Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment

Pengcheng Xiong [†‡1], Yun Chi [‡2], Shenghuo Zhu [‡3], Hyun Jin Moon [‡4], Calton Pu [†5], Hakan Hacıgümüş [‡6]

[†]*School of Computer Science, Georgia Institute of Technology*
266 Ferst Drive, Atlanta, GA 30332, USA
[1,5]{pxiong3,calton}@cc.gatech.edu

[‡]*NEC Laboratories America*
10080 North Wolfe Road, SW3-350, Cupertino, CA 95014, USA
[2,3,4,6]{ychi,zsh,hjmoon,hakan}@sv.nec-labs.com

***Abstract*—In a cloud computing environment, resources are shared among different clients. Intelligently managing and allocating resources among various clients is important for system providers, whose business model relies on managing the infrastructure resources in a cost-effective manner while satisfying the client service level agreements (SLAs). In this paper, we address the issue of how to intelligently manage the resources in a shared cloud database system and present SmartSLA, a cost-aware resource management system. SmartSLA consists of two main components: the system modeling module and the resource allocation decision module. The system modeling module uses machine learning techniques to learn a model that describes the potential profit margins for each client under different resource allocations. Based on the learned model, the resource allocation decision module dynamically adjusts the resource allocations in order to achieve the optimum profits. We evaluate SmartSLA by using the TPC-W benchmark with workload characteristics derived from real-life systems. The performance results indicate that SmartSLA can successfully compute predictive models under different hardware resource allocations, such as CPU and memory, as well as database specific resources, such as the number of replicas in the database systems. The experimental results also show that SmartSLA can provide intelligent service differentiation according to factors such as variable workloads, SLA levels, resource costs, and deliver improved profit margins.**

***Index Terms*—cloud computing, virtualization, database systems, multitenant databases**

I. Introduction

The cloud computing model is changing how the technology solutions are accessed and consumed by the users. In its spirit, the infrastructure resources and computing capabilities are provided as a service to the users by the cloud providers. The users can leverage a range of attractive features, such as resource elasticity, cost efficiency, and ease of management. The cloud computing model also compels the rethinking of economic relationships between the provider and the users based on the cost and the performance of the services. Cloud-based data management services, which are typically as part of Platform-as-a-Service (PaaS) offerings, is one of the most significant components of the new model [1], [2], [3]. In PaaS, service providers generate the revenue by serving the client queries where the revenue is determined by the delivery service-level agreements (SLAs). In order to provide such service, the PaaS providers may rent their resources from Infrastructure-as-a-Service (IaaS) providers by paying for the resource usage. Hence, from a PaaS provider point of view, the profit is determined by two factors: revenue and cost.

PaaS providers may have two different types of problems: (1) management of big data by scale-out [4] and (2) consolidation of many small data for cost efficiency [2]. With the former, they need to find the right scale to meet the SLA of a single client, e.g., how many machines are needed to serve a given workload. In this paper, we focus on the latter, a.k.a. *multitenant databases*, where the service provider looks for the opportunity of cost reduction through tenant consolidation.

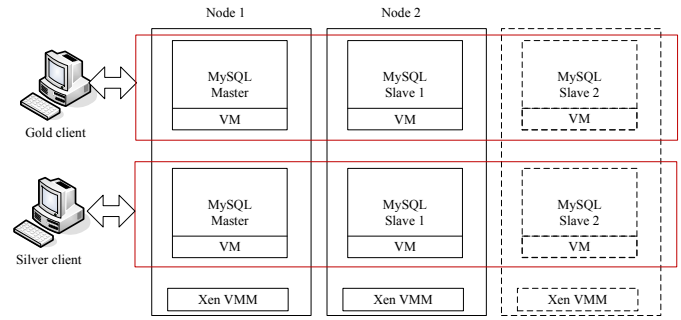


Fig. 1. An illustration example where two kinds of clients share the same database resources.

In multitenant databases, there are several different levels of sharing [5], including private virtual machine (VM), private database, private table, and shared table. In this paper, we consider the case of private virtual machine, where each *tenant* database runs in its own virtual machine. This level of sharing allows us to explicitly control the system resources allocated for each VM, or the corresponding tenant. The current virtualization technologies allow packing of a large number of VMs into physical machines thereby increasing the cost efficiency of infrastructure resources [6]. While it seems quite attractive to consolidate multiple tenants into a physical machine, it requires careful planning and management in order to satisfy tenants' SLAs.

[1] The work was done while the author was at NEC Labs America.

Let us consider an illustrative example shown in Fig. 1. In this example, we assume that there are two kinds of clients, e.g., a gold and a silver one for the cloud database service provider. As their workload demand changes, they add or remove database slaves. The clients share the hardware resources where master and slaves are contained in a separate VM which is common in many web applications hosted on large clusters [7]. The service provider charges an agreed-upon fee if it delivers the service by meeting the SLAs and pays a penalty if it fails to meet the SLAs. Consequently, a failure to deliver on SLAs results in higher penalty for the gold client. In reality, of course, there may be more than two kinds of clients.

The cloud database provider has two main goals: (1) meeting the client SLAs and (2) maximizing its own profits. It is obvious that intelligent management of the resources is crucial for the service provider to achieve these goals. The service provider should intelligently allocate limited resources, such as CPU and memory, among competing clients. On the other hand, some other resources, although not strictly limited, have an associated cost. Database replication is such an example. Adding additional database replicas not only involves direct cost (e.g., adding more nodes), but also has initiation cost (e.g., data migration) and maintenance cost (e.g., synchronization). The key to the successful management of resources are as follows:

Local Analysis : The first issue is to identify the right configuration of system resources (e.g., CPU, memory etc.) for a client to meet the SLAs while optimizing the revenue. Answers to such a question are not straightforward as they depend on many factors such as the current workload from the client, the client-specific SLAs, and the type of resources.

Global Analysis : The second issue that a service provider has to address is the decision on how to allocate resources among clients based on the current system status. For example, how much CPU share or memory should be given to the gold clients versus the silver ones, when a new database replica should be started, etc. Answers to such decisions obviously rely on the result of the above *Local Analysis* decisions.

In this paper, we address the issue of how to intelligently manage the resources in a shared cloud database system by considering those Local and Global Analysis policies. We present SmartSLA[1], a cost-aware resource management system. SmartSLA consists of two main components: the system modeling module, which mainly answers the *Local Analysis* questions, and the resource allocation decision module, which mainly answers the *Global Analysis* questions. The system modeling module uses machine learning techniques to learn a model that describes the potential profit margins for each client under different resource allocations. Based on the learned model, the resource allocation decision module dynamically adjusts the system resource allocations in order to optimize expected profits.

[1]SmartSLA stands for "Resource Management for Resource-Sharing Clients based on Service Level Agreements".

The rest of this paper is organized as follows. In Section II, we provide background information about SLAs and describe our system setting. In Sections III and IV, we describe the system modeling module of SmartSLA. In Sections V and VI, we describe the resource allocation module of SmartSLA. In Section VII, we discuss related work. Finally, in Section VIII, we give conclusions and future directions.

II. Background

In this section, we provide background information. We first introduce service level agreements (SLAs). Then, we give a high-level description of the test bed for our experimental studies. More details about the test bed will be presented in the corresponding sections.

A. Service Level Agreements

Service level agreements (SLAs) are contracts between a service provider and its clients. SLAs in general depend on certain chosen criteria, such as service latency, throughput, availability, security, etc. In this paper, we focus on service latency, or response time.

While SLA cost function may have various shapes, we believe that a staircase function is a more natural choice used in the real-world contracts as it is easy to describe in natural language [8], [9]. We use a single step function for SLA in our paper as a reasonable approximation. Fig. 2(a) shows such an example. The figure denotes that if the response time of the query q is shorter than X_q, then the service provider obtains a revenue R. Otherwise, the service provider pays a penalty P back to the client.

