Self-Adaptive and Sensitivity-Aware QoS Modeling for the Cloud

Tao Chen
School of Computer Science
University of Birmingham
Birmingham, UK. B15 2TT
txc919@cs.bham.ac.uk

Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham, UK. B15 2TT
r.bahsoon@cs.bham.ac.uk

Abstract*—Given the elasticity, dynamicity and on-demand nature of the cloud, cloud-based applications require dynamic models for Quality of Service (QoS), especially when the sensitivity of QoS tends to fluctuate at runtime. These models can be autonomically used by the cloud-based application to correctly self-adapt its QoS provision. We present a novel dynamic and self-adaptive sensitivity-aware QoS modeling approach, which is fine-grained and grounded on sound machine learning techniques. In particular, we combine symmetric uncertainty with two training techniques: Auto-Regressive Moving Average with eXogenous inputs model (ARMAX) and Artificial Neural Network (ANN) to reach two formulations of the model. We describe a middleware for implementing the approach. We experimentally evaluate the effectiveness of our models using the *RUBiS* benchmark and the FIFA 1998 workload trends. The results show that our modeling approach is effective and the resulting models produce better accuracy when compared with conventional models.

***Index Terms*—QoS modeling, machine learning, sensitivity, interference, prediction, cloud computing.**

I. Introduction

Cloud computing enables dynamic scalability and on demand provision of software and hardware resources, which can be bought or leased. Software-as-a service (SaaS) may support different concrete services with varying QoS requirements. Examples of these QoS may include: response time, throughput, availability, security, and so forth. QoS tends to be sensitive to two dimensions of primitives: these are ***Environmental Primitive (EP)*** and ***Control Primitive (CP)***. We posit that CP can be either software or hardware, which could be managed by cloud providers to support QoS provisioning. In particular, software CPs are software tactics and configurations; such as the number of threads in thread pool and its life time, the number of connections in database connection pool, security and load balancing policies etc. Whereas, hardware CPs are computational resources, such as CPU, memory and bandwidth. These software and hardware CPs are offered at the Platform-as-a Service (PaaS) [25] and Infrastructure-as-a Service (IaaS) [26] fronts respectively. On the other hand, we relate EP to highly dynamic scenarios, which can significantly influence the QoS but the provider can not manage and control their behavior. Examples include unbounded workload and unpredictable bound received data etc. If the provider would be able to control the presence of these scenarios, these can be then considered as CPs.

The primitives, which correlate with a QoS are referred to as *relevant primitives* of such QoS. These relevant primitives are numerous and tend to be various in different cloud scenarios. Existing works (e.g., [1]-[12], [20,21]) however, have only focused on deterministic scenarios, where the consideration of CP is limited and of fixed dimension. Research has specifically looked at hardware CPs at the IaaS font. Little attention has been devoted for linking software CPs at the PaaS layer to QoSs. The PaaS layer facilitates building and deploying cloud-based applications by delivering a software stack, which includes middleware (e.g., Tomcat [27] and JBoss [28]), programming APIs [25], and even a development environment. Such software stack can be configured or influenced by various software CPs. In particular, the provision of software CPs can be dynamically adapted at runtime with the help of management tools, which are used by cloud providers to manipulate the software stack [22]. In addition, these primitives tend to be heterogeneous as they can be customized by the PaaS provider, for instance, they are adjustable in either per-application or per-service basis. Their heterogeneity and fluctuation tend to influence QoSs significantly [22,23]. As a result, modeling QoS should cater for both the software and hardware CPs.

QoS models express the correlation between QoS and changes in its relevant primitives. These models can serve as powerful tools to autonomically assist cloud providers in the adaptation of cloud-based services and applications. They can also assist in determining the extent to which services and applications can sufficiently exploit CPs to support QoS objectives taking into consideration the QoS sensitivity to both EPs and CPs. By sensitivity, we are interested in answering these related questions: (i) *Which* primitives correlate with the QoS provision? (ii) *When* these primitives correlate with QoS? (iii) *How* the uncertainty of QoS provision can be apportioned and be sensitive to these primitives?

For services deployed in highly dynamic, elastic and on-demand environment like cloud, their QoS sensitivity tends to be *dynamic*. However, existing QoS models and their sensitivity are either static [1,5,7] or semi-dynamic [2,3,4,6,8,9,10,11,12,20,21]. In particular, the semi-dynamic models have only focused on dynamic expressions of *how* the primitives correlate with QoS but their treatments for *which* and *when* have been static. The absence of such dynamic support can lead to inaccurate prediction of QoS and inappropriate adaptation of the application. We call for novel dynamic and self-adaptive approach for modeling QoS and its sensitivity. The modeling approach shall be self-aware of QoS sensitivity with respect to *which, when* and *how* primitives correlate with QoS at runtime. Moreover, it shall be self-adaptive to the changes in QoS sensitivity. Specifically, modeling QoS with respect to its dynamic sensitivity in cloud poses several challenges:

QoS granularity: A cloud-based application may be composed of numbers of cloud services, each with its QoSs requirements. Existing QoS models (e.g., [1]-[12], [21]) tend to focus on the mean and aggregate QoS of the entire application. Such coarse-grained analysis suffers from limited sensitivity; it does not apportion sensitivity to changes in QoS of individual services and the primitives. As a result, dynamic adaptation and management of QoS tend to be inaccurate and limited. More accurate and effective adaptation needs to be approached from a fine-grained perspective. Fine-grained modeling for QoS also leads to the concern of QoS interferences.

