA. This value can change over time. Initially it is set to 0.

2) Memory controller: We define memory controller state-space as following $S = \{(m, w)|M_{min} \leq m \leq M_{max}\}$, where:

- $m \in \mathbb{N}$ is memory allocated to the VM. $M_{min} = 256$ MB and $M_{max} = 1536$ MB.
- $w \in \mathbb{N}$ is observed request rate which was served without violating SLA. This value can change over time. Initially it is set to 0.

The action set for CPU and memory controllers is defined as following $A = \{a \in \mathbb{Z}|A_{min} \leq a \leq A_{max}\}$. The actions is bounded between $A_{min} = -50\%$ and $A_{max} = 50\%$. It means that the entitlement value c or m can change within these bounds. The following idea lies behind the our actions space definition. We think that is more convenient to change the VM resource capacity by certain percentage rather than add or remove fixed capacity value.

Reward function facilitates the conversion of SLA to VM resource assignment. We use application performance feedback and VM resources usage statistics to calculate the reward. It designed in such a way that it helps to guide the RL agent towards the state which gives higher utility:

$$reward = \frac{perfFeedback}{resUtil} \tag{2}$$

In section III we found that CPU provides smooth response time control. Hence we defined *perfFeedback* as following:

$$perfFeedback = \begin{cases} 1, & \text{if respTime<SLA} \\ e^{-p|\frac{respTime-SLA}{SLA}|} - 1, & \text{otherwise} \end{cases} \tag{3}$$

The agent gets negative reward if SLA violation happens, otherwise the reward value depends on CPU utilization.

Our analysis in section III states performance of the application degrades when the memory utilization goes above certain threshold $hUtil$, therefore memory controller reward depends on the value.

$$perfFeedback = \begin{cases} 1, & \text{if memUtil<hUtil} \\ e^{-p|\frac{memUtil-hUtil}{hUtil}|} - 1, & \text{otherwise} \end{cases} \tag{4}$$

We use default value of $hUtil = 90\%$. However, the exact value of $hUtil$ we can obtain as soon as reached memory utilization triggers the swapping process. The resource utilization feedback of each controller defined as following.

```
repeat
    s_t ← getCurrentState()
    a_t ← chooseNextAction(s_t, Q)
    U_{t+1} ← getResourceUsage()
    r_{t+1} ← calculateReward(U_{t+1})
    w_{t+1} ← getObservedRequests()
    updateRequests(s_{t+1}, w_{t+1})
    updateModel(s_t, a_t, r_{t+1}, w_{t+1} Q)
    t ← t + 1
until Agent is terminated
```

Fig. 8. Agent learning algorithm

```
predValue ← predictWorkload()
for each state s_next connected to s_t do
    g ←getRequests(s_next)
    if g > predValue then
        selectedStates.append(s_next)
    end if
end for
return getBestAction(s_t, selectedStates)
```

Fig. 9. Choose next action

$$resUtil = \frac{(1 - U_r)}{n} \tag{5}$$

where, U_r is observed resource utilization over the last reconfiguration interval.

C. Initializing Q-learning

Q-learning is a model free RL technique. The scaling policy is learned by taking actions and observing a system feedback. So it is important to take wise actions during the model exploration phase. Otherwise the system performance will be affected. Hence, we apply knowledge based exploration. The idea is simply keep resource utilization within desired bounds. For CPU and memory these bounds are 50% and 80%. During the exploration phase the agent takes actions which keep the resource utilization inside the interval. The initialization runs until the predictor collects enough data to perform prediction.

D. Model learning and exploitation

Initially the w parameter of the state description of each controller set to 0. This value changes during the exploration phase. The main loop of Q-learning algorithm is presented on figure 8. The algorithm is the same for both CPU and memory controllers. Each reconfiguration interval the agent chooses an action from the policy learned so far and after fixed interval of time collects resource usage and the application performance statistics. Then it updates states which satisfy observed load and finally updates the policy.

1) CPU controller state update: After the action a_t has been taken w is updated by w_{t+1} for each state which CPU entitlement value c provides lower or equal CPU utilization than observed utilization U_{t+1} . This update rule guarantees that all updated states can serve request rate w with the same MRT. See figure 2.

2) Memory controller state update: The memory state request rate value w is updated by w_{t+1} when the state memory allocation value m higher than observed memory consumption.

The exploitation phase algorithm is presented in figure 9. The agent takes workload prediction value and selects states which have w higher than predicted value. Then it chooses the action which has highest Q-value.

E. Convergence speedup

Q-learning progressively learns environment and updates Q-function for each visited state. It means that only one state-action pair is updated. Even if the state-action space comparatively small it takes long time to learn. Therefore we use approximation technique proposed in [5]. It is a value iteration which uses summary statistics from all observations. We use the technique to update all state-action pairs at each iteration. It allows us to quickly learn optimal policy. The update overhead is comparatively low. The calculation of each resource model on average takes about 20 ms.