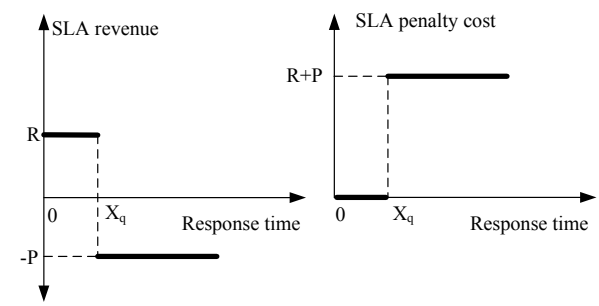


Fig. 2. (a) SLA revenue function. (b) SLA penalty cost model that is equivalent to the revenue model in (a).

From the given SLA revenue function, we derive the SLA penalty cost function, as shown in Fig. 2(b). Here we assume that the service provider has been already paid R in advance and if the query q's response time is later than X_q, the service provider will pay $R + P$ back to the client. With this relationship, SLA revenue maximization is equivalent to SLA penalty cost minimization, and so we focus on the latter in the rest of our paper. For simplicity, we denote the start time and the response time of query q as q_{start} and q_{time}, respectively, and also assume $R + P = 1$ for the moment. So we have the SLA penalty cost of query q as

$$P(q) = \begin{cases} 0 & \text{if } q_{time} \leq X_q \\ 1 & \text{otherwise} \end{cases}$$

We define the average SLA penalty cost *AC* as the sum of SLA penalty cost over the total number of queries L, i.e.,

$$AC = \frac{1}{L} \sum_{q} P(q).$$

For example, if the client sends 10 queries and 5 of them miss the deadline, then the sum of SLA penalty cost is 5 and the average SLA penalty cost is 0.5.

As we motivated above, the database service provider's goal is to maximize its profits while satisfying the client SLAs. Therefore, the SLA penalty cost is the main metric when we define and solve optimization problems throughout the paper.

B. Our Test Bed

We start by giving a high-level description of the experimental setup that we use throughout the paper. We discuss the experiment environment, the workload generator, and the system architecture. More details are described in the following sections.

1) Experiment Environment: For the test bed, we use MySQL v5.0 with InnoDB storage engine as the database server and use the built-in MySQL replication functionality for scale-out solution. In MySQL, replication enables data from one MySQL database server (the master) to be replicated to one or more MySQL database servers (the slaves). Replication is asynchronous and as a consequence, slaves need not to be always connected to the master in order to receive updates from the master. Each part of the MySQL replication (either master or slave) is hosted in one Xen virtual machine [10], [11]. The physical machines are AMD 3GHz dual core, 2GB RAM PC with Gigabit Ethernet connected with switches, running Linux kernel 2.6.18 with Xen 3.0. We reserve one of the cores and 1GB memory for dom0 and pin the virtual machines to the other core and let them share the rest 1GB memory.

2) Workload Generator: We develop workload generators to emulate clients where new queries are generated independently to the completion of previous queries [12]. The arrival rate of queries follows a Poisson distribution with the rate set in each test. We choose Poisson distribution because it is widely used to model independent arrival requests to a website. Extending our work by including these non-Poisson-processes-based models is one of our future directions.

3) System Architecture: Fig. 3 illustrates the system architecture of our test bed as well as SmartSLA, our intelligent resource manager. We assume that the system is monitored continuously and the resource allocations can be changed periodically in each time interval. In all experiments, we set the time interval to be 3 minutes. We choose this value because too short intervals cannot capture the randomness of query types while too long intervals make SmartSLA less responsive.

Our SmartSLA manager contains two modules. The first one is a system modeling module, which learns a model for the relationship between the resource allocation and expected cost for a single client. The second one is a resource allocation module, which dynamically makes decisions on the changes of the resource allocation among clients. We will introduce these two modules in detail in the following sections.

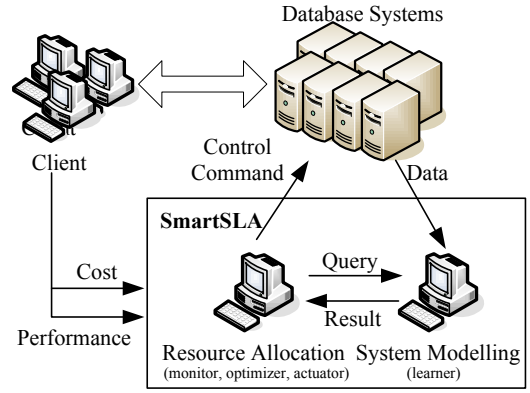


Fig. 3. The architecture of our test bed.

III. System Modeling—Statistical Analysis

In this section and the next one, we describe the system modeling module in SmartSLA. We first investigate the question of "How is the system performance correlated with the system configuration in terms of SLA cost?". More specifically, the service provider wants to know how the system performance would change (under an appropriate cost model) if the resource allocation is changed. We will show the correlation between the SLA cost and some specific resource types.

A. Benchmark Queries and SLAs Used in the Study

We use the TPC-W benchmark [13]. The setting of the benchmark is 100 EBs, 10K items. According to our test and also previous works [14], these queries are CPU-bound when the database can reside in the memory. Therefore the whole database size is set to about 280MB to make sure it can fit in the memory.

Most virtual machine products currently provide mechanisms for controlling the allocation of CPU and memory resources to VMs, but do not provide mechanisms for effectively controlling other resources, such as I/O bandwidth [6]. Hence, in this study we focus mainly on CPU-bound queries and CPU and memory resources.

We use the TPC-W Ordering mix for workload mix in which the browsing requests and the ordering requests are 50%, respectively. The requests contain both write and read queries. The write queries are sent only to the master while the read queries are sent only to the slaves in a round-robin manner.

For each TPC-W query, we assign it an SLA penalty cost function as shown in Fig. 2(b). To decide the X_q value in the SLA, we run each query in a virtual machine with maximum possible resources, i.e., 100 CPU shares and 1024MB memory, to get the average response time. Then we define X_q as two times of the average response time.

B. Statistical Analysis

In order to obtain a high-level overview of the relationship between system resources and database system performance,

we conduct several experiments and statistical analysis on how the parameters, such as CPU share, memory size, client workload, and replica number, affect the system performance. With respect to system performance, we choose to use the average SLA penalty cost per query, i.e., AC. According to our previous definition, the total SLA penalty cost is the average SLA penalty cost multiplied by the workload L.

We measure the database system performance under different parameter settings. Specifically, we set the range of the CPU shares of the virtual machines between 20 and 80; the memory size between 256MB and 768MB; the number of replicas between 2 and 5; and the arrival rate between 1 and 12 queries per second. For the purpose of statistical analysis, we randomly sample 500 data points with different parameter settings. For each sample, we run the experiment for 9000 seconds with 180 second warmup/cool-down time. To view the results, we plot the distribution of the average SLA penalty cost under different settings of the parameters. Notice that we randomly tested all the combination of the above settings. However, in this section, we only report the marginal distributions on each of the parameter independently to make the discussion easy to follow. That is, we plot one parameter at a time.

