Enhancing Rules For Cloud Resource Provisioning Via Learned Software Performance Models

Mark Grechanik
University of Illinois at Chicago,
Chicago, IL, USA
drmark@uic.edu

Qi Luo, Denys Poshyvanyk
College of William and Mary,
Williamsburg, VA, USA
{qluo,denys}@cs.wm.edu

Adam Porter
University of Maryland
College Park, MD, USA
aporter@cs.umd.edu

ABSTRACT

In *cloud computing*, stakeholders deploy and run their software applications on a sophisticated infrastructure that is owned and managed by third-party providers. The ability of a given cloud infrastructure to effectively re-allocate resources to applications is referred to as *elasticity*. To enable elasticity, programmers study the behavior of applications and write scripts that guide the cloud to provision resources for these applications. This is an imprecise, laborious, manual and expensive approach that drastically increases the cost of application deployment and maintenance in the cloud.

We propose an approach, coined as Provisioning Resources with Experimental SofTware mOdeling (PRESTO), to automatically learn behavioral models of software applications during performance testing in order to recommend programmers how to improve provisioning strategies that guide the cloud to (de)allocate resources to these applications. We applied PRESTO to two software applications and our experiments demonstrate that with PRESTO programmers can create rules for provisioning resources with a high degree of precision when the performance is about to worsen, so that the applications maintain their throughputs at the desired level.

Keywords

Cloud computing; Performance testing; Behavioral models

1. INTRODUCTION

In *cloud computing*, stakeholders deploy their software applications on a sophisticated infrastructure that is owned and managed by third-party providers (e.g., public clouds such as Amazon AWS) or in-house installations. Two fundamental properties of cloud computing include provisioning resources to applications on demand and charging their owners for pay-as-you-go resource usage [3]. The elasticity of cloud refers to its capacity to scale resources based on a real workload. Many cloud providers claim that their cloud infrastructures are *elastic*, i.e., they automatically (de/re)allocate resources, both to *scale out* and *up* – adding resources as demand increases, and to *scale in* and *down* – releasing resources as demand decreases. Using elastic clouds, stakeholders pay only for what they use, when they use it, rather than paying up-front and continuing costs to own and maintain their hardware/software and supporting technical staff [3, 4, 23].

In practice, even the most elastic clouds are not perfectly elastic [13, 3]. Understanding when and how to reallocate resources is a hard problem, since it is generally impossible to quickly and accurately match resources to applications' needs. A recent article underscores this point as it describes its state-of-the-art supervisory system that monitors various black box metrics and then directs the cloud to initiate scaling operations based on that data [10]. As a result, some elasticity-related problems for cloud computing include *under-provisioning* applications so they lack the resources to provide appropriate quality of service, or *over-provisioning* applications so stakeholders end up holding and paying for more resources than they need. Specifically, although elasticity is a fundamental enabler of cost-effective cloud computing, existing *provisioning strategies* (i.e., rules used to (de)allocate resources to applications) are typically obtained in ad-hoc fashion by programmers who study the behavior of the application in the cloud. It is a manual, imprecise, intellectually intensive and laborious effort.

Our novel idea for *Provisioning Resources with Experimental SofTware mOdeling (PRESTO)* enhances cloud elasticity by learning and refining models of software applications through performance testing in the cloud and by using these automatically learned models to help programmers to craft application-specific resource provisioning strategies. That is, PRESTO bridges a pure black-box cloud resource provisioning to software engineering, where behavioral models of the application are re-engineered automatically as part of performance testing, and programmers use these models to create rules for provisioning of resources to these applications in the cloud. This paper makes the following contributions:

- We present a general approach for improving the performance of cloud-deployed applications by using models and artifacts automatically derived from application performance testing.
- We rely on performance models of applications and use these models to guide programmers in developing application- specific *provisioning* strategies to improve cloud elasticity.
- We evaluate PRESTO on two software applications. The results strongly suggest that PRESTO is effective and efficient. We believe that our work is the successful and early attempt at achieving precise cloud elasticity by using software engineering artifacts for guiding resource provisioning in the cloud. All experimental results are available online [28].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICPE'16, March 12–18, 2016, Delft, Netherlands.
© 2016 ACM. ISBN 978-1-4503-4080-9/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2851553.2851568

2. BACKGROUND AND PROBLEM STATEMENT

In this section we provide some background on behavioral models, resource provisioning and basic scaling operators, present our hypothesis, and outline the problem statement.