QoS interferences: Multiple services and their QoSs might be sensitive to the same primitive and thus create chances for QoS interferences. By interference, we refer to scenarios where fluctuation of the primitives can directly (or indirectly) interfere with related services and consequently the correlated QoS. For instance, as the workload of a given service increases, it will trigger a demand for more threads of the service. This implies that the throughput of the said service would be improved because more requests can be processed concurrently. However, such change of workload and demand on thread could interfere with other services running on the same Virtual Machine (VM). This is because the said service tends to consume more of the shared computational resource allocation. Such dynamic QoS interference implies that it is insufficient to model QoS for an individual service in isolation.

Cloud dynamics: Cloud-based applications tend to be dynamic: service composition and deployment strategies can dynamically change at runtime as the requirements and environment change. Such changes can introduce new primitives, phase out certain primitives and affect the demand on CPs etc. As a result, QoS model to its relevant primitives can be changed.

In this paper, we propose a self-adaptive, sensitivity-aware and per-service QoS modeling approach. Our approach is capable to adaptively capture the dynamics of QoS sensitivity by determining *which, when* and *how* primitives correlate with QoS at runtime. The resulting model is able to predict the achieved QoS, given a set of relevant primitives. Such model can assist adaptive systems in determining the sufficient provisions of CPs to achieve certain QoS objectives. To the best of our knowledge, we are the first to consider dynamic sensitivity of QoS with respect to their relevant primitives while simultaneously considering concerns of QoS granularity, QoS interferences and cloud dynamics. In particular, we make the following novel contributions:

Firstly, we abstract the problem of fine-grained QoS and its sensitivity in a dynamic cloud. We propose a per-service model to address QoS granularity and a *relevant primitives matrix* attached to each model to cope with the QoS interferences and cloud dynamics. Unlike previous work, which had mainly considered specific QoSs (e.g., performance), our model is generic and applicable to a wider range of QoS attributes.

Secondly, we use *symmetric uncertainty* [16] to dynamically determine *which* and *when* primitives correlate with QoS. Consequently, we adaptively update the results in the relevant primitives matrix. Symmetric uncertainty is proven to be effective in predicting correlated random variables [16]. Unlike previous work, we explicitly consider the software CPs in conjunction with hardware CPs in QoS modeling.

Thirdly, we combine the abstract QoS model and symmetric uncertainty. We apply machine learning techniques to this combination and we reach two *alternative* formulations of the QoS model. In particular, we explore the potentials of Artificial Neural Network (ANN) and Auto-Regressive Moving Average with eXogenous inputs model (ARMAX) for adaptively revealing *how* primitives correlate with QoS, based on relevant primitives matrix. These two techniques are superior to closed-form models (e.g., queuing network) as they are capable to produce an accurate model without knowing the internal structure of a service and the underlying infrastructure [3]. Their ability to adapt to changing dynamics has been shown to be effective [4,6,9,10].

Fourthly, we describe a middleware, which implements our self-adaptive approach based on feedback control mechanism.

Fifthly, our models are experimentally evaluated on the *RUBiS* [14] benchmark using FIFA 1998 workload trend [24]. Experiments show that our models produce better accuracy when compared with conventional ones. In comparison of the two resulting models, our S-ANN can better handle dynamic QoS sensitivity and produce higher accuracy when the fluctuation of measured QoS increases, whereas our S-ARMAX produces less error when such fluctuation decreases.

The paper is structured as follows. Section II motivates and describes the assumption and design decisions of our models. Section III describes the self-adaptive and sensitivity-aware QoS modeling approach. Section IV reports on the evaluation and the experimentation results. Section V and VI discuss related work and present the conclusion respectively.

II. Models

In this section, we present our assumptions, the system model and abstract QoS model that drive our design.

A. System Model

We assume that a cloud-based application is formed of one or more independent or composable services. These applications are hosted on the cloud infrastructure where resources are shared via VMs. It is possible to host multiple applications on the same VM. However, in this work we assume one-to-one relationship between an application instance and a VM instance. In distributed environment like cloud, an application, composed of concrete services $\{S_1, S_2, \dots S_i\}$ may have multiple replicas deployed to different VMs. A replica of an application running on a VM is assumed to have its services replicas running on the same VM. In this work, we refer to the replicas of concrete services as service-instances: the *jth* service-instance of the *ith* concrete service is denoted by S_{ij}. Unlike previous work [3,4,6], which focus on homogenous scenario, we look at heterogeneous cases where different software stacks, VMs and Physical Machines (PMs) in the cloud could have various types of primitives and capacity; henceforth, the relevant primitives of QoSs for different service-instances tends to be heterogeneous. To cope with such heterogeneity, we aim to adaptively create QoS models in relation to each service-instance. In such context, service-instances on a VM use the same computational resource; service-instances may functionally depend on instances running on other VMs. These facts imply that their QoSs could be sensitive to the same primitives causing likely QoS interferences. Often, the virtualization techniques in cloud allow service-instances to be redeployed on the fly. As a scenario expressing an example of cloud dynamics, the database service can be live-migrated to another VM or PM. This situation results in fluctuation of QoS sensitivity.