In Fig. 4 we show how the system performance, in terms of average SLA penalty cost, changes when the parameters change. In each subfigure, the x-axis represents the parameter setting under investigation and the y-axis indicates the performance that corresponds to the given setting, in terms of average SLA penalty cost per query,

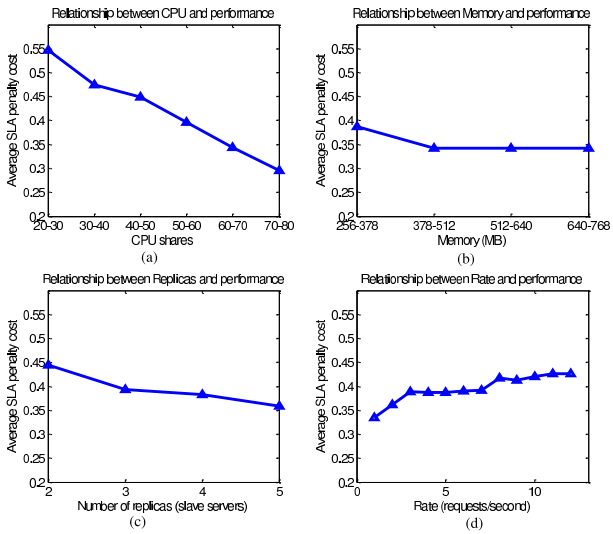


Fig. 4. Relationship between system resources and database system performance.

1) CPU: Fig. 4(a) shows the distribution of average cost under different CPU shares. From the plot we can see a clear correlation between the CPU share and the corresponding average cost. As the CPU share increases, the average cost is reduced in a near-linear fashion. This result is not unexpected, because in these experiments we used the CPU-bound queries from the benchmark.

2) Memory: Fig. 4(b) shows the distribution of average cost when we change memory size from 256MB to 768MB. From the figure we can see that increasing memory can help reduce the average cost initially. However, when memory is larger than certain threshold, e.g., larger than 512MB, adding more memory does not help further. This phenomenon is due to the cache effect—the database of TPC-W is rather small, only about 280MB, and so when the memory size is large enough, memory stops to be a main bottleneck. Thus, the relation between the average cost and memory size is not linear as we expected.

3) Replica: Fig. 4(c) shows the distribution of average cost versus the number of replicas. From the plot we can see a correlation between the number of replicas and the corresponding average cost. As the number of replicas increases, the average cost is reduced. This result is also expected: increasing the number of replicas reduces the chance of queuing delay (i.e., the chance for a newly arrived query to find all replicas being busy and so has to wait) and reduces the average SLA penalty cost.

4) Rate: Fig. 4(d) shows the distribution of average cost under different arrival rates. We can see a trend that as the arrival rate increases, the average query cost increases slowly. This result suggests that when the arrival rate becomes higher, the system is under more stress and there is a higher chance for a query to have queuing delay and so its average cost grows.

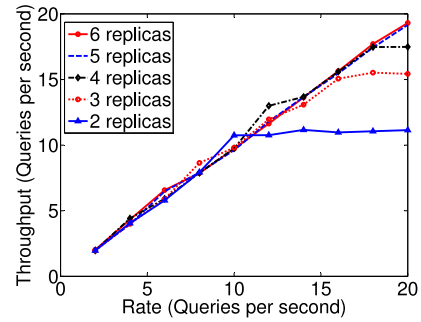


Fig. 5. System throughput under different query arrival rates.

We also perform stress test experiments where we increase the rate to up to 20 queries/second as shown in Fig. 5. In our stress test, we fix the CPU shares to 50 and the memory size to 512MB. We change the number of replicas from 2 to 6, and plot the rate and the real system throughput as shown in Fig. 5. As we can see from the figure, the system is saturated at 10 queries per second when there are only 2 replicas. In order to avoid frequent saturation, we only vary the rate from 1 to 12 queries per second in the experiments.

IV. System Modeling—Machine Learning Techniques

In this section, we present the system modeling module in SmartSLA. The main question we want to answer in this section is "How can we accurately predict the system performance?".

Since the observations in the previous sections are obtained under our specific testbed, using particular benchmark data

and queries, they cannot simply be generalized to other system settings. This motivates us to adopt machine learning techniques to build adaptive models to precisely predict system performance under different resource allocations. The structure of our machine learning strategy is shown in Fig. 6. The appeal of machine learning techniques is that they are *data-driven*, i.e., the *resulting models* may be different for different systems, but the *modeling process* is the same.

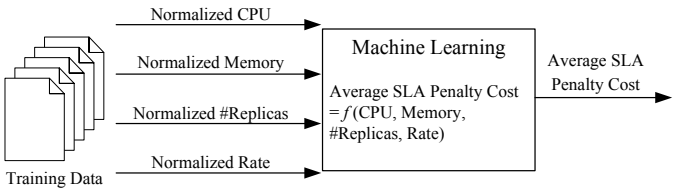


Fig. 6. The architecture of our machine learning strategy.

The system performance is predictable by using the given features with the statistical analysis in the previous section. However, as observed in the previous section, some features such as memory size affects the system performance in a *nonlinear* manner. In addition, we expect that the features affect the system performance in a *joint* fashion, e.g., CPU is the main bottleneck only when the memory size is large enough.

As a result, next we investigate a series of mature machine learning techniques. We start with a standard simple *linear regression* model. However, the accuracy turns out to be unsatisfactory, because of the nonlinear relationship between the cost and some of the resources, as shown in the previous section. Then, we show that the prediction accuracy is improved by the *regression tree* model (e.g. [15]), which takes the nonlinearity into account. To further improve the prediction accuracy, we use a boosting approach, called *additive regression* method [16], using regression tree as the underlining weak learner. We use those mature regression methods in WEKA package [17], because the focus of this work is how to apply machine learning techniques in virtualized resource management, not to invent new models.

Before building the models, we have a data-preprocessing procedure to normalize the parameters to the same data range (from 0 to 1). For example, we use the value between 0 and 1 to represent the CPU shares between 0 and 100 by scaling the face value by 0.01 times, and the number of replicas (between 2 and 5) by 0.33. Therefore, the magnitudes of all coefficients of the model are comparable.

A. Linear Regression

Linear regression is one of the most widely used regression techniques. The basic idea of linear regression is to fit a linear model, which is a function describing the relationship between the input parameters (the normalized CPU share, memory size, number of database replicas, request rate) and the output (the average SLA penalty cost), where the mean-squared error is to be minimized. Following standard practice in machine learning, we adopted a 10-fold cross validation to measure the accuracy of the model. The linear model learned by linear regression is as follows:

$$\begin{aligned}\text{Average SLA penalty cost} &= f(\textit{cpu}, \textit{mem}, \textit{\#replicas}, \textit{rate}) \\ &= -0.5210 \times \textit{cpu} - 0.5392 \times \textit{mem} \\ &\quad - 0.1319 \times \textit{\#replicas} + 0.1688 \times \textit{rate} + 0.9441,\end{aligned}$$

where in the formula, *cpu*, *mem*, *#replicas* and *rate* represent normalized CPU shares, memory size, number of database replicas and arrival rate, respectively.

There are several observations that we can obtain from the model. First, the signs of the coefficients in the learned model make sense—increasing CPU share, memory size, and replicas help reduce the average query cost, whereas increased arrival rate increases the system load and therefore increase the average SLA penalty cost. Second, we can roughly estimate that increasing CPU share by 1 reduces the average query cost by $0.5210 \times 0.01 = 0.5\%$ while increasing the number of replicas by one reduces the cost by $0.1319 \times 0.33 = 4.3\%$.

However, as shown in Table I, the standard linear regression model predicts with an error as high as 44% of total standard deviation of the cost.

TABLE I
PREDICTION ERROR OF LEARNING ALGORITHMS

Algorithms	Root mean square error	Relative absolute error
Linear Regression	0.0632	44.0%
Regression tree	0.0414	37.3%
Boosting	0.0317	28.6%

B. Regression Tree

A weak point of the linear regression is that it assumes that the parameters (resource allocation) affect the output (average cost) in a linear fashion. From the statistical analysis in the previous section we can clearly see that some parameters, such as the memory size, impact the average SLA cost in a strongly nonlinear way. We believe this is a main reason for the standard linear regression approach to have poor performance. To address the nonlinearity, we partition the entire parameter spaces into several regions, where linear regression is performed locally in each region. More specifically, we use regression tree [15] to partition the parameter space in a top-down fashion, and organize the regions into a tree style. The tree is constructed by recursively splitting the data points along the optimal partition of a single variable to maximize the residual reduction. To reduce the generalization error, a pruning is performed to reduce the size of the tree. We used an off-the-shelf algorithm M5P [15], an existing regression tree module in the well-known WEKA package [17].