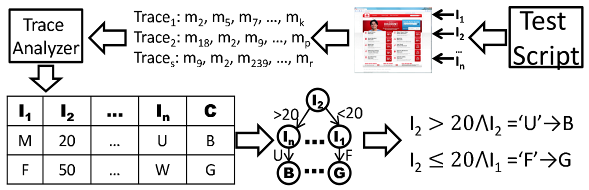

Figure 1: Learning performance rules.

2.1 Obtaining Behavioral Models

A performance model of the Application Under Test (AUT) is learned automatically during performance testing to establish under what loads and the ranges of input values the AUT loses its scalability. Creating application's performance models is fundamental to our approach for determining effective resource allocation strategies and for evaluating their costs and benefits. A recent work, FOREPOST [11], abstracts the AUT as a function that maps properties of the input data (i.e., client requests) to the classes of performance behavior of the AUT. That is, performance tests are generalized as relations (i.e., performance rules) that map input values to AUT performance behaviors.

To obtain performance rules automatically during performance testing, the AUT is executed using a test script with different values and loads for its inputs, $I_1, \ldots, I_n$ as shown in the upper right corner in Figure 1. As a result of executing the AUT, its execution traces are collected with different performance counters (e.g., elapsed execution time, memory consumption). These traces are summarized by the Trace Analyzer into a table whose columns represent input values and numbers of users who use these input values and rows correspond to the collected execution traces. The last column, C, designates the performance class of the execution trace, i.e., **G**ood or **B**ad, a summary measure of the QoS using the throughput associated with a given trace. This table is processed by a machine learning algorithm that computes a decision tree, where nodes represent the AUT inputs and their counts and the leaves represent quality of service (QoS)/throughput classes. Directed edges connect nodes and they are labeled with decisions (e.g., $I_2 > 20$) for taking a specific edge. This decision tree is translated into performance rules, examples of which are shown in Figure 1.

2.2 Resource Provisioning for Cloud Apps

In this paper, we focus on client-server data-centric applications because of their widespread use. Specifically, the front end of the application accepts requests from users (i.e., the input data), the middle tier performs some business logic computations, and the back-end encapsulates data storage. We term the set of related client requests and computations as a work unit or *transaction*. Each client request is the input data for an application, and its performance behavior depends on the type and the size of the input data, among other things. The performance of the application is typically measured as its *throughput*, i.e., the number of transactions per time interval that the application executes. Effective provisioning strategies will allocate resources to maintain a desired level of the throughput for the application.

Depending on the types of resources, their provisioning to applications has different effect on the throughput. Consider scaling up this application where more CPUs are added as new users arrive. Since the application spawns a new thread for each user, adding CPUs and assigning newly spawned threads to these CPUs will increase the throughput of the application. At the same time, adding more RAM may not necessarily help. On the other hand, if the database appears to be a main bottleneck, scaling out this application by starting its new instance or a new instance of its database may increase the throughput by parallelizing transaction processing. Thus, resource provisioning in the cloud is a main technique for maintaining a desired level of the throughput to guarantee some quality of service for applications.

2.3 Scaling Operators of Cloud Computing

In a cloud computing model, software components run inside *virtual machines (VMs)*, which provide emulated physical machines [32]. In general, the cloud allocates different amounts of resources to different VMs and these resources have different costs. For example, if a VM contains more components that involve computationally intensive operations, the cloud scales it up by assigning more CPUs and memory units. Alternatively, the cloud can scale out this application by replicating VMs, thus enabling multiple requests to be processed in parallel by these VMs that run these components. The cloud uses two main scaling operators: $\texttt{sres}(r,a,i)$ and $\texttt{sinst}(a,i)$, where r is the type of a resource (e.g., memory, CPU), a is its quantity, and i is the VM identifier. The scaling operator `sres` (de)allocates the resource, r, in the quantity, a, to the VM, i, and the scaling operator `sinst` (de)allocates a instances of the VM, i. Of course, different costs are associated with these scaling operators. For example, allocating a new VM is more expensive than assigning more CPU or RAM to an already running VM. In theory, elastic clouds "know" when to apply these operators to (de)allocate resources with high precision and efficiency.

