2016 15th International Symposium on Parallel and Distributed Computing

HybridScaler: Handling Bursting Workload for Multi-tier Web Applications in Cloud

Song Wu, Binji Li, Xinhou Wang*, Hai Jin*
Services Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Emails: {wusong, libinji, xwang, hjin}@hust.edu.cn

***Abstract*—Cloud elasticity allows users to dynamically allocate resources for their applications to adapt with the fluctuant demand. But allocating right amount of resources at right time to handle the bursting workload is still challenging. Most practical auto scaling approaches allocate resources in *horizontal* or *vertical* manner while the *horizontal scaling* usually causes considerable overhead and extra cost for short-term bursting workload. *Vertical scaling* is lightweight and timely but lacks scalability and capacity guarantee in public cloud. In order to find an accurate and cost-effective auto scaling method, we propose a hybrid auto scaling solution called *HybridScaler* which combines long-term predictive *horizontal scaling* and timely reactive *vertical scaling* properly. Specially, our method is based on a *resource-pressure* model which can provide suitable amount of resources matching the changing workload. We implement our prototype in an OpenStack private cloud. We evaluate the effectiveness and efficiency of *HybridScaler* by comparing with mainstream auto scaling methods. *HybridScaler* decreases 16-39% average response time and 34-50% SLO violation rate than both static threshold-based scheme and prediction-based scheme. Meanwhile, it uses less *instance-hours* than static threshold-based scaling method and keeps CPU utilization almost between 60% and 70% which can avoid significant resource waste and SLO violation in the other methods.**

***Keywords*—Cloud application; Auto scaling; Elasticity; Bursting workload**

I. Introduction

Nowadays, cloud computing has gained its popularity due to the *elastic* nature: cloud users can dynamically acquire and release resources in the form of *virtual machines* (VMs) on demand. Unfortunately, allocating proper resources for dynamically-changing workload to meet the *service level objectives* (SLOs) is still challenging for existing clouds [1]. Even though cloud providers have offered threshold-based mechanisms (e.g., Auto Scaling in Amazon EC2 [2]) to help users automatically scale out. But these mechanisms are reactive and *horizontal scaling* [3]: i.e. adding new server replicas to distribute load after SLO violation has been detected. These reactive mechanisms will cause long reactive lag time which includes the detecting period of SLO violation and the overhead of VM booting.

In order to solve the disadvantages of reactive mechanisms, some researches [1]-[4] have proposed proactive *horizontal scalings*. In these works, resource allocations are made based on the prediction of resource utilization or workload request rate before SLO violation happens. These proactive *horizontal scaling* schemes can solve the long lag time existed in reactive mechanisms, but it still has two problems when the workload is bursting.

First, minutely prediction for short-term bursting workload always has low accuracy [4]. So it would be meaningless to change the number of VMs if the prediction is inaccurate. Second, even if we can accurately predict the burst, acquiring/releasing VMs too often still causes significant overhead and cost. Fortunately, *vertical scaling* [5]-[6], a lightweight and hot-plugged resource configuration at runtime within one second, can solve the problem existed in the *horizontal scaling* schemes. But scalability is its weakness while considering the contention and interference in real cloud environment. As a result, an effective auto scaling approach is still needed to handle wide-range and bursting workload.

It is a non-trivial task to find an accurate and cost-effective auto scaling approach [7]. We highlight the problems in existing approaches that need to be solved for bursting workload. Short-term bursting workload is hard to predict because of the unpredictable part in the workload, which makes *horizontal scaling* not suitable. It is more likely to cause significant acquisition overhead and running cost than *vertical scaling*. The cost problem will be enlarged if we take consideration of hourly pricing model [8]. *Vertical scaling* can respond within one second, however, capacity limitation is a barrier for it to deal with wide-range of workload.

In this paper, we propose a hybrid auto scaling algorithm to handle multi-tier cloud web applications with bursting workload pattern (e.g. *Cyclic/Bursting* [3]). The algorithm takes advantages of both *horizontal scaling* and *vertical scaling*. Our goal is to minimize allocated resources while keeping the SLO violation rate under a pre-defined threshold. Considering the predictability in workload trend, our algorithm firstly uses an hourly workload prediction [9], [10] to capture the cyclic feature and long-term trend. We use *horizontal scaling* to change the number of VMs to match the predicted average request rate in the next hour. It makes two benefits: hourly adjustment in VMs causes no waste of *instance-hours*. Meanwhile it provides considerable capacity for the next hour. If workload bursts within one hour, our algorithm employs lightweight *vertical scaling* that hot adds VCPUs and RAM to handling unpredicted bursting overload.

Specially, many *vertical* and *horizontal scaling* methods are

*The corresponding authors are Hai Jin and Xinhou Wang.

978-1-5090-4152-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ISPDC.2016.26

based on resource utilization, such as "Add 2 small instances when the average CPU is above 70% for more than 5 minutes." However, it is not always straightforward for users to select the right scaling indicators and thresholds, especially when the application models are complex. In other words, these trigger mechanisms do not really solve the *performance-resource* mapping problem, and sometimes the resource utilization indicators are not expressive enough to allocate the right amount of resources. Inspired by [10], we construct a theoretical and updatable *resource-pressure* model for multi-tier application to determine how many resources needed for a given workload state.