B. QoS Model

The common practice of machine learning techniques for modeling QoS in the cloud is to apply the provision value of the relevant CPs as training inputs [3,4,6,9,10,11,12]. By provision value, we refer to the allocation bound of CP. However, it is generally impossible to guarantee that a given QoS can always be achieved by fully utilizing the allocated CPs. Such fact obfuscates the sensitivity of QoS to its primitives as using the provision values to train QoS model would take those idle proportions of provisions into account. As a result, using provision values of CPs as inputs is ill-suited in our case. To cope with this issue, we apply the actual demands of CPs (e.g., real-time usage of CPU) as inputs, which better reveal QoS sensitivity. Moreover, training the model with CP demands implies that our model is likely to determine the minimal requirement of CP provisions for achieving certain QoS objectives. This will potentially improve the elasticity of CPs provision in cloud when our modeling approach is used in adaptation. Note that certain CP types (e.g., thread) can be partitioned to each service-instance , whereas others (e.g., CPU and memory) are non-partitionable. If a CP type is non-partitionable, the demands of the shared service-instances are measured as an identical input. Otherwise, the partitions of such CP type are seen as distinct inputs.

To model the QoS and its sensitivity, we first formulate a QoS model, which abstracts the dynamic sensitivity of QoS with respect to its relevant primitives over time. The model determining the *kth* QoS of a service-instance S_{ij} at sampling interval *t* can be formally expressed as:

$$QoS_k^{ij}(t)=f(SP_k^{ij}(t),\delta) \tag{1}$$

where $QoS_k^{ij}(t)$ is the average value of *kth* QoS of S_{ij} at interval *t*. *f* is the QoS function, which subjects to dynamic changes. δ refers to any other inputs that are required by the technique to train *f* apart from the primitives. To cope with QoS interferences and cloud dynamics, we denote the input $SP_k^{ij}(t)$ as the relevant primitives matrix of $QoS_k^{ij}(t)$ at interval *t*. The column entries of this matrix are relevant primitives that indicate *which* primitives correlate with the QoS. The number of row entries reveals how many historical values of these primitives could impact the accuracy of QoS model. Formally, the relevant primitives matrix is expressed in Eq. 2:

$$SP_k^{ij}(t)=\begin{pmatrix} CP_a^{xy}(t) & \dots & EP_b^{mn}(t-1) & \dots \\ \dots & \dots & \dots & \dots \\ CP_a^{xy}(t-q+1) & \dots & EP_b^{mn}(t-q) & \dots \end{pmatrix} \tag{2}$$

where q is the number of order, which determines the number of row entry; its value depends on the technique that trains *f* (see Section III-B and III-C). $CP_a^{xy}(t)$ and $EP_b^{mn}(t-1)$ denote average demand of the *ath* CP of S_{xy} at *t* and average value of the *bth* EP of S_{mn} at *t-1* respectively. The column entries in $SP_k^{ij}(t)$ are selected from the primitives associated with S_{ij} and other correlated service-instances. In particular, a primitive of a service-instance is included as a column in such matrix only if information of the said primitive correlates with that of $QoS_k^{ij}(t)$ (see Section III-A). Due to the dynamicity of QoS sensitivity, $SP_k^{ij}(t)$ may be subject to change over time. When $SP_k^{ij}(t)$ has non-partitionable CP types, the redundant column entries should be merged as they refer to the same input.

Based on Eq. 2, to predict QoS at interval *t,* the latest input for CP is the demand at interval *t*, whereas the latest value for EP is the data at *t-1.* This is because at interval *t-1,* one could apply the model to determine the minimal required CP provisions for the next interval *t* in a proactive manner. However in such case, we can not sense or control any EP values for the future interval *t* as the EP is uncontrollable and unmanageable. Therefore the newest possible EP value is the one at interval *t-1.* Previous EPs (up to interval *t-1)* could still contribute to the uncertainty of future QoS at interval *t* since the measured data follow continuous time series.

All the inputs and output in Eq. 1 and Eq. 2 are normalized values, which are calculated as the ratio between the value at current interval and the biggest ever value through the entire time series. This is because the original values have arbitrary magnitude, which obfuscates the sensitivity of output to inputs and hence affecting the accuracy. Conversely, the normalized values improve numeric stability as they range from 0 to 1.

At the end, for a given $QoS_k^{ij}(t)$, the ultimate goal of our self-adaptive and sensitivity-aware QoS modeling is to: firstly adaptively determine the content of $SP_k^{ij}(t)$ and secondly, adaptively train function *f*.