V. Evaluation

To evaluate VscalerLight we build the test bed presented in section III. The testbed for our experiments is hosted on quad-core Xeon 2.66GHz with 16 GB memory, 100 Mbps network and Ubuntu 12.04 running on top of Xen 4.1. VscalerLight is evaluated against real world scenario. We apply workload trace from the World Cup 98[9]. It consists of HTTP requests made to the 1998 World Cup Web site. For our experiments we use 6 hour trace starting at 1998-05-10:03:00. The trace has high fluctuations. The ratio between min and peak loads is 12. The application is assumed to require an SLA where the response time is no greater than 20 milliseconds.

In our evaluation we compare VscalerLight against widely used threshold scaling and peak load allocation. The most used thresholds are $ThDown = 30\%$ and $ThUp = 60\%$. These thresholds obviously lead to resource under-utilization. Therefore we experimented with *Threshold policy 1*: 50% - 80% and *Threshold policy 2*: 70% - 80% to improve resource usage efficiency. *Threshold policy 1* changes CPU entitlement by value of 5 and memory entitlement by value of 16 MB. *Threshold policy 2* has values 2 and 8 MB accordingly. We choose different allocation step sizes and thresholds to show that it is difficult to achieve resource usage efficiency and meet SLA at the same time. Moreover, in order to improve the threshold based scaling we added also SLA feedback. It means that reconfiguration is triggered when the application performance violates SLA.

In figure 10 we show the achieved response time in real world scenario. The threshold policy 1 keeps MRT below SLA bound. VscalerLight has a few SLA violations for unseen load spikes. However, the reactive component of VscalerLight quickly brings MRT to desired level. Moreover, it learns and later it does not degrade the application performance when the same load spike appears, while the threshold policy 2

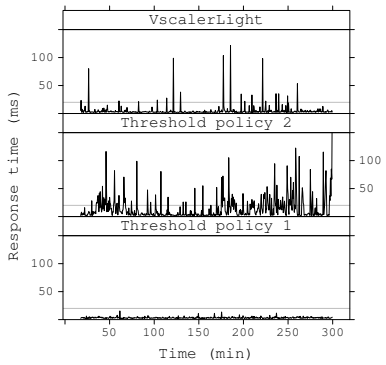

Fig. 10. Response time

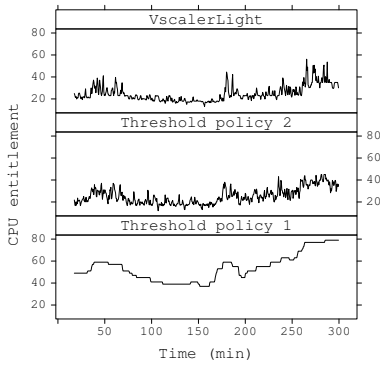

Fig. 11. CPU allocation

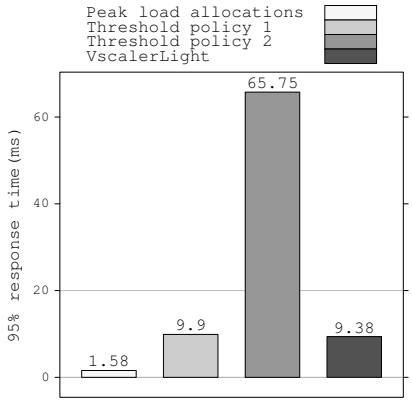

Fig. 12. MRT cumulative distribution

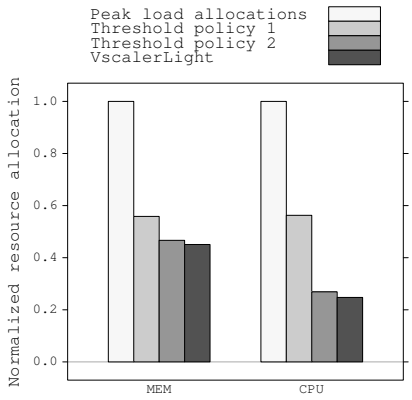

Fig. 13. Normalized resource usage

repeatedly violates SLA for these spikes. To better understand the quality of control of the presented scaling mechanisms we plot the actual CPU allocation of the web server VM on figure 11. All presented threshold policies quickly react to workload increase, because we added SLA feedback. However the policy 1 slowly react to the load decrease, which leads to resource wastage. The policy 2 reacts quickly, but wrongly chosen $ThDown = 70\%$ leads to SLA violation again, so the system out of the steady state. Due to lack of the space we omit graph with actual memory allocation.

In order to compare the efficiency of resource allocation of evaluated policies we calculated total amount of resources allocated to WS and DB VMs. The results are presented on figure 13. The peak load allocation has highest resource consumption among the presented policies. The threshold policy 1 allocates more than the rest dynamic policies. The difference is higher for CPU. The reason is that CPU consumption more volatile than memory. The most efficient scaling mechanism provides VscalerLight.