The main contributions of this paper are summarized as follows:

- **A *resource-pressure* model.** We construct a *resource-pressure* model to provide the right amount of resources to allocate. This model records theoretical or updated tolerable request rate for specified resource configuration. So if the request rate is given, then the model gives appropriate resources (VMs or unit resources) allocation.
- **A cost-effective hybrid auto scaling algorithm.** We propose a cost-effective hybrid auto scaling algorithm for the bursting workload. Our hybrid algorithm takes advantages of both *horizontal scaling* and *vertical scaling*. *Horizontal scaling* is based on hourly workload prediction to adjust the number of instances. It results in less *instance-hours* waste than booting VMs at any time. To capture the anomaly peak or valley load, we use the lightweight *vertical scaling* which hot adds/removes VCPUs and RAM timely.
- **A system prototype, *HybridScaler*.** We implement our scaling system prototype *HybridScaler* based on OpenStack. Nova is patched to support elastic instance type and interference-aware *vertical scaling*. In *HybridScaler*, long-term prediction-based *horizontal scaling* provides a basic computing capability for next hour. Furthermore, *vertical scaling* is responsible for the unpredicted bursting load.
- **Less SLO violation with short response time.** We evaluate our *HybridScaler* by using the three-tier RUBiS benchmark and the ClarkNet workload trace [11]. We demonstrate the effectiveness and efficiency of *HybridScaler* without affecting the SLOs. *HybridScaler* decreases 16-39% average response time and 34-50% average SLO violation rate than other auto scaling schemes while using fewer resources.

The rest of the paper is organized as follows: section II illustrates the problems of existing auto scaling methods to handle bursting workload and presents our motivation. We present the *resource-pressure* model and our hybrid auto scaling algorithm in section III. In section IV, we show the design and implementation of *HybridScaler*. Section V provides the evaluation of *HybridScaler*. Finally we introduce the related work in section VI and conclude this paper in section VII.

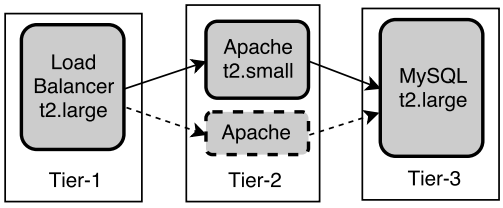

Fig. 1: A common deployment of multi-tier web applications

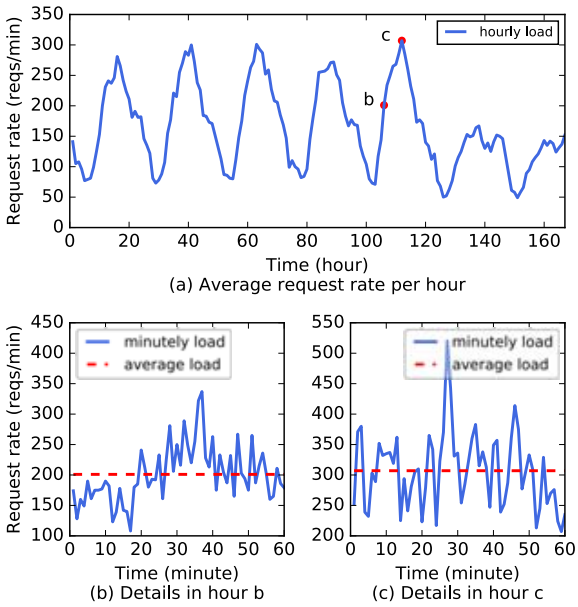

Fig. 2: The variation of the real-world ClarkNet workload

II. Motivation

In this section, we first introduce a common multi-tier architecture of web applications and then discuss the problems existed in traditional auto scaling algorithms when handling bursting workload for multi-tier applications. At last, we present the motivation of our auto scaling method.

A. Application and Workload

Multi-tier architecture is commonly used to build today's Internet applications [10]. As shown in Figure 1, a three-tier e-commerce website represents a large amount of applications in cloud. All requests first enter the *Tier-1* which provides static HTML pages and forwards the dynamic requests to backend nodes. The *Tier-2* runs application's logic and business module. If there are database queries, requests will further go into the *Tier-3*. We can easily capture all access logs and statistic end-to-end response time at *Tier-1*. Internet servers always face bursting workload [3] just as the real trace of ClarkNet [11] shown in Figure 2 (a). There exists some coarse-grained predictable trends. While in one hour, the peak of the request rate may be twice as much as the average load, as shown in Figure 2 (b) (c), which is very challenging to predict [4].