III. Design and Implementation of the Modeling Approach

To adaptively build a fine-grained, sensitivity-aware QoS models, we implemented our modeling approach as a middleware. As shown in Fig. 1, the QoS modeling process is realized as decentralized feedback loops; in particular, a dedicated *middleware instance* is attached to each VM, and it could be deployed on the root domain of a PM (e.g., Dom0 of the hypervisor Xen [15]).

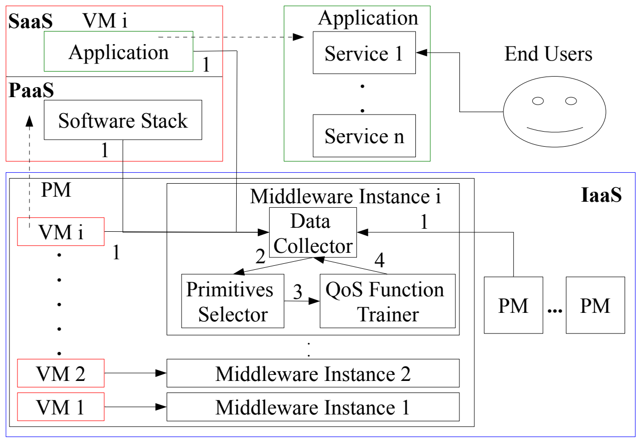


Fig. 1. Self-adaptive and sensitivity-aware QoS modeling middleware

Our middleware encapsulates three components: *data collector, primitives selector* and *QoS function trainer*. In Fig. 1, each *Middleware Instance i* is attached to *VM i.* The supported QoS attributes and primitive types are provided by the cloud administrators. *Middleware Instance i* monitors the data of each service-instance on *VM i* across SaaS, PaaS and IaaS layers in the cloud via cloud providers' own measurement facilities, while adaptively producing QoS models based on this data. More precisely, the *data collector* continuously senses QoS values, EP values and demand of CPs from all service-instances on its corresponding VM. It is also responsible for recording all historical data (step 1). The *data collectors* may need to collect data from the external service-instances, which could be on other VM/PM. Because these external service-instances may be functionally required by the service-instances running on the VM attached to *data collectors.* All historical data is then passed to our *primitives selector* to determine *which* and *when* primitives are correlated with a QoS (step 2). Once the relevant primitives are selected for each QoS, the *QoS function trainer* can apply the data set to dynamically train *how* these primitives correlate with QoS and produce the final QoS models, based on machine learning techniques (step 3). To capture dynamic sensitivity of QoS, the entire process should be repeated periodically (step 1-4). The frequency of this repetition is discussed in Section IV-E.

A. Primitives Selector

As shown in Eq. 1 and Eq. 2, the first task for modeling $QoS_k^{ij}(t)$ is to explore *which* primitives should be included as column entries in $SP_k^{ij}(t)$, and determine *when* is the appropriate interval to consider these primitives. In the *primitives selector*, we apply *symmetric uncertainty* [16] to determine *which* and *when* primitives correlate with QoS. The symmetric uncertainty represents mutual dependency between two random variables. Symmetric uncertainty between all historically measured data of a QoS and that of a primitive results in 1, indicating that all information of the primitive is correlated with the QoS (and vice versa). At the other extreme, symmetric uncertainty value of 0 implies that changes in the primitive's behavior are independent of that of the QoS. Formally, symmetric uncertainty *U(X,Y)* is calculated as:

$$U(X,Y)=\frac{2\times I(X,Y)}{H(X)+H(Y)} \tag{3}$$

$$I(X,Y)=\sum_{y\in Y}\sum_{x\in X} p(x,y)\log\left(\frac{p(x,y)}{p(x)\times p(y)}\right) \tag{4}$$

$$H(X)=\sum_{x\in X} p(x)\log(p(x)) \tag{5}$$

where X and Y are collections of all historical data of a QoS and a primitive. Eq. 4 shows the formula for mutual information and Eq. 5 expresses entropy. p(x,y) denotes the joint probability function between a concrete value of QoS *x* and a concrete value of primitive *y* at a specific interval. p(x) and p(y) are the marginal probability function of *x* and *y* respectively.

Precisely, the *primitive selector* workflow is as follows: firstly, based on the sensed data, it calculates the symmetric uncertainty between an individual QoS of a given service-instance and the primitives, which are likely to be correlated with this QoS. Secondly, it updates the corresponding relevant primitives matrix by adding the primitives, which result in nonzero value; while removing primitives that have zero value.

For each QoS of a service-instance S_{ij}, the likely relevant primitives are selected from two sets of service-instances: the first set includes all the service-instances on the same VM that S_{ij} belongs to (including S_{ij} itself). The second set are all service-instances that directly and functionally required by S_{ij}. We observe that the primitives of these service-instances are most likely to result in nonzero symmetric uncertainty values with S_{ij}'s QoSs (the scope of likely relevant primitives can be easily changed). To handle dynamicity, the relevant primitives matrix can be continuously updated with newly-measured data.