The learned regression tree model is shown in Fig. 7. Compared to the linear regression, Fig. 7 shows that regression tree captures the relationship between the cost and memory size better. For example, the tree shows that when CPU is more than 0.545, i.e., 54.5 shares, there are two linear regression models (LM) that we can use corresponding to different memory shares, i.e., over or below 0.49 (around 512MB). With such a regression tree model, according to Table I, the model error is reduced to 37.3%.

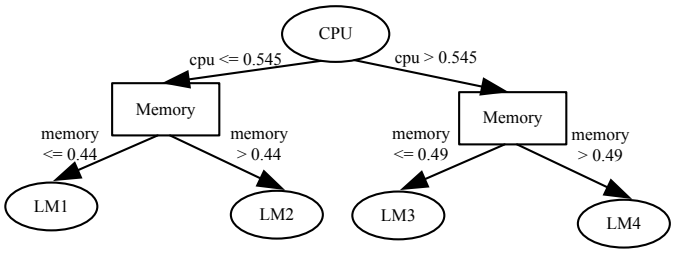


Fig. 7. A regression tree model

C. Boosting Approach

To further reduce the prediction error, we use a boosting approach (cf. [16]) that iteratively obtains weak learners (namely, learners that do not necessarily have good performance individually), and combines the weak learners to create a strong learner which reduces the error. Here we use a boosting approach called additive regression [16] in WEKA package[17], and we use the regression trees described above as weak learners. As shown in Table I, with the boosting, the model error is further reduced to 28.6%.

One negative side of boosting approach is that it is difficult to interpret the model parameters in a simple plain language. As reducing cost is more important than the interpretation in virtualized resource management, we take the model with the least prediction error.

D. Discussion

In summary, in this section we applied a series of mature, but increasingly more sophisticated, machine learning techniques to obtain a system model that can accurately predict the system performance under different resource allocations. The magnitude of absolute error for the model is partly due to the complex system state space. For example, the running time of queries is not necessarily deterministic based only on CPU share and memory share. Concretely, the queries that belong to the same query type (e.g., Search Request) may have different cardinality and also different running neighbors in the run time. While we have captured a significant number of features in the study, the magnitude of the error indicates that more features such as the ones mentioned above may be relevant. We plan to study these unexplored features and analyze their sensitivity to the system performance in the future work.

V. Resource Allocation—CPU and Memory

In this section and the next, we present the resource allocation module in SmartSLA. The key issue is that, given a predictive model (obtained in the previous section), how to intelligently make decisions on resources to minimize the total SLA cost.

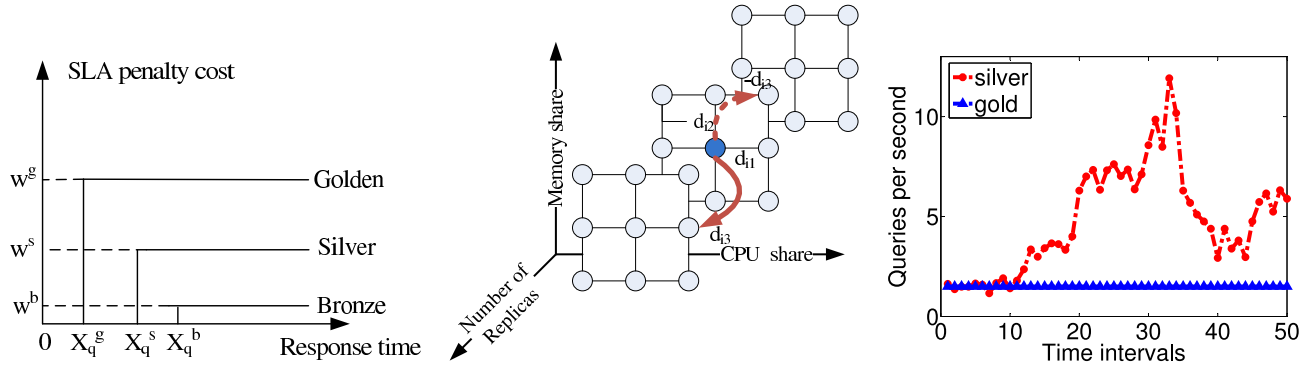
We divide the resource allocation problem into two levels. Resources on the first level, such as CPU share and memory size, have fixed capacity when the number of database replicas is fixed. In such a case, the decision is simply on how to optimally split the resources among clients who share the resources. Resources on the second level, such as number of database replicas, are expandable. That is, for example, the service provider can add additional database replicas, at certain prices, if doing so can reduce the total SLA cost. In this section, we focus on the first level resource allocation, i.e., how to allocate CPU and memory shares while we assume that the number of replicas is fixed. In the next section, we discuss the second level resources allocation, where we use database replica as an example.

TABLE II
Notations

k	Interval
T	The length of the interval
$I(k)$	Infrastructure cost
$M(k)$	The number of replicas
N	Type of clients
$SLA(k)$	Weighted total SLA penalty cost during the k-th interval
q	Query, where q_{start} is the start time of the query q_{time} is the response time of the query
$P(q)$	SLA penalty cost function for query q
$w(i)$	Weight for the i-th class of client
$P_w(q,i)$	weighted SLA penalty cost for the i-th class of clients
$L(i,k)$	The total number of queries for the i-th class of clients during the k-th interval
$cpu(i,k)$	The cpu shares for the i-th class of clients during the k-th interval
$mem(i,k)$	The memory shares for the i-th class of clients during the k-th interval
$AC(i,k)$	The average SLA cost for the i-th class of clients during the k-th interval

A. Multiple Classes of Clients and Weighted SLAs

In the previous sections, we considered only one client. Starting from this section, we consider multiple clients sharing the resources. We rely on the CPU/memory allocation and isolation of the underlying virtualization software – Xen in our case. Since we set upper limit on the CPU time consumable by a particular VM, we have a clear service differentiation between clients with respect to the CPU and memory shares. However, the service differentiation between clients with respect to the other system resources like the disk and network I/O is beyond our control. We perform experiments to show the interference with respect to the other system resources other than CPU and memory among different clients sharing the same physical machine is small for certain workloads. For example, we conducted the following test with the same TPC-W benchmark and the same setup as in Section III. We compare the performance of a client (we call it client A) in two cases. In the first case, client A is the only client occupying the physical machine. For example, only 20 shares of CPU is allocated to A and the remaining 80 is idle. In the second case, we have another client B consuming the resources unused by A. We also try different shares in experiments. The experiments show that the performance difference is small, with less than 5%. This is mainly due to the CPU-bound queries that we used. The result shows that the statistical analysis for a client is valid even when the client is sharing the physical machine with others.



(a) Weighted SLA penalty cost. (b) Searching directions. (c) Workload trace for gold and silver clients.

Fig. 8. Weighted SLA penalty cost, search directions and workload traces

We start by specifying the total SLA cost. We will use the notations in Table II in our discussion. Basically, we use two indices: k means the k-th time interval and i means the i-th class of clients (where i is between 1 and N).

Compared with the previous sections, we consider N classes of clients instead of a single one. For the i-th class of clients we need an SLA penalty cost function $P(q,i)$ which depends on the deadline X_q^i. For the i-th class of clients, we use a weight $w(i)$ to denote the penalty when query q misses the deadline. For example, as shown in Fig. 8(a), assume that we have gold, silver, and bronze clients, then the deadlines for them are X_q^g, X_q^s and X_q^b, respectively and the SLA penalty costs for them are w^g, w^s and w^b, respectively.