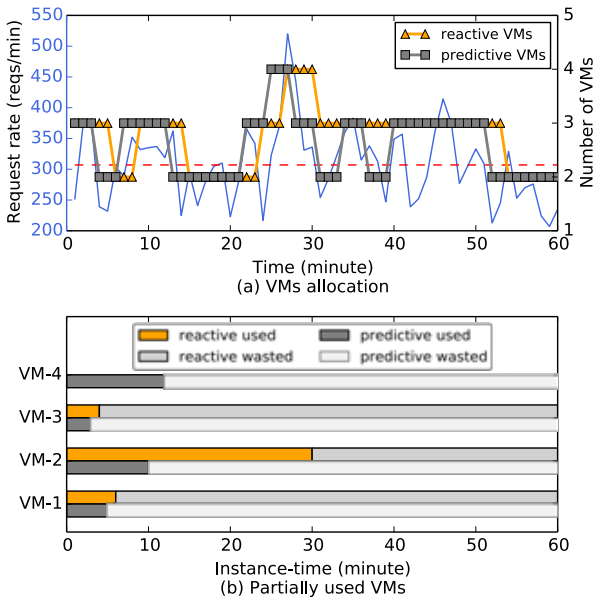

Fig. 3: VM allocations of reactive and predictive *horizontal scaling* to handle minutely bursting workload during one hour

B. Problems in Traditional Auto Scaling Methods

We use a traditional threshold-based scaling and an ideal workload prediction-based scaling to keep up with the fluctuant demand during a specific hour. These two methods implement dynamic resource allocation by starting or stoping instances. For the ideal workload prediction-based scaling, we assume that its minutely prediction is 100% accurate. We plot the results in Figure 3 (a). Obviously, the threshold-based approach has a reactive latency which leads to mismatch with the unstable demands. The prediction-based approach assumes an accurate workload prediction, so it can satisfy bursting workload. But in reality, ideal and accurate short-term prediction is unrealistic because of unexpected bursts. Taking the acquisition overhead and extra cost into consideration, *horizontal scaling* is not suitable for frequent short-term (2 minutes in [1]) reallocation.

Suppose the pricing policy for on-demand instance is hourly charging just the same as Amazon EC2, then the above variations of instance amount will cause much *instance-time* waste and extra cost. We show the running time and wasting time of VMs during scaling period in Figure 3 (b). There are 4 VMs only running for a few minutes in the prediction-based *horizontal scaling*. Even though we adopt a lazy policy that keeps one VM running until its *instance-hour* ends, it will bring out a confliction between stopping time and unexpected next booting time.

C. Motivation

In order to handle the auto scaling problem existed in cloud web applications with bursting workload, we need to make the best use of scalable *horizontal scaling* and lightweight *vertical scaling*. Our goal is to design a cost-effective auto scaling method which can meet the SLOs. We first use an hourly workload prediction to capture the cyclic feature and long-term trend, then use the *horizontal scaling* to change the number of VMs matching the predicted average request rate. It makes two benefits: hourly adjustment in VM number causes no waste of *instance-hours*. Meanwhile it provides a basic capacity for the next hour. If bursting workload comes during the calm down time of the *horizontal scaling*, our algorithm employs lightweight *vertical scaling* that hot plugs VCPUs and RAM to handle unpredicted bursting overload.

III. Resource-Pressure Model and Hybrid Auto Scaling Algorithm

In this section, we provide the methodology of *resource-pressure* model construction to determine how many resources need to be changed for a given workload state. Then we describe the main algorithm of auto scaling to schedule *horizontal scaling* and *vertical scaling*.

A. Resource-Pressure Model

Inspired by previous work [10], we construct a theoretical and updatable *resource-pressure* model for multi-tier applications to determine proper resource allocation for a given workload state.

We use a set LS to express a *m*-tier application:

$$LS = \{L_1, L_2, ..., L_i, ..., L_m\}$$

For each tier L_i in LS, the current resource allocation R_i is:

$$R_i = \{I_{vcpu}, I_{num}, A_{num}\}$$

Where I_{vcpu} is the initial number of VCPUs of the instance type, I_{num} is the number of instances in tier L_i, and A_{num} is the number of VCPUs changed by *vertical scaling* in this layer. For example, $I_{num} = 1, A_{num} = 0$ means 1 instance and no VCPUs adjustment in the tier. We capture the request rate and the response time at each tier using black-box tracing [12]. The end-to-end response time is measured at *Tier-1*, which is used to check if the performance violates the pre-defined SLO. We can easily identify the bottleneck tier by comparing change rate of each tier. For bottle tier L_i, we can get the current workload pressure W_i. At the beginning of any scaling action, we have $I_{num} > 0, A_{num} = 0$, then we can get a simple *VM-Pressure* equation:

$$I_{num} \cdot \alpha_{num} \cdot U_{vm} = W_i$$

Empirically, double resources cannot produce $2\times$ processing capacity but close to $2\times$. So we just use a decay factor α_{num} which is correlated with instance number I_{num}. Then we derive approximately bearable workload pressure (request rate) of per VM when the number of VMs is I_{num}:

$$U_{vm} = \frac{W_i}{I_{num} \cdot \alpha_{num}}$$

TABLE I: Parameters in the Hybrid Auto Scaling Algorithm

Parameter	Description
$RS_{t,i}$	At time period *t*, resource configuration of *Tier-i*, RS_t is resource configuration of all tiers at time t
$WS_{t,i}$	At time period *t*, request rate of *Tier-i*, WS_t is a set of request rates of all tiers
T_{slo}	Predefined performance SLO: tolerable end-to-end response time
$isHScaler$	A boolean variable, if it is True, *horizontal scaling* will be taken
$T_{hscaler}$	Period of *horizontal scaling*, equals to period of workload prediction
$T_{vscaler}$	Period of *vertical scaling*, equals to period of performance detection

Similarly, the tolerable workload pressure of per VCPU U_{vcpu} is:

$$U_{vcpu} = \frac{U_{vm}}{I_{vcpu} \cdot \beta_{num}}$$

Where β_{num} is also a decay factor but for *vertical scaling*. These two factors are initialized to 1. Later, when bottleneck is detected again, these two factor α and β are updating according to current workload state and resource amount.