B. Sensitivity-Aware ARMAX Model

Recall from the Eq. 1, once the $SP_k^{ij}(t)$ is defined by the *primitives selector*, our next goal of QoS modeling is to determine *how* those primitives correlate with $QoS_k^{ij}(t)$ by dynamically building the function *f*. To achieve such goal, we apply two *alternative* techniques in *QoS function trainer*. One of these techniques, which we discuss in this section, is the linear Auto-Regressive Moving Average with eXogenous inputs model (ARMAX) [19]. This model particularly fits our case because it is based on continuous time series. The correlation between primitives and QoSs in our case is unlikely to be linear, however, the behavior of a service instance can be approximated locally as a linear model [9]. We adopt ARMAX such that the output is the QoS; inputs are historical QoS values and relevant primitives of the said QoS. Formally, based on the abstraction of QoS model and symmetric uncertainty, our Sensitivity-aware ARMAX (S-ARMAX) is defined as:

$$\begin{aligned} QoS_k^{ij}(t) &= \sum_{z=1}^{q} \alpha_z(t)\times QoS_k^{ij}(t-z) \\ &+\sum_{z=1}^{q}\sum_{a=1} \beta_{za}(t)\times CP_a^{xy}(t+1-z)+\sum_{z=1}^{q}\sum_{b=1}\theta_{zb}(t)\times EP_b^{mn}(t-z) \end{aligned} \tag{6}$$

$$\text{subject to} \qquad CP_a^{xy}(t+1-z),\, EP_b^{mn}(t-z)\in SP_k^{ij}(t) \tag{7}$$

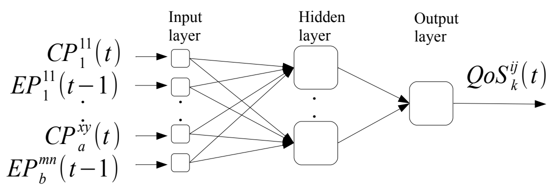
where q is the number of order, $\alpha_z(t)$, $\beta_{za}(t)$ and $\theta_{zb}(t)$ are the coefficients of corresponding QoS values and relevant primitives at sampling interval t. Constraint Eq. 7 ensures that any primitives should be selected from $SP_k^{ij}(t)$, which is determined by Eq. 3.

We train the S-ARMAX model using linear Least Mean Square (LMS) approach [17]. The number of order q determines how many historical intervals should be considered, which influences the accuracy of ARMAX model. The optimal number of order depends on the column entries in $SP_k^{ij}(t)$ and significant diversity of measured data, in our case, data and $SP_k^{ij}(t)$ are both fluctuated over time. To cope with this problem, before training function f, we additionally adopt a step-wise algorithm to dynamically find the suitable q for each structure of column entries in $SP_k^{ij}(t)$ and significant diversity. To validate the potential accuracy of prediction, we apply 2-fold cross-validation [29], which divides the measured data into two continuous and equivalent portions; one for training and the second one for validating. We used 2-fold because the decision of selecting the folds has been primarily driven by the computation demands versus the likely accuracy. We have observed that by increasing the number of folds, it is very likely to significantly increase the computation of step-wise algorithm without gaining significant accuracy improvement. Specifically, the step-wise algorithm works as follow: whenever the column entries in $SP_k^{ij}(t)$ change (as when compared to $SP_k^{ij}(t-1)$) or the accuracy of current model is lower than certain threshold, the step-wise algorithm trains f with incremental q starts from 1 using the first half of measured data, whereas the accuracy of model is assessed via the rest half. The suitable number of order is selected when the next order does not offer better accuracy. Finally, the actual function f can be trained against all the measured data with previously defined q.

By the end of each interval, all coefficients can be updated with the newly-measured and normalized data. Therefore S-ARMAX is capable to adaptively handle the dynamic proportion of each relevant primitive in the uncertainty of QoS.

C. Sensitivity-Aware ANN Model

Artificial Neural Network (ANN) [13] is applied as the second technique to build our function f. A detailed explanation of ANN can be found in [13]. We chose this model because it is capable for modeling complex nonlinear correlations [4]. In particular, we adopt ANN with one hidden layer. This is because we observed in our experiments that using two or more hidden layers could exacerbate the problem of local minima, which significantly increases the training time. More precisely, the ANN, which we adopt is a single-output, feedforward and fully connected three layered network, where the inputs are the relevant primitives determined by *primitive selector* and output is the corresponding QoS. Sigmoid function is chosen as the activation function on each neuron node. We found that the number of order q does not influence ANN's accuracy significantly, therefore we simply set q as 1. Based on the abstraction of QoS model and symmetric uncertainty, our Sensitivity-aware ANN (S-ANN) model is expressed as:



(8)

$$\text{subject to} \quad CP_a^{xy}(t), EP_b^{mn}(t-1) \in SP_k^{ij}(t) \tag{9}$$

Constraint Eq. 9 again ensures that any primitives should be selected from $SP_k^{ij}(t)$. ANN model can be trained with arbitrary quality, which reveals the potential accuracy of the model prediction. By model quality, we refer to the degree to which the model is fit with respect to the training data. In this perspective, a good quality model means that the fitness should not be too low or too high; otherwise, the model will suffer from under- and over-fitting. To guarantee model quality, we define minimum and maximum thresholds to represent the 'good enough' quality of model. The resulting model should be re-trained immediately if its quality is not good enough.