2.4 The State of the Art and Practice

Today, stakeholders typically deploy their applications in the cloud, using ad-hoc scripts in which they encode the behavior of these applications and rules on how the cloud should apply the scaling operators based on a coarse collection of performance counters and otherwise "guesstimating" on how to provision resources to their applications in the cloud. The Google Cloud, the Amazon EC2 and Microsoft Azure clouds have issued guidelines for manually load balancing their elastic clouds, directing their users to manually specifying the conditions that trigger work routing and scaling operations. The key takeaway here is that existing cloud providers understand that their users often need to manually configure the cloud. Such manual activities are tedious, error-prone and expensive, and clearly demonstrate that clouds, such as Amazon's EC2, have a long way to go in their quest for better elasticity (see http://aws.amazon.com/autoscaling/).

2.5 The Problem Statement

We address the following problem – how to enable programmers to provision resources with a high degree of precision to specific VMs to maximize the throughput of an application that runs in these VMs. Our goal is to enable stakeholders to create more precise rules for resource provisioning. In this paper we concentrate on the resource consumption by application for specific combinations of input values rather than for different loads. In order to know resource demands for an application, its performance analysis should be done, ideally, with all allowed combinations of values of the inputs. Unfortunately, this is often infeasible because of the enormous possible number of combinations. Thus, a subgoal is to approximate the performance behavior of the application.

We hypothesize that by using application-specific performance models that are re-engineered automatically during software performance testing it is possible to provision resources to specific VMs to maximize the application's throughput. To do that, it is important to know which resources affect more the performance of specific VMs that host software components. Therefore, our goal is to recommend to stakeholders what type of resources and in what quantity should be provisioned for certain types of input data, so that this information can be used to make the cloud highly elastic.

3. APPROACH

In this section, we give an overview of PRESTO and explain provisioning strategies as well as our proposed algorithm.

3.1 Overview of PRESTO

To address the problem of enhancing cloud elasticity, the cloud should provision adequate quantities of specific resources to the designated VMs using rules that are supplied by programmers. Adequate quantities of resources are those that do not lead to under- and over-provisioning. A key idea of our solution, PRESTO, is that stakeholders should create application-specific provisioning strategies using models that are obtained during performance testing in the cloud. PRESTO methodology combines obtaining application's *behavioral model* (i.e., a collection of workload profiles, constraints, performance counters, and various relations among components of the application [21, 26]) with *sensitivity analysis* that parameterizes resources and samples the parameter space to determine the types of resources that have the highest impact on the throughput of the application. Eventually, stakeholders synthesize the results of modeling and sensitivity analysis into provisioning strategies that they import into the cloud to provision resources based on specific client requests that arrive to the cloud for a given application. To summarize, provisioning strategies for the application are obtained during its performance testing. These provisioning strategies concisely describe for what types of inputs and input loads the application loses its scalability and what types of resources and in what quantities should be provisioned to maintain the application's quality of service by not allowing its throughput to fall below some level as dictated by an SLA.

3.2 Obtaining Provisioning Strategies

Using the learned model, the user of PRESTO discovers provisioning strategies that most effectively alleviate decreasing throughputs. The user searches through a space of possible cloud provisioning operations. For example, if a software component involves computationally intensive operations or requires a lot of memory, the cloud could scale-up the VM in which the component runs by giving it more CPU and memory units. If the component's performance has unacceptable latency, resulting from database interactions, then the cloud could scale-out the VM that contains this database. These strategies can be applied to the system and if additional testing shows performance improvements, then a new provisioning strategy is automatically generated.

Definition 1 *A provisioning strategy is a relation $P \Rightarrow (R \bullet R)^*$, where P is a performance rule and $R \bullet R$ is a resource provisioning scheme, where $R \in \{\texttt{sres}, \texttt{sinst}\}$ are resource (de)allocation operators defined in Section 2.3 and $\bullet$ stands for logical connectors* `and` *and* `or` *and $*$ is the Kleene star.*

An example representation of a provisioning strategy for this rule is $(A, \mathbb{R}) \Rightarrow \texttt{sres}(\mathbb{P}, 3, VM_i) \wedge \texttt{sinst}(2, VM_n)$, meaning that if we observe inputs to the application, A, then the rule $\mathbb{R}$ holds, meaning that we will triple the number of CPU units, $\mathbb{P}$, that are assigned to the VM_i (i.e., scale up), and double the number of virtual machines, VM_n that run the AUT (i.e., scale out). If this provisioning strategy is made available to the cloud in advance, then when the application, A is executed with the input values that satisfy the performance rule $\mathbb{R}$, the cloud will provision resources according to the designated provisioning strategy instead of waiting until the performance of the application demonstrably worsens as is done in existing clouds [10]. Conversely, the cloud will deallocate resources to some baseline level if no consequent is present.