Figure 12 shows the application performance under control of tested policies. The peak load allocation provides the lowest response time. However, VscalerLight and threshold policy 1 also show good MRT, below SLA requirement. The threshold policy 2 violates SLA.

VI. Related work

Cloud resource management has been widely studied in the past. There is a large body of works which use various techniques to perform dynamic resource allocation for multi-tier applications.

Industrial companies such as RightScale [14], Amazon [1] provide auto-scaling support for the user applications. However they typically require the user to define thresholds to trigger scaling actions. Moreover, threshold based approaches have to be tuned to specific demand pattern and workload. In contrast VscalerLight does not require specific threshold and learn scaling policy online.

Padala et al. [11] apply control theory to solve the problem we addressed in this paper. The approach requires offline controlled experiments to design a system model. The data from existing production systems is hard to use for modeling, because it is often lacks the relevant correlation information.

Our work is close with papers from Rao et al. [12] in terms of approach taken. Authors also address the problem of multi-tier application resource allocation. They perform online system behavior exploration using Q-learning technique. But the state-space size of proposed approach is fairly

large. In order to improve learning time authors use two approaches. First, they reduce the number of states, which leads to worse granularity of resource allocation. Second, neural network(NN) approximation is applied instead of use lookup table. However, NN requires additional computational overhead. In our work we reduce the state space by splitting control knobs models.

Urgaonkar et al. [18] use the classical queuing theory to perform dynamic resource provisioning. The approach works well for the systems with stationary nature. However real applications running the cloud face changes such version updates or workload dynamics. In the presented paper authors addressed the problem of scaling multi-tier application using queuing network. The provisioning of the application is performed based on the peak load. It is clear that provisioning for peak load lead to high resource under-utilization. As result user overpays for computational resources.

Shen et al. [16] proposed a framework which analyzes repeating patterns in resource usage traces to automatically scale application resources. The framework performs CPU and memory scaling, but it only considers the scaling of single VM scaling. Another limitations are the large number of parameters in pattern match algorithm which affects the performance and the time required to collect history trace.

VII. Conclusion

In this work we presented a novel dynamic resource allocation controller. The controller uses reinforcement learning approach to learn scaling policy. The main problem of RL is the state-space complexity. In order to overcome the problem we propose to split CPU and memory models without loosing the quality of control. As result the state-space complexity is reduced from NxM to $Max(N, M)$. Our experiments show that VscalerLight successfully performs dynamic resource allocation for multi-tier application.

Current version of VscalerLight supports only vertical scaling. For the future we plan to extend our work. In particular we want combine vertical and horizontal scaling.

Acknowledgment

This research was funded as part of the ParaDIME project supported by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 318693.

References

[1] Amazon auto scaling service. http://aws.amazon.com/autoscaling/.

[2] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, p. n/a, May 2012.

[3] Cloudsigma, iaas provider. http://www.cloudsigma.com/.

[4] Credit-based cpu scheduler. http://wiki.xen.org.

[5] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck, "Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow," in *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, Venice/Mestre, Italy, May 2011, pp. 67–74.

[6] S. Dutta, S. Gera, A. Verma, and B. Viswanathan, "Smartscale: Automatic application scaling in enterprise clouds," in *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, ser. CLOUD '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 221–228.

[7] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *NOMS*, 2012, pp. 1327–1334.

[8] J. Heo, X. Zhu, P. Padala, and Z. Wang, "Memory overbooking and dynamic control of xen virtual machines in consolidated environments," in *Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, ser. IM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 630–637.

[9] The ircache project. http://www.ircache.net/.

[10] 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 *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 289–302.

[11] 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 *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 13–26.

[12] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: a reinforcement learning approach to virtual machines auto-configuration," in *Proceedings of the 6th international conference on Autonomic computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 137–146.

[13] J. Rao, Y. Wei, J. Gong, and C.-Z. Xu, "Dynaqos: model-free self-tuning fuzzy control of virtualized resources for qos provisioning," in *Proceedings of the Nineteenth International Workshop on Quality of Service*, ser. IWQoS '11. Piscataway, NJ, USA: IEEE Press, 2011, pp. 31:1–31:9.

[14] Rightscale web site. http://www.rightscale.com.

[15] Rubis online auction system. http://rubis.ow2.org/.

[16] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, ser. SOCC '11. New York, NY, USA: ACM, 2011, pp. 5:1–5:14.

[17] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Cluster Computing*, vol. 10, no. 3, pp. 287–299, Sep. 2007.

[18] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1:1–1:39, Mar. 2008.

[19] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 423–436.

[20] L. Yazdanov and C. Fetzer, "Vscaler: Autonomic virtual machine scaling," in *IEEE CLOUD*. IEEE, 2013, pp. 212–219.

[21] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner, "Justrunit: experiment-based management of virtualized data centers," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 18–18.