Similar to S-ARMAX, by the end of each interval, the weights in S-ANN can be retrained with the newly-measured and normalized data. To achieve this goal, we apply the RPROP [18] as the actual training algorithm for the network. This is because RPROP can efficiently reach 'good enough' model quality. To avoid training forever, we have defined a training time threshold such that if this threshold has been reached, the training is concluded with the best ever model.

Although ANN is not influenced by the number of order q significantly, its prediction accuracy and likelihood of under- and over-fitting is affected by the number of hidden neurons in hidden layer. The number of hidden neurons is directly related to the extent to which we can capture sensitivity between inputs and output. As a result, insufficient number of neurons could easily result in under-fitting of the model. Conversely, over-fitting is very likely to occur when the importance of relations between inputs and output are amplified. The proper number of hidden neurons is related to the changes in $SP_k^{ij}(t)$ and significant diversity of measured data. Therefore similar to S-ARMAX, before training the actual function f, we apply a step-wise algorithm to dynamically determine the suitable number of hidden neurons for each network. To assess the potential accuracy of prediction, we apply the aforementioned 2-fold cross-validation. Concretely, the algorithm works as follow: once the column entries in $SP_k^{ij}(t)$ change (as when compared to $SP_k^{ij}(t-1)$) or the accuracy of current model is lower than certain threshold, the algorithm increases the number of hidden neuron gradually (starting from 1) and trains the new network with the new setup using half of the measured data. When the next network setup could not produce better accuracy (validated against the rest half data), the algorithm then examine whether the current network setup could train 'good enough' model within the training time threshold, against all the measured data. If it fails to do so, the hidden neuron number would continuously be increased; otherwise, the algorithm terminates. Eventually, we pick the least number of hidden neurons that results in the best accuracy. At the end, the training of *f* and validation of quality can be done against all measured data with the discovered number of hidden neurons.

IV. Experiments and Evaluation

The primary intention of our experiments is to evaluate accuracy and effectiveness of the proposed QoS modeling approach with respect to the scale of data. Specifically, we compare the accuracy of the proposed S-ARMAX and S-ANN models with conventional models in continuous time series. We also assess the sensitivity of our models to the size of training data. Finally, we examine training efficiency by looking at the training overhead.

Our experiments use *RUBiS* [14] benchmark, which is an online auction application. *RUBiS* defines 26 different services e.g., *BrowseCategory*, *ViewItems* etc. Simulation of the artificial workload is based on workload patterns, which define the probability of an end-client to go from a service to another. *RUBiS* offers two categories of workload patterns: the browsing pattern assumes read-only services whereas the biding pattern simulates both read and write intensive services. These patterns help us simulate QoS interference because the request rates vary among service-instances. Instead of using random and arbitrary workload trend to simulate the number of clients interacting with *RUBiS* services, we vary the number of clients proportionally according to the FIFA 1998 trend [24]. We compress the FIFA 1998 trend in the way that the fluctuation of one day in the trend corresponds to 200 secs in our experiment. The use of FIFA workload provides ready sample for simulating scenarios related to service invocations.

The testbed consists of three PMs forming a min-cloud. One of them runs the client emulator, while the rest PMs are used for hosting VMs where the *RUBiS* application and its replicas are deployed on. Each PM is an Intel Core i7 2.80GHz, 4 GB RAM computer with Gigabit Ethernet connected to the switch. All machines run Linux kernel 2.6.16.29. The hosting machine runs Xen v3.0.3 [15]. We use Apache v2.0.54 as the web server, Tomcat v6.0.28 as the application server and MySQL v3.23.58 as the database server. We implemented our middleware using Java, JDK 1.6. In particular, the training of S-ARMAX is based on the Apache Mathematics Library [30] and the S-ANN is trained via Encog Framework [31]. To avoid dependency with hardware, we optimize the training performance of the model by enabling Encog's multi-threaded training feature instead of using its support for GPU. Our middleware is running on Dom0 of each PM. To eliminate interference caused by model training, we allocated 25% CPU and 1 GB memory to Dom0, which tends to be sufficient.

To simulate cloud dynamics, we apply two deployment strategies for the benchmark. The first strategy *D1* assumes that all services of the application are hosted on one VM. The second strategy *D2* involves two VMs for each application replica, where the database server and web/application server are deployed on different VMs. To facilitate the dynamic deployment in the cloud, we switch the deployment from *D1* to *D2* on the fly by live-migrating the database service. To further verify our modeling approach under unusual workload changes, we apply the biding workload pattern for *D1* whereas the browsing pattern is used for *D2*. The entire FIFA 98 workload trend is used for *D1*, and it is repeated again for *D2*. Due to limited space, we only report on evaluation of the QoS models for one service-instance of a concrete service named *SearchByCategory*.

A. Examined QoSs and Primitives

For the simplicity of exposition, we report on a scenario, which considers the following dimensions: three QoS attributes, two hardware CPs, one software CP and one EP. Based on the real-life workload and benchmark, this setup sufficiently provides us with valuable insight on the models' behavior when handling a large stream of live data in complex and dynamic systems.