When scaling is needed, we calculate the workload variation between the nearest two periods $\Delta W_i = W_{i,t+1} - W_{i,t}$ for the bottleneck tier L_i, $\lceil \frac{\delta W}{U_{vm} \cdot \alpha_{num}} \rceil$ is used for *horizontal scaling* ΔI_{num} while $\lceil \frac{\Delta W}{U_{vcpu} \cdot \beta_{num}} \rceil$ for *vertical scaling* ΔA_{num}. For example, we get $\alpha_2 = 0.80$, $\alpha_3 = 0.73$ and $\beta_1 = 0.70$ in our experiment. The decay factor β can be used to update the weight in the *Load Balancer*.

B. Hybrid Auto Scaling Algorithm

To handle the bursting workload, we propose a cost-effective hybrid auto scaling algorithm which can fit both workload trend and burst. First, it predicts average request rate hourly, and adjusts the number of VMs for the next hour. After *horizontal scaling*, if bursting workload exceeds the pressure of current resource pool, our algorithm detects the SLO violation caused by overload then *vertical scaling* is taken to react to the burst. Currently, the SLO violation conditions to trigger *vertical scaling* are described as follows:
$SLOViolationRate > 5\%$, Scale up;
$SLOViolationRate < 1\%$, Check if it is safe to scale down.

In order to understand the algorithm easily, Table I lists the main parameters used throughout the Algorithm 1.

Function $HorizontalAdjusting()$ receives the parameters W_{t+1} and RS_t. For each tier, it gets the workload pressure of current resource amount from the *resource-pressure* model. Then it calculates the workload variation ΔW according to W_{t+1} and the pressure value. If scaling out is needed, it uses ΔW to derive the number of VMs need to be created. It uses $\frac{\Delta W}{2}$ for safe scaling in. So do $VScaleUp()$ and $VScaleDown()$.

Algorithm 1 Hybrid Auto Scaling Algotithm

Input: $RS_t = \{R_{t,1}, R_{t,2}, \ldots, R_{t,m}\}$, T_{slo}, $isHScaler = true$, $T_{vscaler} = 1$, $T_{hscaler} = 60$
Output: RS_{t+1}, updated in each $T_{vscaler}$ period
Begin
//Period of horizontal scaling is $T_{hscaler}$, while $T_{vscaler}$ is for periodically detecting and vertical scaling
$detectRound = T_{hscaler}/T_{vscaler}$
while true do
  $WS_t \Leftarrow GetLastWorkloadState()$
  if $isHScaler$ **then**
    $WS_{t+1} \Leftarrow PredictThscaler(WS_t)$
    $HorizontalAdjusting(WS_{t+1}, RS_t)$
    $detectRound = detectRound - 1$
    $Calmdown(T_{vscaler})$
    //check if it is tolerance to release some vertical resources
    $P_t \Leftarrow GetLastPerformance(T_{slo})$
    $WS_t \Leftarrow GetLastWorkloadState()$
    if $P_t < 1\%$ **then**
      $VScaleDown(WS_t, RS_t)$
      $detectRound = detectRound - 1$
      $Calmdown(T_{vscaler})$
    end if
    $isHScaler = false$
  else
    $P_t \Leftarrow GetLastPerformance(T_{slo})$
    if $P_t < 1\%$ **then**
      $VScaleDown(WS_t, RS_t)$
    else
      if $P_t > 5\%$ **then**
        $VScaleUp(WS_t, RS_t)$
      end if
    end if
    $detectRound = detectRount - 1$
    if $detectRound == 0$ **then**
      $isHScaler = true$
      $detectRound = T_{hscaler}/T_{vscaler}$
    end if
    $Calmdown(T_{vscaler})$
  end if
end while
End

IV. Design and Implementation

We have determined when to use *horizontal scaling* or *vertical scaling* and how many resources are needed in section III. At this point, we provide an overview of our prototype architecture as shown in Figure 4. *HybridScaler* is built on the top of OpenStack cloud platform with the KVM hypervisor, which includes five main components and one agent.

A. Trackerd and SeriesGenerator

Currently, the *Trackerd* traces access logs of the *Load Balancer* and *HTTP Server*. For *Tier-3* which has no access log, it monitors TCP activities of the specific port which has been used similarly in [12]. In general, many third-party software components record their behaviors in log files. Self-defined *Trackerd* can obtain request rate and response

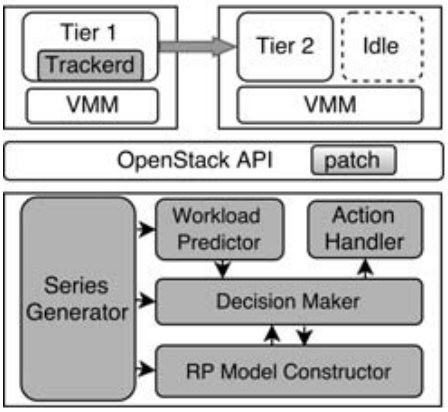

Fig. 4: The Architecture of *HybridScaler* prototype

time from logs as [13] dose. The *SeriesGenerator* continually aggregates request rate and response time minute by minute. It also generates average request rate per $T_{hscaler}$ used by coarse-grained workload prediction.