We define the weighted SLA penalty cost function as:

$$P_w(q,i) = P(q,i) \times w(i)$$

Within the k-th interval, we define the average SLA penalty cost for the i-th class of clients as the total SLA penalty cost over the total number of queries $L(i,k)$ during the interval.

$$AC(i,k) = \frac{1}{L(i,k)} \sum_{T\times(k-1)\le q_{start}<T\times k} P(q,i)$$

Given that there are N classes of clients, we have the total weighted SLA penalty cost for the k-th interval as

$$\begin{aligned} SLA(k) &= \sum_{i=1}^{N} \sum_{T\times(k-1)\le q_{start}<T\times k} P_w(q,i) \\ &= \sum_{i=1}^{N} AC(i,k) \times L(i,k) \times w(i). \end{aligned}$$

Our goal is to minimize the total weighted SLA penalty cost under the constraints on resources.

B. Dynamic Resource Allocation in SmartSLA

From the system model learned in the previous section, for a given workload (measured in real time) and the corresponding SLAs from the clients, theoretically we are able to find an optimal resource allocation that offers the minimum expected SLA cost. However, such a theoretical solution is very difficult to find. Recall that the system model we learned is a highly non-linear one (it consists of weighted sum of several regression trees, where each regression tree cuts the feature space into several regions). With such a non-linear model, which may have many local minimums, finding a globally optimal solution is challenging. Therefore in SmartSLA, we apply a grid based search where the grids for CPU are separated every 5 shares and those for memory size are separated every 5MB.

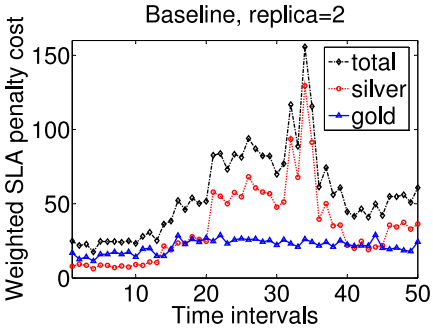
When the previous time interval ends, SmartSLA decides the optimal direction for each client. We use a 3-D array, i.e., $d_i = [d_{i1}, d_{i2}, d_{i3}]$ to denote the direction for the i-th client, as shown in Fig. 8(b). The first two dimensions, i.e., d_{i1} and d_{i2} denote the directions for CPU and Memory tuning respectively, which belong to the first-level resource allocation. The third dimension d_{i3} is used for replica tuning, which belongs to the second-level resource allocation. We use $D = [d_1, d_2, ..., d_N]$ to denote the global decision. Since the clients share the machines, we define the third direction as $D_3 = d_{13} = d_{23} = ... = d_{N3}$. We also define $M(k)$ as the number of replicas during the k-th interval for all the clients as they share the machines.

We define the $cpu(i,k)$, $mem(i,k)$ and $L(i,k)$ as the cpu shares, memory shares and rate (workload) for the i-th class of clients during the k-th interval.

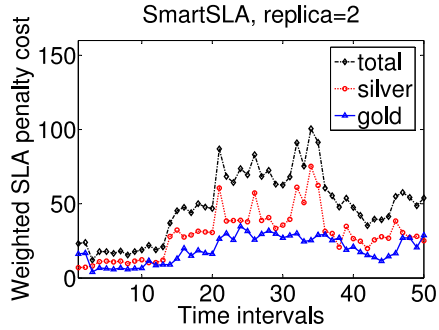
We formulate the problem as allocating the system resources to minimize the total SLA penalty cost as follows. f_i is the model that is learned by machine learning techniques according to different SLA penalty costs for the i-th client,

$$\begin{aligned} &\textit{next direction} = \arg\min_{D} \textit{SLA}(k) \\ &\text{s.t.} \quad \textit{SLA}(k) = \sum_{i=1}^{N} \textit{AC}(i,k) \times L(i,k) \times w(i) \\ &\qquad \textit{AC}(i,k) = f_i(\textit{cpu}(i,k-1) + d_{i1}, \textit{mem}(i,k-1) + d_{i2}, \\ &\qquad\qquad M(k-1), L(i,k)) \\ &\qquad \sum_{i=1}^{N} d_{i1} = 0, \qquad \sum_{i=1}^{N} d_{i2} = 0 \qquad (1) \\ &\qquad D_3 = d_{13} = d_{23} = ... = d_{N3} = 0 \qquad (2) \end{aligned}$$

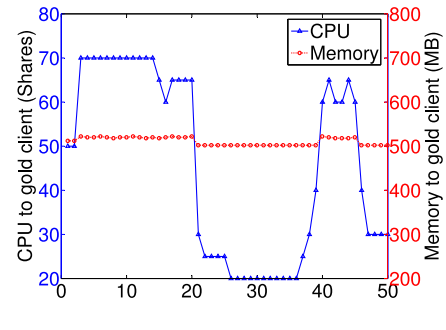
For example, for the i-th client, the previous CPU and memory are *cpu*(i,k-1) and *mem*(i,k-1), e.g., 30 shares and 512MB



(a) Weighted SLA penalty cost for baseline with number of replicas=2.



(b) Weighted SLA penalty cost for SmartSLA with number of replicas=2.



(c) CPU/Memory to the gold client with number of replicas=2.

Fig. 9. Experimental result for CPU and memory resource allocation

respectively. If the direction is $d_{i1} = 10$ and $d_{i2} = -100$, then the next CPU and memory for this client are 40 shares and 412MB respectively. Since we fix the number of machines to use for the replicas in this section, for the third direction we have $d_{13} = d_{23} = ... = d_{N3} = 0$. Now every term in the optimization can be calculated or obtained except $L(i,k)$, which denotes the rate for the i-th client. Here we make an assumption that its value is equal to that of the previous interval, i.e., $L(i,k) = L(i,k-1)$.

C. Evaluation

1) Clients, SLAs, and Workloads: In all our experiments, we use two classes of clients, the gold clients and the silver ones. Both of their arrival processes are Poisson process and both of the request mixes are TPC-W ordering. For the silver clients, we use a real workload trace as shown in Fig. 8(c) to evaluate the performance of the system under control. The workload trace is generated based on the Web traces from the 1998 World Cup site [18]. We scale down the rates to fit our experiment environment, e.g., we use 1req/s to mimic 10k rate. For the gold clients, we use a constant workload of 1.56 req/s as shown in Fig. 8(c), which is the average arrival rate of the silver client in the first 10 time intervals (before the burst happens). For simplicity we set the same deadline for the gold and silver clients, i.e., $X_q = X_q^g = X_q^s$. Then we have the same machine learning model for them accordingly. However, we set different weights, i.e., $w^g = 0.2$ and $w^s = 0.1$ for gold and silver clients, respectively. When a request from gold/silver clients misses the deadline, the service provider will pay 0.2/0.1, respectively.

The initial CPU shares and memory shares were set to 50 and 512MB respectively. Each experiment runs over 50 intervals, i.e., 9000 seconds. The warmup and the cool-down times for each experiment are both 180 seconds. It takes about 76 seconds for the learner to build the boosting model which can be done offline. The resource allocation decision can be made in 800ms which is negligible compared with the length of the interval. Each experiment is repeated 5 times and the average of them is reported as the result.

2) First Level Resource Allocation Results: We use the default CPU/Memory allocations (50 shares/512MB) to both of the clients as the baseline case. Fig. 9(a) and Fig. 9(b) show the weighted SLA penalty cost over time for the baseline and SmartSLA. Fig. 9(c) shows the percentage of total CPU and memory that is allocated dynamically to the gold client.