Concretely, the three QoS attributes, which we monitor and predict are availability, response time and throughput. To simulate availability, we assign a waiting time threshold on client emulator. That is, the service is considered as unavailable for the waiting period if the client does not receive response within the waiting time. Throughput is measured by calculating the rate of completed requests. We denote response time as the leaped time between services receiving a request and a response being sent out. All of these QoSs are measured in average value of an interval for each service-instance. We have implemented the measurement toolkit in Java.

The primitive inputs we considered are two hardware CPs: CPU and memory; one software CP: number of threads (in Tomcat) and one EP: workload. To better simulate the heterogeneity of software CPs, we modify the original configuration named *maxThreads* in Tomcat to a per-service basis. In other words, it is possible to restrict the number of threads per service-instance. In addition, we measure workload as requests rate per-service. We monitor VM's CPU and memory demand via Xen, whereas the threads demand and workload of each service-instance are measured by the management facility provided by Tomcat.

To sum up, there are three QoS models that need to be adaptively created for each service-instance. The actual inputs to train and predict these three QoSs are the primitives in their relevant primitives matrix, which are adaptively determined by the primitive selector and the used machine learning techniques as explained in Section III-B and III-C. In the case of our experiments, the likely relevant primitives of each QoS would be selected from: CPU and memory demands of the web/application server VM; CPU and memory demands of the database server VM (this is only available when switch to *D2*); threads demands and workload for each of the 26 service-instances on the same VM. To compare S-ANN and S-ARMAX, the *QoS function trainer* would simultaneously produce two alternative models for each QoS.

B. Continuous Accuracy

To validate the correctness, we measure the accuracy of our QoS modeling approach on the fly. The sampling interval is 30 secs with the total of 700 intervals. In particular, we examine the accuracy of one interval ahead prediction. That is, by the end of interval *t*, our middleware trains QoS models based on

Table. 1. Comparative summary of QoS prediction accuracy for a service-instance of SearchByCategory

QoS	SMAPE of Prediction (%)						RSD of Measured QoS Trend (%)
	S-ANN	*C-ANN*		*S-ARMAX*	*C-ARMAX*		
		per-service	*per-application*		*per-service*	*per-application*	
Response Time	**6.97**	12.76	31.72	**11.43**	15.08	34.03	120.61
Throughput	**11.14**	16.88	35.28	**7.99**	13.22	37.82	86.41
Availability	**0.96**	0.38	1.36	**0.01**	0.01	1	2.24

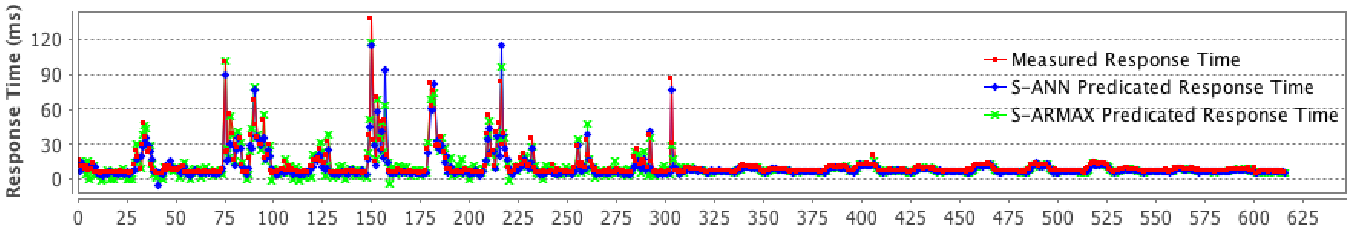


Fig. 2. Actual and predicated response time (x-axis is sampling interval)

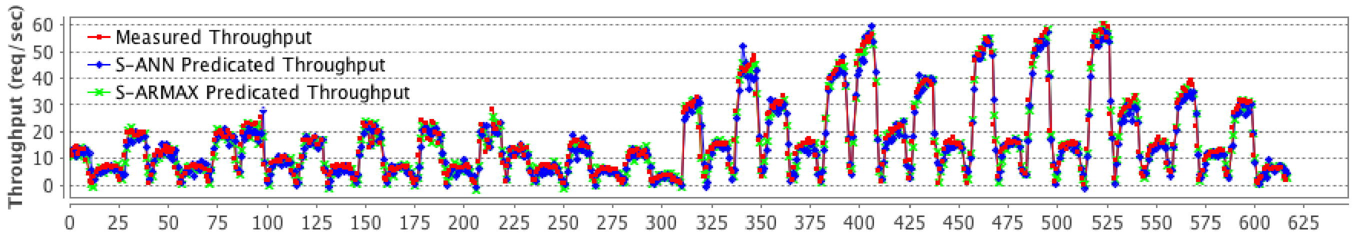


Fig. 3. Actual and predicated throughput (x-axis is sampling interval)