3.3 PRESTO Algorithm

PRESTO's algorithm for synthesizing provisioning strategies, $\mathcal{S}$ is shown in Algorithm 1. The input to Algorithm 1 is the AUT and VM configuration that includes all VMs in which the AUT runs as well as resources assigned by the cloud to these VMs.

The algorithm builds the behavioral model using `FOREPOST` in line `2`, outputting function $f_{\mathcal{A}}$ to represent the performance model of the AUT by learning rules to map the groups of inputs to the classes that describe different AUT performance behaviors (see details in [11]). We defined classes for behaviors: good class, where performance of the AUT is scalable and bad class, where the AUT is not scalable. But for nontrivial AUT, the range of performance behaviors is broader, thus there can be more classes. In the outer `for` loop between lines `4-13`, the AUT is checked for each class for different loads if it loses its scalability. If it does, method `GetBottleneckModel` in line `5` returns types of fault models, describing violations of different properties regarding resource use, like CPU load, memory utilization, and database bottlenecks. Theses defaults are likely to cause the AUT to lose its scalability. Essentially, the method determines the consumption of resources and operations in the execution of the AUT that led to this consumption. In the `for` loop between lines `6-12`, for each detected fault model, m, a set of allocated resources, R_m is obtained in line `7`. Then, between lines `8-11`, different types of allocated resources are perturbed by scaling them up or out. All the provisioning strategies, the performance rules and the corresponding AUT's behaviors are added to $\mathcal{S}$ in line `10`, guiding programmers in developing provisioning strategies to improve cloud elasticity.

Algorithm 1 PRESTO Algorithm.

```
 1: Inputs: AUT $\mathcal{A}$, VM Configuration $\Omega$
 2: Behavioral Model:( $\mathcal{A}, \Omega$ ) $\mapsto f_{\mathcal{A}} : I \to \mathcal{C}$
 3: $\mathcal{S} \leftarrow \emptyset$ {Initialize the set of provisioning strategies}
 4: for all $c \in \mathcal{C} \wedge \neg$ Scalable($\mathcal{A}, c$) do
 5:    GetBottleneckModel( $\mathcal{A}, c$ ) $\mapsto \mathcal{M}$
 6:    for all $m \in \mathcal{M}$ do
 7:       GetVMResource( $m$ ) $\mapsto R_m$
 8:       for all $r_m \in R_m$ do
 9:          $\Omega_{\pm\Delta} \leftarrow$ Perturb($\Omega, \Delta_{R_m}$)
10:          $\mathcal{S} \leftarrow \mathcal{S} \cup$ GetRule($f_{\mathcal{A},\Delta}$)
11:       end for
12:    end for
13: end for
14: return $\mathcal{S}$
```

4. EXPERIMENTAL EVALUATION

In this section, we pose research questions (RQs), describe subject AUTs, explain our methodology and variables, formulate hypotheses, and discuss threats to validity.

4.1 Research Questions and Hypotheses

A main goal of our proposed work is to investigate if learned performance models of applications can enable stakeholders to create precise and effective provisioning strategies for applications running in the cloud. To do that, we will pursue and evaluate the following objectives. One objective is to show that the resulting provisioning strategies should be more effective than those strategies produced by existing state-of-the-art automated black-box approaches and manually created ad-hoc provisioning scripts (see Section 2.4). Another equally important objective is to learn these strategies quickly and automatically without placing a significant demand for resources. To better quantify these objectives, we will seek to answer the following research questions.

RQ_1: How effective is PRESTO in maintaining the throughput of the applications in the cloud?

RQ_2: How fast and efficient is PRESTO in learning provisioning strategies?