B. WorkloadPredictor and DecisionMaker

For workload prediction, we use the *Sparse Periodic Auto-Regression* (SPAR) [9] which is a simple time series analysis method:

$$W_t = \sum_{i=1}^{n_0} a_i W_{t-i\cdot T} + \sum_{j=1}^{n_1} b_j \Delta W_{t-j}$$

$$\Delta W_{t-j} = W_{t-j} - \frac{1}{n_0}\sum_{i=1}^{n_0} W_{t-j-i\cdot T}$$

It is feasible to predict the cyclic part in the bursting workload. The *WorkloadPredictor* does periodic prediction over a time period $T_{hscaler}$ that is 1 hour in this paper. The cyclic period T of the workload is 24 hours. The workload state W_t at time t is correlated with $W_{t-1\cdot T}$, W_{t-1} and $W_{t-1-1\cdot T}$ at least. When detecting period or predicting period is changed, the prediction algorithm can be replaced with other algorithms to fit different time grains prediction. Where a_i and b_j in the above equation are obtained through the least squares method. As same as [9], we also set $n_0 = n_1 = 1$. The predicted result is acceptable in our experiment.

According to the auto scaling algorithm in section III, the *DecisionMaker* combines hourly prediction-based *horizontal scaling* and minutely reaction-based *vertical scaling*. It allocates VMs base on the predicted average request rate for the next hour. Then if SLO violation is detected during the hour, the *DecisionMaker* updates the *resource-pressure* model and calls lightweight *vertical scaling* to handle the bursting workload.

C. ActionHandler

Resource allocating action is transferred to OpenStack by the *ActionHnalder* in *HybridScaler*. We patch OpenStack Nova-12.0.0 to enable elastic instance type that two *extra specs* are set in the flavor just as $extra\_specs$: {"$scale\_ram$" : "4096","$scale\_vcpus$" : "2"}. In our experiment, twice as initial capacity for *vertical scaling* is enough. When underlying hypervisor management tool such as Libvirt receives the elastic instance configuration file, it can set *maxvcpus* and *maxMemory* to enable *vertical* hot-plug at VM runtime. To our knowledge, CPU hot-plug requires QEMU version newer than 1.5.0 and Libvirt version newer than 1.1.0. Moreover, QEMU-Guest-Agent is required for CPU hot-unplug.

The drivers of *horizontal scaling* and *vertical scaling* call OpenStack and Libvirt to adjust resources. The *ActionHandler* implements how to start or stop VMs and how to add or remove VCPUs/RAM in a running VM. It directly uses the APIs that Libvirt and Novaclient provide.

V. Evaluation

We evaluate *HybridScaler* in a private cloud environment using an online auction benchmark, RUBiS. We compare our scheme with other auto scaling schemes from different aspects: effectiveness and efficiency. As for effectiveness, the most important performance metrics are response time and SLO violation rate. While for efficiency, we choose resource allocation amount and CPU utilization.

A. Experiment Setup

Our private cloud environment is based on KVM virtualization and OpenStack management platform. Our experiments are conducted with 5 physical nodes. Each node has 2 Xeon E5-2670 CPUs and 16 logical processors in total, 64GB memory and 1Gbps network bandwidth, and runs 64bit RHEL 6.2 OS with QEMU-2.5.0. Each guest VM runs 64bit CentOS 7 initialized with 1 VCPU and 2GB memory.

We choose ClarkNet trace [11] as our workload which has about two weeks access logs. We first obtain the request rate per minute for the first week, then calculate the average request rate per hour as historical time series to train the hourly prediction model. We choose the workload of a specific day as our original experiment workload which contains peak and valley.

For comparison, we also implement a set of alternative auto scaling schemes: *threshold-based horizontal* scaling (TBH-perf for high performance and TBH-save for saving resource), hourly workload *prediction-based horizontal* scaling (PBH). The configurations are shown in Table II. These two threshold-based algorithms use CPU utilization as the performance metric. If the same scaling condition is continuously satisfied for 3 periods (the detecting period is 1 minute), *horizontal scaling* will be taken.

B. Response Time and SLO Violation Rate

We describe the SLO as the end-to-end response time under $0.02s$. So if end-to-end response time exceeds $0.02s$, SLO violation incurs. In experiments, our performance goal is to keep the SLO violation rate under a pre-defined ratio 5%.