In this experiment, we fix the number of replicas as 2, i.e., $M(k) = 2$. For the overall performance, the total weighted SLA penalty cost is 2802 for the baseline and 2364 for SmartSLA. That is, SmartSLA reduces about 15% SLA penalty cost by dynamically tuning CPU and memory shares between gold and silver clients. We find that CPU is tuned more often and in a wider range than memory in our experiment. For example, the CPU shares changes from 70 to 20 while the memory size only change from 522MB to 502MB. We can divide the whole period into 3 parts and analyze them in more details as below.

The first part starts from the first interval and ends around the 20th interval. During this period of time, both the gold and silver clients have almost the same rate, i.e., around 1.5 queries/second. Since gold queries have higher weight than the silver ones, the SmartSLA intelligently determines to give more resources, e.g., more CPU shares to the gold client than the silver one. Compared with the SmartSLA, the baseline gives equal resources to the clients. The benefit that comes from the silver client is compromised by the cost from the gold client since the gold client has a higher weight. As a result, the weighted total SLA penalty cost is higher than that for SmartSLA.

The second part starts from the 20th interval and ends around 35th interval. During this period, the silver queries come in a large volume as we can see the peak in the world cup trace. However, SmartSLA intelligently detects the burstiness of the rate of the silver client and decides to give more resources to the silver client than the gold client. As we can see in the figure, when the peak comes, the weighted SLA penalty cost jumps to 150 for silver clients in the baseline. However, the SLA penalty cost stays less than 100 with SmartSLA. Had SmartSLA given more resources to the gold client than the silver client, then the benefit that comes from the golden client would have been compromised by the cost from the silver client. This is due to the fact that, although the gold client has a higher weight, the silver client has a very high arrival rate.

The third part starts from around 35th interval. During this period, the rate of silver client first drops and then increases.

SmartSLA intelligently detects the changes in the rate of gold and silver clients and makes right decisions during this period as well.

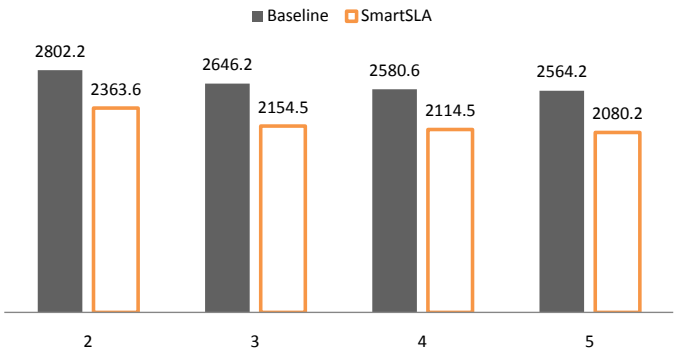


Fig. 10. Total weighted SLA penalty cost with number of replicas=2,3,4,5.

We also conducted the experiments with different number of database replicas, i.e., $M(k) = 2, 3, 4, 5$. Figure 10 shows the total weighted SLA penalty costs for the baseline and SmartSLA. We can get two observations from the graph. (1) As the we add replicas from 2 to 5, in both cases the total weighted SLA penalty cost decreases. This is expected as more replicas will decrease the SLA penalty. (2) Compared with the baseline, SmartSLA always reduces cost. This result verifies that SmartSLA can adaptively allocate resources considering both the request rates and also the weights.

VI. Resource Allocation—Database Replicas

In this section, we focus on the second level resource allocation, i.e., how to tune the number of database replicas to reduce the total cost, where the total cost includes not only the SLA penalty cost but also the infrastructure and action costs. As we have discussed before, the action of tuning the number of database replicas is different from tuning CPU share and memory size, because it involves additional cost models. In this section, we analyze the cost model by changing the number of database replicas and show that by taking this cost model into consideration, SmartSLA can further improve the cost efficiency.

A. Infrastructure Cost

1) Infrastructure Cost Model: From the statistical analysis in Section III and experimental results in the previous section, larger number of database replicas is always beneficial in terms of SLA cost. That is, within the range of our investigation (i.e., the replica number M is between 2 and 5), the overheads of additional database replicas, such as data synchronization, never outpace the further SLA cost reduction brought by the additional replicas. Therefore, the system always operates with maximum number of allowable database replicas. In reality, however, additional database replicas come with costs, which can be due to infrastructure cost (e.g., adding more nodes) or initiation cost (e.g., data migration). A practical resource management system should take such costs into consideration.

From a service provider's point of view, infrastructure cost may involve many factors: hardware, software, DBA expenses, electricity bills, etc. For example, the price table from Amazon Relational Database Service[2] shows that they adopt a simple linear model between the infrastructure cost and the number of machines. We define c as the cost per node per interval. Thus we have $I(k)$ to be proportional to the number of database replicas $M(k)$ as $I(k) = c \times M(k)$. Then the infrastructure cost for direction D can be calculated as $I(k) = I(k-1) + c \times D_3$.

Including such an infrastructure cost, our target function becomes

$$\textit{next direction} = \arg\min_{D} \textit{SLA}(k) + I(k)$$

$$\text{s.t.} \quad \textit{SLA}(k) = \sum_{i=1}^{N} \textit{AC}(i,k) \times L(i,k) \times w(i)$$

$$\textit{AC}(i,k) = f_i(\textit{cpu}(i,k-1) + d_{i1}, \textit{mem}(i,k-1) + d_{i2}, M(k-1) + D_3, L(i,k))$$

$$\sum_{i=1}^{N} d_{i1} = 0, \qquad \sum_{i=1}^{N} d_{i2} = 0$$

$$I(k) = I(k-1) + c \times D_3$$

$$D_3 = d_{13} = d_{23} = \ldots = d_{N3} \in \{-1, 0, 1\}$$

We choose $D_3 \in \{-1, 0, 1\}$ as we want to add/remove at most one a replica during one interval. We give this limitation since adding a replica may involve an action cost as will be shown later.

B. Evaluation with the Infrastructure Cost

We test SmartSLA by using different values of c in the infrastructure model. Table III shows the SLA penalty cost, infrastructure cost, and total cost when $c = 1$ and $c = 3$, respectively. In each table, the first 4 rows are the SmartSLA results with *fixed* number of replicas (2 to 5). In the last row we show the SmartSLA results when SmartSLA is allowed to automatically decide and tune the best number of replicas to use in each time interval.

From the results we can obtain the following observations. (1) When more replicas are used, the SLA penalty cost decreases as expected. But if we take the infrastructure cost into consideration as well, then a larger number of replicas is not necessarily more cost effective. (2) When the replica number is allowed to change dynamically, SmartSLA fuses the infrastructure cost with the system model and makes intelligent decisions on the number of replicas to use in each time interval and it achieves lower cost than *any* of the cases with fixed number of replicas. (3) When the relative infrastructure cost is higher (i.e., c is 3 rather than 1), SmartSLA tends to use fewer replicas (2.72 versus 3.74) and such a result shows a

[2](Nov. 8, 2010) Small DB Instance: 1.7 GB memory, 1 ECU (1 virtual core with 1 ECU), 64-bit platform, Moderate I/O Capacity is $0.11 per hour; Large DB Instance: 7.5 GB memory, 4 ECUs (2 virtual cores with 2 ECUs each), 64-bit platform, High I/O Capacity is $0.44 per hour.

TABLE III
COMPARISON OF COST WITH/WITHOUT SECOND LEVEL CONTROLLER.

Number of replicas	SLA penalty Cost	Infrastructure Cost(Average number of replicas)	Total Cost
$c = 1$			
2	2363	100(2)	2463
3	2154	150(3)	2304
4	2115	200(4)	2315
5	2080	250(5)	2330
SmartSLA	2097	187(3.74)	2284
$c = 3$			
2	2363	300(2)	2663
3	2154	450(3)	2604
4	2114	600(4)	2714
5	2080	750(5)	2830
SmartSLA	2151	408(2.72)	2559

key feature of SmartSLA that the optimal replica number is infrastructure-cost-sensitive.