The rationale for RQ_1 is to determine if PRESTO strategies will enable subject applications to maintain their throughputs at some desired levels. Suppose that the application's throughput drops below some level that is dictated by an SLA for certain combinations of its input values. By applying PRESTO strategies, we expect the cloud to increase the throughput to an acceptable level. We compare how effective a cloud infrastructure using the PRESTO methodology is with respect to a cloud infrastructure without PRESTO that uses a commercial black-box application agnostic autoscaler. We introduce the following null hypothesis to evaluate how close the means are for throughputs for different approaches. We seek to evaluate the hypothesis at a 0.05 level of significance.

H_0 The primary null hypothesis is that there is no difference in throughputs of the subject applications for PRESTO and the competitive approaches.

The rationale behind the H_0 is that with PRESTO-based methodology, elastic resource provisioning will achieve the same application's throughput as the competitive approaches. We expect to reject this hypothesis to confirm our conjecture that the PRESTO-based cloud configuration will enable the cloud to provision resources to subject applications resulting in higher throughputs. The other aspect of RQ_1 is to investigate the economical aspect of autoscaling in the cloud. Recall that different resources have different costs. It is important that PRESTO can give a tradeoff between the improved throughput and its cost. To address RQ_2, we instrument our system to determine the time and resources that PRESTO needs to learn provisioning strategies. In addition, we want to establish how long it takes to converge to stable provisioning strategies.

4.2 Subjects and Cloud Configurations

We evaluate PRESTO on two three-tier Java applications, JPetStore and Dell DVD Store, which are widely used as industry performance benchmarks [30, 15]. JPetstore is a Java implementation of the PetStore benchmark. We used JPetStore 4.0.5 [18], which consists of 36 classes in 8 packages and 382 methods with the average cyclomatic complexity of ≈ 1.23. It is deployed in Tomcat 6 and uses Apache Derby as its backend database. In this paper, we only present the results for JPetStore. The experimental results for Dell DVD Store can be found in the online appendix [28].

We build a private cloud by using an open source cloud, Cloudstack 4.2.0 [7], with an integrated load balancer - NetScaler VPX 10.1[24]. NetScaler VPX is a virtual NetScaler appliance that includes load balancing/traffic management, application acceleration, application security, and offload functionality. Multiple reports and Citrix documents confirm that Netscaler is the state of the art load balancing and provisioning tool that gives us the ability to compare PRESTO with the baseline approach that is considered to be one of the best in the cloud computing industry.

4.3 Methodology

A key driver for choosing an experimental methodology is to compare the values of the dependent variable, throughput for subject applications given the following independent variables: a cloud platform, manually created resource provisioning scripts, user loads, and PRESTO. User loads are simulated for five, 15 and 30 users. An experiment involves randomly choosing client requests for transactions and measuring an average throughput. Since random URLs is unlikely to show the worst performance of the applications, we expect that an average throughput will be higher compared to the one that results from using the inputs selected in FOREPOST.

Recall that FOREPOST automatically constructs behavioral models of applications to choose inputs and user loads for which the application's throughput falls below some acceptable level. For these inputs and the predefined user loads we experiment with different provisioning strategies. Our goal is twofold: 1) we show that different provisioning strategies lead to a large variability in the resulting throughput of the applications, and 2) given that resources have different costs, we show that PRESTO can choose a provisioning strategy that reduces the cost of resource provisioning and improve the performance of the applications when they lose their throughput. We aligned our methodology with the guidelines for statistical tests to assess randomized approaches in software engineering [1, 2]. Given the high variability in the resources allocated to different applications, we execute each experiment multiple times to perform statistical tests and draw reliable conclusions from these tests.

4.3.1 Forming the Load

In JPetStore, the GUI front end is web-based and it communicates with the J2EE-based backend that accepts `HTTP` requests in the form of URLs. Recall that a set of URL requests is defined as a *transaction*. The backends of the subject applications can serve multiple transactions from multiple users concurrently. Test scripts are written using JMeter [16], which generates a large number of virtual users who send `HTTP` requests to web servers of AUTs thereby creating significant workloads. We limit the number of URLs in each transaction to 50, since we observed that users explored approximately 50 URLs before switching to other activities.