TABLE II: Configurations of Auto Scaling Schemes

Schemes	Metric	Parameters
TBH-perf	Average CPU utilization	$(lowerBound, upperBound) = (15, 75), consecutivePeriods = 3$
TBH-save	Average CPU utilization	$(lowerBound, upperBound) = (30, 80), consecutivePeriods = 3$
PBH	Average request rate	$predictionPeriod = 1hour$
Hybrid (*HybridScaler*)	Average request rate SLO violation rate	$predictionPeriod = 1hour$ for proactively *horizontal scaling* $consecutivePeriod = 1$ for reactively *vertical scaling*

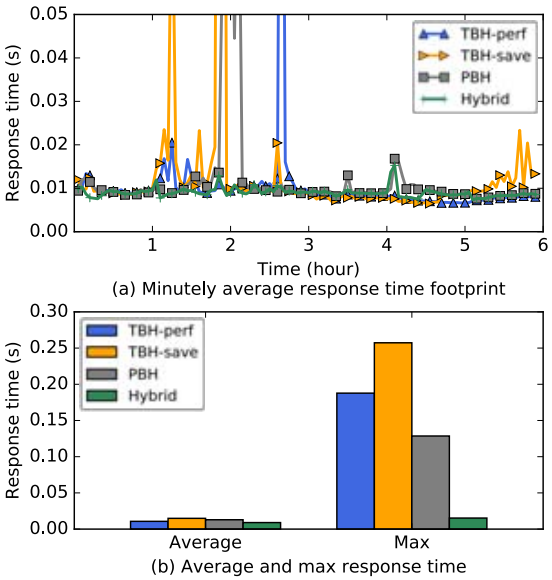

Fig. 5: End-to-end response time of RUBiS *ViewItem* transaction

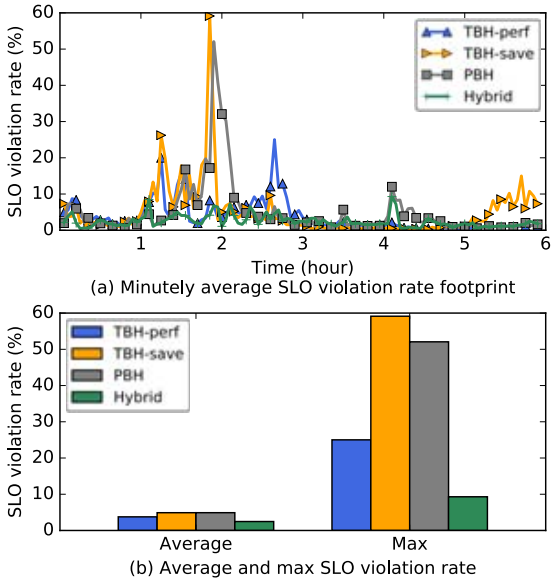

Fig. 6: SLO violation rate. The SLO is described as keeping the end-to-end response time under $0.02s$

Figure 5 shows the end-to-end response time of RUBiS *ViewItem* transaction while using *HybridScaler* and other schemes to dynamically allocate resources. In Figure 5 (a), we can observe that minutely average response time of our scheme is almost stable and only 4 points exceed $0.01s$. It indicates that *HybridScaler* has powerful capability to handle bursting workload. Both TBH schemes (TBH-perf and TBH-save) and PBH schemes have long latency period. The peak response time of TBH-save is the highest. Because it has the longest lag time and insensitive upper bound to handle burst. PBH also has many long-latency points. The reason is that PBH scheme is a long-term proactive resource allocation method. It provides a fixed number of VMs for a long term without any changes in this period until the next period. Our *HybridScaler* is based on PBH and takes fine-grained *vertical scaling* during each long period. We evaluate the average and maximal response time of *HybridScaler* compared to other schemes and plot the results in Figure 5. From the figure we can see, *HybridScaler* reduces the average response time by 16-39% than TBHs and 30% than PBH. While for maximal response time, the reductions are 92-94% and 88%.

Figure 6 (a) plots the SLO violation rate against time for different schemes. The transient SLO violation rate sometimes exceeds 20% when using TBH-perf auto scaling. The value is up to 50% in TBH-save and PBH schemes. But it is always less than 10% in our scheme. Though TBH-perf is a reactive method, it has less SLO violation than PBH. Because PBH allocates 2 VMs to handle continuously growing workload in the 2nd hour. TBH-perf firstly uses 2 VMs and then allocates a new VM when CPU utilization exceeds the upper threshold (75%). *HybridScaler* firstly allocates 2 VMs just as PBH, then adds one or more extra VCPU(s) to meet the excessive demand. We plot the average and maximal SLO violation rate in Figure 6 (b). *HybridScaler* reduces the maximal minutely SLO violation rate by 63-84% than TBHs and 82% than PBH.

C. Resource Allocation and Utilization

With the SLO guaranteed, the other goal of auto scaling is to use minimized resources. Efficient auto scaling method incurs little resource waste and cost. The number of VMs used directly reflects the efficiency. In our experiments, these auto scaling schemes dynamically allocate resources for RUBiS *Tier-2*. The other two tiers do not support *horizontal scaling* so that we only analyze the resource allocation of *Tier-2*.