C. Action Cost

In this subsection, we further investigate a database specific issue in the system and show that by considering action cost properly, SmartSLA can achieve further cost reduction. Regarding database replication related costs, so far we only considered the infrastructure cost, where the cost is determined by the number of replicas. However, *dynamically* changing the replica number also involves additional action cost.

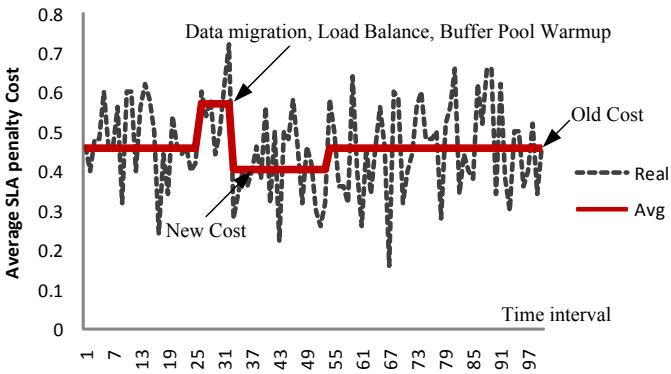


Fig. 11. Adding and removing replicas.

For example, Fig. 11 shows the detailed performance of SmartSLA, by using a static Poisson workload with a typical setting (5 req/sec, 50 CPU shares and 512MB memory), when the number of database replicas changes over time. In order to see the detail more clearly, we zoom in and use a time interval of 20 seconds. The number of replicas used by SmartSLA is 2, up to time interval 24. At the beginning of the 25-th interval, we start an additional replica to bring the total replica number to 3, and at time 55, we stop this additional replica and fall back to the original 2 replicas. In the figure, in addition to the average cost at each time interval, we also show the average over time windows [1,24], [25,29], [30,54], and [55,100]. At time 25, the additional replica is started. However, the average cost distinctively *increases* initially, then falls to a new cost level after time 30. Such an increase in cost is due to data migration. A typical database replication process includes the following steps: (1) stop one of the slaves, (2) copy the data from the slave to the new slave, (3) synchronize the two slaves with the master, (4) start the two slaves, and (5) buffer pool warmup and load balancing. Obviously, because of the suspension of an existing slave in step (1), the system performance suffers initially. The length is about 4 to 5 time intervals for a total of about 80 seconds. Note that this time depends on a number of other factors, such as database size, whether the hypervisor is busy or not, the network bandwidth, and the disk I/O bandwidth from the source slave to the destination slave.

In order to model the action cost involved in adding replicas, we use $\overline{D}$ to denote the semi-reverse direction of D where we have $\overline{d_i} = [d_{i1}, d_{i2}, -d_{i3}]$ as shown in Fig. 12. Compared with D, the first and the second dimensions are the same but the third dimension is opposite in $\overline{D}$. Similarly, we have

$$\overline{SLA}(k) = \sum_{i=1}^{N} \overline{AC}(i,k) \times L(i,k) \times w(i)$$

where

$$\overline{AC}(i,k) = f_i(cpu(i,k-1) + d_{i1}, mem(i,k-1) + d_{i2}, M(k-1) - D_3, L(i,k))$$

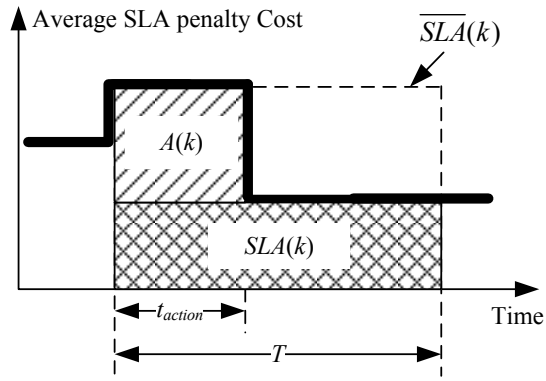


Fig. 12. Modeling action cost for adding replicas.

As depicted in Fig. 12, when the system starts to add a new replica, e.g., decide to follow direction D, we actually follow the semi-reverse direction $\overline{D}$ during the action time t_{action}. And when the replica is ready, we follow D. By carefully studying the behavior, we can model the action cost involved in the adding process as

$$A(k) = \begin{cases} \frac{\alpha t_{\text{action}}}{T}(\overline{SLA}(k) - SLA(k)) & : \text{if } D_3 = 1 \\ 0 & : \text{otherwise} \end{cases}$$

Here we also introduce a parameter α as an amortization factor. The amortization factor indicates the confidence of the the controller in a new replica as a 'future investment". When the amortization factor is low, e.g., when $\alpha = 0$, the action cost will be distributed along the infinity intervals. In this

case, SmartSLA is optimistic about the future intervals and it does not consider the action cost at all. However, when the amortization factor is high, e.g., $\alpha = 1$, SmartSLA is pessimistic about the future intervals and adds a new replica only when the the action cost can be compensated in the next interval.

Compared with adding a replica, stopping an existing replica only needs to wait for the current query request to this replica to finish. In our experiments, the whole process can be done in less than 1 second. Thus stopping an existing replica involves almost no cost and as can be seen from Fig. 11, the average cost jumps almost immediately to its next level.

Based on this study, we refine the problem formulation as follows:

$$
\begin{aligned}
&\textit{next direction} = \arg\min_{D} \textit{SLA}(k) + I(k) + A(k) \\
&\text{s.t.} \quad \textit{SLA}(k) = \sum_{i=1}^{N} \textit{AC}(i,k) \times L(i,k) \times w(i) \\
&\qquad \textit{AC}(i,k) = f_i(\textit{cpu}(i,k) + d_{i1}, \textit{mem}(i,k) + d_{i2}, \\
&\qquad\qquad M(k-1) + D_3, L(i,k)) \\
&\qquad I(k) = I(k-1) + c \times D_3 \\
&\qquad A(k) = \begin{cases} \frac{\alpha t_{\text{action}}}{T}(\overline{\textit{SLA}}(k) - \textit{SLA}(k)) & \text{if } D_3 = 1 \\ 0 & \text{otherwise} \end{cases} \\
&\qquad \sum_{i=1}^{N} d_{i1} = 0, \qquad \sum_{i=1}^{N} d_{i2} = 0 \\
&\qquad D_3 = d_{13} = d_{23} = \ldots = d_{N3} \in \{-1, 0, 1\}
\end{aligned}
$$

D. Evaluation with the Action Cost

Fig. 13 shows the real actions, in terms of replica number over time for the three cases. As shown in the figure, there are 4 replica addition actions when $\alpha = 0$ while there is no action at all when $\alpha = 1$. When $\alpha = 0.1$, a new replica is started only when the workload of the silver clients becomes bursty, and stopped when the burst is gone. Table IV summarizes the performances of SmartSLA under this cost model with α =0, 0.1, and 1. As can be seen, both too aggressive ($\alpha = 0$) and too conservative ($\alpha = 1$) settings give relatively inferior performance, while with $\alpha = 0.1$, the total cost is further reduced.

TABLE IV
COMPARISON OF COST WITH DIFFERENT AMORTIZATION FACTORS.