4.3.2 Experimenting With Performance Bottlenecks

To determine how well PRESTO allows the cloud to provision resources to maintain good performance of applications, we push the subject applications to worsen their throughputs by injecting computationally intensive operations into their source code. We consider CPU and database performance bottlenecks. CPU bottlenecks perform computationally intensive operations, e.g., arithmetic computations in a loop. Adding more CPUs to a VM can improve the performance of applications with CPU bottlenecks, especially if these bottlenecks are executed by multiple threads. Database bottlenecks address database locking strategies, so resources are locked and applications cannot proceed because one transaction is waiting on resources that are held by some other transactions. We randomly seeded nine CPU and nine database bottlenecks into JPetStore to create two versions (CPU and database version).

4.3.3 Resource Perturbation Modes

During performance testing, stakeholders perturb resource provisioning by applying scaling operators (see Section 2.3) to determine if the throughput of applications can be improved by assigning more resources to VMs. Baseline experiments are carried out using the basic cloud infrastructure, which was one VM with 1.0 GHz CPU and 1.0 GB memory. Different operators are shown as following. Δ_1, Δ_2 and Δ_3 are scale up operators, and Δ_4 is a scale out operator.

Δ_1: one VM with 1.0 GHz CPU and 1.5 GB RAM;
Δ_2: one VM with 1.5 GHz CPU and 1.0 GB RAM;
Δ_3: one VM, two 1.0 GHz core CPUs, 1.0 GB RAM;
Δ_4: two VMs, one 1.0 GHz CPU, 1.0 GB RAM each.

4.4 Threats to Validity

A threat to the validity is that our subject programs are relatively small; however, we used these applications since they are open-source and have been previously used for evaluating performance testing approaches [30, 15]. It is hard to obtain access to large enterprise-level applications, and increasing the size of subject applications is unlikely to affect the time and space demands of our analysis because PRESTO only considers approximations of the behaviors of these applications.

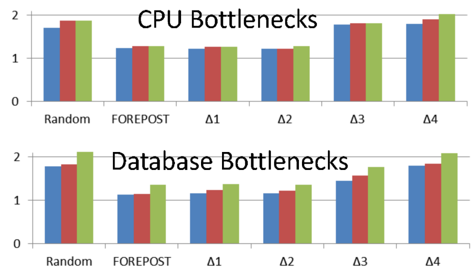

Figure 2: Throughputs for JPetStore using PRESTO on Cloudstack. The X axis shows the throughputs (URLs per second).

A threat to validity is that application's behavioral models are easier to learn for smaller applications, however this is not a point that we address in this paper. We rely on our previously developed tool FOREPOST to learn behavioral models in this paper, however, other approaches for obtaining such models can be used in PRESTO [25, 20]. Since the focus of the paper is on provisioning strategies, we leave the work on experimenting with other approaches for deriving behavioral models for the future.

Another threat to validity relates to the fact that PRESTO uses FOREPOST for learning provisioning strategies. We do not claim that FOREPOST is able to learn sound and complete behavioral models. FOREPOST may miss some of the bottlenecks (and thus, it may miss opportunity to explore testing provisioning strategies in that context). However, in real contexts this may be less of a problem, especially when some of the "typical" usages of the application are known beforehand. Yet, we leave investigation on how undetected bottlenecks can impact performance of the applications deployed in the cloud for future work.

A threat to validity may come from relatively small loads which include at most 30 users simultaneously. However, our underlying experimental cloud platform has limited capabilities, and this threat is countered by the load chosen in a balanced way with respect to available resources. By increasing the load by five orders of magnitude, the underlying cloud platform capabilities would be increased by the same order and our experimental evaluation will stand.

5. EXPERIMENTAL RESULTS

The experimental results are shown in Figure 2. The left and right figures show experiments with CPU and database bottlenecks respectively. Experiments with the random inputs (i.e., Random) and selected input chosen in FOREPOST are common in all graphs. Other experiments are carried out for cloud configurations Δ_1-Δ_4 that we described in Section 4.3. Each experiment shows three bars. The blue, red, and green bars show throughputs for five, 15, and 30 users respectively. Exploratory performance random testing is notoriously difficult to find input types and loads to worsen an AUT's throughput below some acceptable level. Thus, we expect that the throughput for Random experiments will be higher as compared to the throughput for the selected inputs. As Fig. 2 shows, the throughput for selected-input experiments shows an average drop in performance by more than 50% as compared to an average throughput for the corresponding Random experiments, which demonstrates that FOREPOST effectively selects inputs that significantly reduce the throughput. We take the levels of throughput for selected-input experiments as baseline levels for the corresponding applications and the cloud configurations.