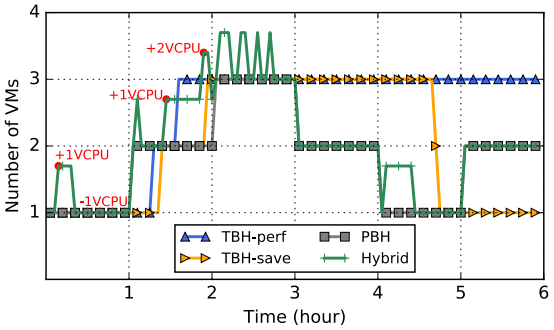

Fig. 7: Resource allocation of RUBiS *Tier-2*. TBHs (TBH-perf and TBH-save) and PBH only use VMs. The VMs allocation of *HybridScaler* is the same as PBH, but it uses extra VCPUs for unpredicted bursts

The area between resource allocation line of each scheme and x-axis in Figure 7 represents the resources that *Tier-2* occupies during the application running. From the figure we can see, TBH-perf consumes the most VM running time while TBH-save uses fewer resources than TBH-perf. PBH and *HybridScaler* use the same *instance-hours*. Both schemes reallocate VMs hourly. But *HybridScaler* needs minutely vertical adjustment such as $+1VCPU$ and $-1VCPU$ marked in Figure 7. It is necessary that using extra VCPUs to handle the temporary bursting workload. Otherwise we have to start a new VM. In total, PBH and *HybridScaler* consume 11 *instance-hours*. TBH-perf consumes 16 *instance-hours* and TBH-save consumes 13 *instance-hours*. Moreover, there are 2 incomplete *instance-hours* in TBH-perf and 4 incomplete *instance-hours* in TBH-save. If the pricing model is hourly pricing, application owner needs to pay full cost for these partial hours.

CPU utilization is another important performance metric for auto scaling schemes. Extreme high utilization will incur SLO violation, and low utilization means resource wasted. In Figure 8 (a), we can observe that the CPU utilization of *HybridScaler* keeps around at 60%. The truth is that our *resource-pressure* model automatically allocates right amount of resources for the specified workload state. It leads to efficient use that CPU utilization is rarely over 70% and under 50%. But both TBHs and PBH have overloaded time. The maximal CPU utilization of them are above 90%. Meanwhile, the lower CPU utilization around the 5th hour means more resource wasted in TBH-perf than the other schemes. TBH-save corrects the mistake by using a bigger lower bound than TBH-perf. Figure 8 (b) shows the distribution of CPU utilization levels. During the running period, the CPU utilization of *HybridScaler* is always between 60% and 70%. The time ratio is close to 60%. The CPU utilization is approximately uniform distribution between 20% and 80% for both TBH-perf and TBH-save. It means that TBHs has the lowest resource utilization. PBH and TBH-save have about 5% time ratio that CPU utilization exceeds 80% while *HybridScaler* has none.

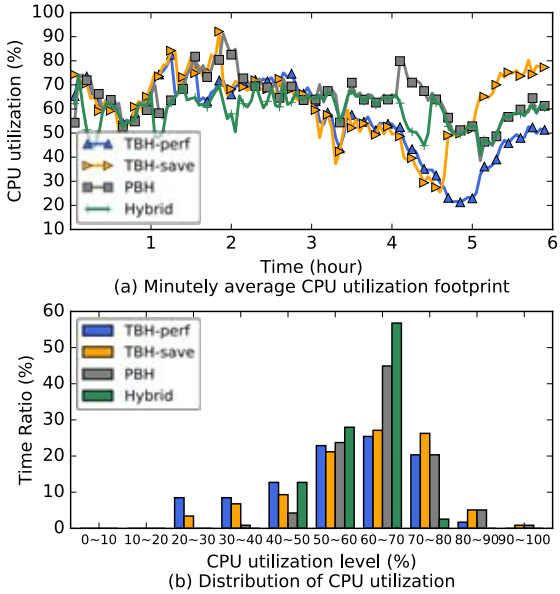

Fig. 8: Average CPU utilization of RUBiS *Tier-2 Apache*

The high utilizations of TBH-save and PBH lead to more SLO violations than *HybridScaler*.

VI. Related Work

To solve the auto scaling problems for multi-tier applications in cloud, we use *request tracing*, *modeling* and *profiling* techniques to make decision. Our hybrid auto scaling algorithm improves previous policies. We make it better to handle bursting workload. So our related works include request tracing and auto scaling in cloud.

A. Trace-based Profiling for Multi-tier Applications

Truly precise black-box *request tracing* approaches are rare. Many previous tracing techniques instrument middlewares or infrastructures to record the applications' traces, meanwhile avoiding the modification of the applications' source codes. Pinpoint [14] locates component faults in J2EE platforms by tagging and propagating a globally unique request ID with each request. For black-box tracing and path discovering, vPath [13] and PreciseTracer [12] have a relative high precision. Unlike intrusive instrumentation, they monitor thread and network activities to discover their causality, without the knowledge of applications' source codes or dependent libraries. Wang et al. [15] observe transient bottlenecks in an n-tier application which significantly cause performance loss by using fine-grained black-box tracing. We take the advantages of these tracing techniques to monitor the requests incoming rate and latency of every components in multi-tier applications as our high-level performance metrics for precise auto scaling.