Amortization factor	SLA penalty Cost	Infrastructure Cost (Average number of replicas)	Total Cost	Number of added replicas
0	2151	408(2.72)	2559	4
0.1	2130	378(2.52)	2508	2
1	2364	300(2)	2664	0

VII. RELATED WORK

The virtual resource management in cloud environments has been studied with goals such as QoS awareness, performance

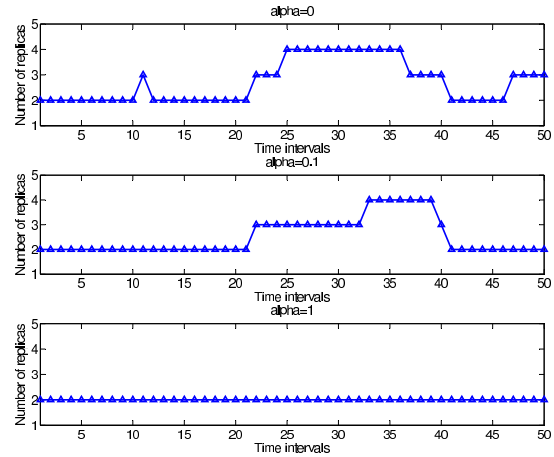


Fig. 13. The number of replicas used over time by SmartSLA under different amortization factors.

isolation and differentiation as well as higher resource utilization. From a general system point of view, most of the related works can be divided into dynamic CPU partitioning, dynamic memory partitioning, dynamic storage partitioning and also dynamic power partitioning.

There are a plethora of works towards optimal CPU and memory partitioning with respect to the performance guarantees. For example, Pradeep et al. develop an adaptive resource control system that dynamically adjusts the resource shares to applications in order to meet application-level QoS goals while achieving high resource utilization in the data center [19]. Urgaonkar et al. [20] present techniques for provisioning CPU and network resources in shared hosting platforms. Most of the early works assumed that the system under control is linear, and that the parameters can be identified offline [21]. Lu et al. dynamically adjust the cache size for multiple request classes [22]. Chou et al. present an algorithm called DBMIN for managing the buffer pool of a relational database management system [23]. If we consider power as another kind of system resource, then it is natural to do dynamic power allocation. Chase et al. propose the importance of managing energy and server resources in hosting centers [24]. Wang et al. controls both power and application level performance separately [25]. Most of the previous works use linear model to design and implement the controller. However, there will be oscillation and the system will be unstable once the operation point moves out of the linear area. For example, as we shown in this paper, there is a significant non-linear relationship between the performance and some of the system metrics. Compared with these previous works, we use machine learning technique to build the relationship based on tree models. One of the benefits of using a tree model is to overcome the non-linear obstacle.

Besides the general system metrics that can be tuned for general systems, there are also lots of special database system intrinsic metrics that can be tuned to improve the performance. For example, Duan et al. [26] tune the parameters of a database in order to get a better database performance. The most important problem is that the search space is huge and the

optimal configuration is hard to find. Ganapathi et al. [27] use a machine learning technique called KCCA to predict metrics such as elapsed time, records used, disk I/Os, etc. Compared with KCCA, we focused on popular and easy-to-use techniques such as linear regression and boosting. Moreover, KCCA is sensitive to some modeling parameters such as the definition of the Euclidean distance and the scale factor. Although well-tuned parameters can give good prediction, bad parameter settings may cause significant degradation in the model's predictive power.

The work presented in Soror et al. [6] is most closely related to ours. There are two significant differences. (1) They model the problem as a service differentiation problem under the resource constraints. However, we model the problem as a two level optimization/control problem. Compared with theirs, we consider the cloud environments where database service provider can enjoy more flexibility to extend their resources. (2) They model the relationship between the performance and the system metrics like CPU and memory individually. However, in our work, we combine the system metrics, the number of replicas, and the arrival rate as the multiple input. Consequently our model is comprehensive to capture various relationships among system metrics and performance. Moreover, we also consider the action cost related to database systems and provide a model for replica tuning.

VIII. Conclusion

In this paper, we investigated the problem of virtual resource management for database systems in cloud environments. We used machine learning techniques to learn a system performance model through a data-driven approach. The model explicitly captures relationships between the systems resources and database performance. Based on the learned predictive model, we designed an intelligent resource management system, SmartSLA. SmartSLA considers many factors in cloud computing environments such as SLA cost, client workload, infrastructure cost, and action cost in a holistic way. SmartSLA achieves optimal resource allocation in a dynamic and intelligent fashion. Experimental studies on benchmark data and real-life workloads demonstrated that such an intelligent resource management system has great potentials in improving the profit margins of cloud service providers.

Acknowledgment

We thank Michael Carey, Hector Garcia-Molina, and Jeffrey Naughton for the insightful discussions and comments. We thank Oliver Po and Wang-Pin Hsiung for their great help with the experimental studies.

References

[1] D. Florescu and D. Kossmann, "Rethinking cost and performance of database systems," *SIGMOD Rec.*, vol. 38, pp. 43–48, June 2009.

[2] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden, "Relational cloud: The case for a database service," *MIT CSAIL Technical Report*, 2010.

[3] H. Hacıgümüş, S. Mehrotra, and B. R. Iyer, "Providing database as a service," in *Proc. of ICDE*, 2002.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, pp. 1277–1288, August 2008.

[5] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-tenant databases for software as a service: schema-mapping techniques," in *Proc. of SIGMOD*, 2008.

[6] A. A. Soror, U. F. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath, "Automatic virtual machine configuration for database workloads," in *Proc. of SIGMOD*, 2008.

[7] F. Yang, J. Shanmugasundaram, and R. Yerneni, "A scalable data platform for a large number of small applications," in *Proc. of CIDR*, 2009.

[8] L. Zhang and D. Ardagna, "Sla based profit optimization in autonomic computing systems," in *Proc. of ICSOC*, 2004.

[9] S. Malkowski, M. Hedwig, D. Jayasinghe, C. Pu, and D. Neumann, "Cloudxplor: a tool for configuration planning in clouds based on empirical data," in *Proc. of SAC*, 2010.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of SOSP*, 2003.

[11] G. Jung, K. R. Joshi, M. A. Hiltunen, R. D. Schlichting, and C. Pu, "A cost-sensitive adaptation engine for server consolidation of multitier applications," in *Proc. of Middleware*, 2009.

[12] P. Xiong, Z. Wang, G. Jung, and C. Pu, "Study on perform-ance management and application behavior in virtualized environment," in *Proc. of NOMS*, 2010.

[13] "Transaction processing performance council. tpc benchmark w (web commerce). number revision 1.8, february 2002."

[14] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. M. Nahum, and A. Wierman, "How to determine a good multi-programming level for external scheduling," in *Proc. of ICDE*, 2006.

[15] R. J. Quinlan, "Learning with continuous classes," in *Proc. of the 5th Australian Joint Conference on Artificial Intelligence*, 1992.

[16] J. Friedman, "Stochastic gradient boosting," 1999.

[17] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," in *Proc. of the 5th Australian Joint Conference on Artificial Intelligence*, vol. 11, no. 1, 2009.

[18] M. Arlitt and T. Jin, "Workload characterization of the 1998 world cup. web site," *HP Tech. Rep.*, 1999.

[19] 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 *Proc. of Eurosys*, 2009.

[20] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," in *Proc. of OSDI*, 2002.

[21] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury, "Mimo control of an apache web server: Modeling and controller design," in *Proc. of 2002 American Control Conference*, Anchorage, Alaska, 2002.

[22] Y. Lu, T. Abdelzaher, and A. Saxena, "Design, implementation, and evaluation of differentiated caching services," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 5, 2004.

[23] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *Proc. of VLDB*, 1985.

[24] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle, "Managing energy and server resources in hosting centers," in *Proc. of SOSP*, 2001.

[25] X. Wang and Y. Wang, "Exploring power-performance tradeoffs in database systems," in *Proc. of ICDE*, 2010.

[26] S. Duan, V. Thummala, and S. Babu, "Tuning database configuration parameters with ituned," *Proc. VLDB Endow.*, vol. 2, pp. 1246–1257, August 2009.

[27] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, O. Fox, and M. Jordan, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in *Proc. of ICDE*, 2009.