The experiment with selected inputs shows how the throughput falls below some unacceptable level, and the experiments for $\Delta_k, 1 \leq k \leq 4$, show how PRESTO provisions resources proactively to VMs to increase the throughput levels compared to the unacceptable level. Δ_1 scales up the VMs by 0.5GB of RAM and Δ_2 scales up the VMs by increasing the CPU speed by 0.5GHz. However, it improves the throughput by less than 5%. Δ_3 scales up the VMs by adding one more CPU and it increases throughput by more than 40% in the CPU version and by around 30% in the database version compared to the throughput for selected-input experiments. Δ_4 allows two VMs executing requests in parallel. The throughput for Δ_4 increases almost 50% as compared to the throughput for selected-input experiments. A key here is that using the scale out operator to parallelize client request processing may achieve better throughput even with more users compared to the baseline level, thus improving the scalability of the deployed applications.

However, the scale out operator is expensive – starting and maintaining a new instance of VM leads to a higher cost compared to provisioning more RAM or CPU. Since we conduct separate experiments with CPU and database bottlenecks, our goal is to determine if PRESTO finds more economic provisioning strategies to improve the scalability of the AUTs. Note that CPU bottlenecks can be alleviated by adding more CPUs. Specifically, the scale up operator Δ_3 adds one more CPU while keeping the same quantity of RAM and the same number of VMs. On average, the difference between throughputs with Δ_3 and Δ_4 is less than 8% for CPU bottlenecks and it is over 20% for the database bottlenecks. Other scale up operators that add different quantities of RAM show little to none improvement. Naturally, PRESTO selects the configuration Δ_3 as a cheaper one for CPU bottlenecks and Δ_4 for database bottlenecks. **Our conclusion is to support RQ_1 in stating that not only does PRESTO determine effective provisioning strategies, but it also chooses them economically, so that stakeholders can balance the cost versus scalability when running applications in the cloud.**

Recall that we re-ran multiple experiments with PRESTO, since these experiments involve the random strategy. To statistically compare throughput values, we used one-way ANOVA and t-tests for paired two sample to evaluate the null hypothesis H_0 (Section 4.1). The results of t-tests showed that the applications' throughputs of PRESTO-based strategies Δ_1 and Δ_2 were comparable to the throughput of selected inputs with basic cloud infrastructure. However, for Δ_3 and Δ_4, most of other p-values were much smaller than 0.05, implying that the applications' throughputs when using other different PRESTO-based strategies were statistically significantly different as compared to the baseline approach. We also used one-way ANOVA to evaluate H_0. The results show that all p-values were substantially larger than the critical value (2.246). Hence, we reject H_0 and conclude that there is statistically significant difference in throughputs for the subject applications while using PRESTO-based strategies compared to the baseline ones.

PRESTO builds performance model for applications as a function that maps the inputs to outputs, finding performance rules that guide the input selection and pinpoint computationally intensive paths. In our experiments, it needed less than two hours to converge to stable rules for JPetStore. Once we obtained performance rules, we tried different provisioning strategies until we found the appropriate strategy that made the throughput for the application to increase to an acceptable level. There was no other manual effort needed in this process. **We support RQ_2 in stating that PRESTO is fast and efficient in learning provisioning strategies.**

Summary. Based on experimental results, we answer affirmatively to RQ_1 that the PRESTO provisioning strategies are effective in terms of improving the AUTs' performance and making them scalable. Moreover, once we obtained performance rules, it is easy to map the provisioning strategies to performance rules without much manual effort, thus we affirmatively answer RQ_2.

6. RELATED WORK

Learning rules helps stakeholders to optimize distributed systems for dynamically changing workloads [34, 31, 22]. In contrast, PRESTO uses feedback-directed adaptive test scripts to locate most computationally intensive execution profiles and bottlenecks.