B. Auto Scaling in Cloud

There are large amount of works which use various techniques to perform dynamic resource allocation for multi-tier web applications, including *schedule-based/threshold-based*, *reactive/predictive*, *vertical/horizontal* methods. Our studies are similar with these approaches which aim to meet the application performance SLOs and minimize the costs of resource rented from the cloud provider. Most cloud companies such as Amazon EC2 provide auto scaling support for users' applications. However they typically use a static *threshold-based* policy, which requires the users to define thresholds to trigger scaling actions. Yazdanov et al. [5] analyze the impact of CPU utilization and entitlement of VMs to response time, then use an online learning approach to predict the request rate and mapped requests load to CPU entitlement reallocation. Dynamic tuning the entitlement of CPU and memory is *vertical scaling*, while Amazon EC2 [2] provides *horizontal scaling* which adds or removes some VMs. Han et al. [16] use *vertical scaling* and *horizontal scaling*, but they only take the advantages of fine-grained resource-level scaling and extensible VM-level scaling. We also combine these two resource allocation grains, but through a better way: *horizontal scaling* provides a computing baseline for long term and *vertical scaling* answers for minute-level bursting demand.

There are some works [1]-[4] similar with ours in which short-term or medium-term CPU utilizations are predicted then correlated with resource-level or VM-level resource allocation. Shen et al. [4] present CloudScale, a system that employs online resource demand prediction and automates fine-grained elastic resource scaling. They extend the idea to adjust the number of VMs [1] instead of only CPU capacity tuning. These approaches do not care about *instance-hour* waste problem caused by frequently acquiring/releasing instances. We make efforts on this issue and use long-term workload prediction for hourly charging.

VII. Conclusion

In this paper, we present *HybridScaler*, a hybrid auto scaling scheme to handle a representative bursting workload pattern for multi-tier applications in cloud. *HybridScaler* firstly uses a prediction-based *horizontal scaling* to fit with the cyclic feature and long-term trend of the workload. Then if the unexpected bursting demand exceeds the resource capability, it employs a lightweight *vertical scaling*. To provide proper amount of resources matching a given demand, we build a *resource-pressure* model using black-box request tracing techniques. *HybridScaler* can provide almost right resource allocation by combining horizontal VMs adjustment and vertical resources hot-plug. We implement *HybridScaler* on top of the OpenStack cloud platform with KVM virtualization. Compared with other mainstream auto scaling methods, *HybridScaler* reduces 16-39% in average response time and 34-50% in SLO violation rate than both static threshold-based schemes and prediction-based schemes. It keeps a stable performance line for the application to avoid unexpected long latency. Meanwhile, *HybridScaler* keeps the CPU utilization almost between 60% and 70% with less redundancy and overload than the others, because it allocates proper VMs and unit resources. *HybridScaler* achieves 31% reduction in *instance-hours* than static threshold-based scheme which is more cost-effective than traditional auto scaling methods when the pricing model is hourly pricing.

Acknowledgements

The research was supported by National Science Foundation of China under grant 61232008, National 863 Hi-Tech Research and Development Program under grant 2015AA01A203 and 2014AA01A302, and Chinese Universities Scientific Fund under grant 2013TS094.

References

[1] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for Infrastructure-as-a-Service," in *Proc. of ICAC*, pp. 69–82, 2013.

[2] Amazon, "EC2 AutoScaling," [Online]. Available: http://aws.amazon.com/autoscaling, 2016.

[3] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proc. of SC*, pp. 49:1–49:12, 2011.

[4] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *Proc. of SOCC*, pp. 5:1–5:14, 2011.

[5] L. Yazdanov and C. Fetzer, "Lightweight automatic resource scaling for multi-tier web applications," in *Proc. of CLOUD*, pp. 466–473, 2014.

[6] R. da Rosa Righi, V. F. Rodrigues, C. A. da Costa, G. Galante, L. C. E. D. Bona, and T. C. Ferreto, "AutoElastic: Automatic resource elasticity for high performance applications in the cloud," *IEEE Transactions on Cloud Computing*, vol. 4, no. 1, pp. 6–19, 2016.

[7] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proc. of ICDCS*, pp. 559–570, 2011.

[8] H. Jin, X. Wang, S. Wu, S. Di, and X. Shi, "Towards optimized fine-grained pricing of IaaS cloud platform," *IEEE Transactions on Cloud Computing*, vol. 23, no. 6, pp. 1159–1167, 2012.

[9] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services." in *Proc. of NSDI*, pp. 337–350, 2008.

[10] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Proc. of ICAC*, pp. 217–228, 2005.

[11] ClarkNet, "ClarkNet-HTTP Trace," [Online]. Available: http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html, 2016.

[12] B. Sang, J. Zhan, G. Lu, H. Wang, D. Xu, L. Wang, Z. Zhang, and Z. Jia, "Precise, scalable, and online request tracing for multitier services of black boxes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 6, pp. 1159–1167, 2012.

[13] B.-C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "vPath: Precise discovery of request processing paths from black-box observations of thread and network activities." in *Proc. of ATC*, pp. 1–14, 2009.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. of DNS*, pp. 595–604, 2002.

[15] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *Proc. of ICDCS*, pp. 31–40, 2013.

[16] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proc. of CCGrid*, pp. 644–651, 2012.