Several papers focused on improving the performance of applications deployed in the cloud [17, 27, 5, 29, 14, 6, 12, 33]. Klein et al. [19] defined a self-adaptation programming paradigm to “skip" optional functionality in the cloud-deployed applications. Frey et al. [8] used a simulation-based genetic algorithm for finding optimized cloud deployment options for the software in the cloud. An approach, ATUoCLES, allows collecting execution information for applications, which have all the logic to scale up and down automatically [9]. Spinner et al. proposed a model-based approach to improve AUT performance by adding/removing VMs [33]. However, none of these approaches analyze impact of specific inputs on the performance of deployed programs and efficient resource allocation in the cloud-based environments, which is done in PRESTO.

7. CONCLUSION AND FUTURE WORK

Our novel solution for *Provisioning Resources with Experimental SofTware mOdeling (PRESTO)* enhances cloud elasticity by learning and refining models of under-constrained applications throughout performance testing and using these models stakeholders can craft resource provisioning strategies for the cloud that are highly tailored for specific applications. Experimental results suggest that PRESTO is effective and efficient - up to 40% better response in provisioning resources on average when the AUT throughput worsened significantly. In summary, we extend the theory of cloud computing by utilizing performance testing in its load balancing and resource provisioning. We believe that our work is a successful attempt of using software engineering artifacts to guide cloud deployment of software. The future work will involve automatically searching for scaling operators to (de)allocate different resources to VMs and determining the provisioning strategies to maintain AUT's performance at an acceptable level.

Acknowledgements

This work is supported in part by the NSF IIP-1547597 and NSF CCF-1217928 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

8. REFERENCES

[1] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE '11*.

[2] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *STVR*, 2012.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*.

[4] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News'09*.

[5] P. C. Brebner. Is your cloud elastic enough?: Performance modelling the elasticity of infrastructure as a service (iaas) cloud applications. In *ICPE '12*.

[6] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *ASE'11*.

[7] Cloudstack. http://cloudstack.apache.org/.

[8] S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *ICSE '13*.

[9] A. Gambi, W. Hummer, and S. Dustdar. Automated testing of cloud-based elastic systems with autocles. In *ASE '13*.

[10] Google. Auto scaling on the google cloud platform. https://cloud.google.com/resources/articles/auto-scaling-on-the-google-cloud-platform.

[11] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *ICSE'12*.

[12] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, and S. Kounev. Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments. *SOCA'14*.

[13] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for cloud platforms. In *ICPE '12*.

[14] P. Jamshidi, A. Ahmad, and C. Pahl. Autonomic resource provisioning for cloud-based software. In *SEAMS '14*.

[15] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *ICSM '09*.

[16] JMeter. https://jmeter.apache.org.

[17] K. Johnson, S. Reed, and R. Calinescu. Specification and quantitative analysis of probabilistic cloud deployment patterns. In *HVC'11*.

[18] JPetStore. http://sourceforge.net/projects/ibatisjpetstore.

[19] C. Klein, M. Maggio, K.-E. ρ Arzén, and F. Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *ICSE '14*.

[20] D. Lo and S. Maoz. Scenario-based and value-based specification mining: Better together. In *ASE '10*.

[21] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ICSE '08*.

[22] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk. Forepost: Finding performance problems automatically with feedback-directed learning software testing. *EMSE*, 2016.

[23] H. Mendelson. Economies of scale in computing: Grosch's law revisited. *Commun. ACM*.

[24] NetScaler. http://www.citrix.com/netscalervpx.

[25] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *ASE '14*.

[26] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining precise performance-aware behavioral models from existing instrumentation. In *ICSE Companion '14*.

[27] D. Perez-Palacin, R. Calinescu, and J. Merseguer. Log2cloud: Log-based prediction of cost-performance trade-offs for cloud deployments. In *SAC '13*.

[28] PRESTO. http://www.cs.wm.edu/semeru/data/ICPE16-PRESTO/.

[29] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *CAC '13*.

[30] A. Shankar, M. Arnold, and R. Bodik. Jolt: Lightweight dynamic analysis and removal of object churn. In *OOPSLA '08*.

[31] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik. Automating performance bottleneck detection using search-based application profiling. In *ISSTA '15*.

[32] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer'05*.

[33] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith. Runtime vertical scaling of virtualized applications via online model estimation. In *SASO'14*.

[34] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Machine learning for on-line hardware reconfiguration. In *IJCAI'07*.