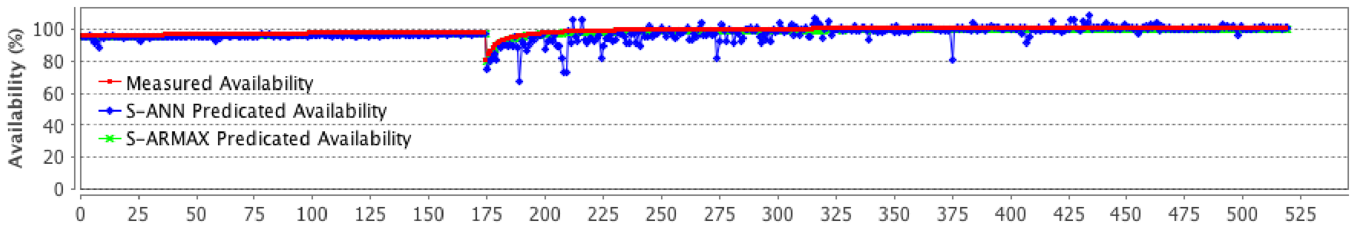


Fig. 4. Actual and predicated availability (x-axis is sampling interval)

historical data up to *t-1*, the resulting model predicts the QoS at *t* by using historical QoS values (for S-ARMAX), the measured demands of CPs up to current interval *t* and value of EP up to interval *t-1*. For all predictions, the accuracy is assessed via Symmetric Mean Average Percentage Error (SMAPE), which is computed as:

$$SMAPE = \frac{1}{K}\sum_{t=1}^{K}\frac{\left|QoS_k^{ij}(t)' - QoS_k^{ij}(t)\right|}{QoS_k^{ij}(t)' + QoS_k^{ij}(t)} \qquad (10)$$

where K is the total number of intervals, $QoS_k^{ij}(t)'$ denotes the measurement of the *kth* QoS of S_{ij} at interval *t* whereas $QoS_k^{ij}(t)$ denotes the prediction for the same QoS at the same interval. Note that we regard zero value of QoS as invalid measurement, because it only represent the fact that no one has requested a certain service at a point in time.

To further evaluate the improvement to conventional semi-dynamic approaches, we compare the accuracy of our S-ARMAX and S-ANN models against the conventional ARMAX [e.g., 9,10] and ANN [e.g., 3,4,6] based models, which only consider limited and fixed hardware CPs such as the CPU and memory of the web/application server VM. In addition, they do not cater for QoS interference and cloud dynamics. We denote these conventional models as ***C-ARMAX*** and ***C-ANN***. Because these models rely on fixed number of relevant primitives, their number of order and hidden neurons are fixed and are obtained by examining given set of measured data (we discovered that in our case set *q* as 2 for C-ARMAX and 18 hidden neurons for C-ANN could result in the best model). These conventional models predict QoS on per-application basis, whereas our models are per-service models. Thereby to eliminate noise caused by granularity, we also compare our models with modified, per-service C-ARMAX and C-ANN. To analyze the correlation between model accuracy and the variation of measured QoS trend, we apply Relative Standard Deviation (RSD) to measure how fluctuation of the QoS tends to be in a relative manner, such metric is calculated as: RSD = σ / μ , where σ is the standard deviation and μ is the mean of all measured QoS values.

The accuracies of all the comparative models are summarized in Table 1. It clearly indicates that our S-ANN reduces the error from 31.72% to 6.97% for response time; from 35.28% to 11.14% for throughput and from 1.36% to 0.96% for availability, when compared to per-application C-ANN model. In contrast to per-service C-ANN, the S-ANN also reduces 5.79% error (12.76% to 6.97%) for response time and 5.74% error (16.88% to 11.14%) for throughput. The only exception is that the S-ANN tends to produce 0.58% (0.38% to 0.96%) higher error for availability. We believe that this is because the RSD of availability is relatively small and thus the influence caused by dynamic QoS sensitivity tends to be trivial, which could easily cause over-fitting. On the other hand, our S-ARMAX is superior to both per-application and per-service C-ARMAX models. In particular, S-ARMAX reduces the error from 34.03% to 11.43% for response time; from 37.82% to 7.99% for throughput and from 1% to 0.01% for availability, when compared to the per-application C-ARMAX model. In contrast to per-service C-ARMAX, the S-ARMAX also reduces 4.45% error (15.08% to 11.43%) for response time and 5.23% error (13.22% to 7.99%) for throughput. The prediction error for availability remains the same. To conclude, it is clear that both of our S-ANN and S-ARMAX offers better accuracy than the C-ANN and C-ARMAX models.

An interesting discovery is that, when we compare our S-ANN and S-ARMAX models, the results reveal that S-ARMAX reduces 0.95% error (0.96% to 0.01%) for availability, which is nearly 99% of S-ANN. It also produces 3.15% less error (11.14% to 7.99%) for throughput, which is around 28% of S-ANN. However, S-ANN tends to be better on response time by reducing 4.46% error (11.43% to 6.97%); such improvement is around 39% of S-ARMAX. We can also see from Table 1 that the RSD for availability, throughput and response time are 2.24%, 86.41% and 120.61% respectively. These observations indicate that nonlinear model like S-ANN handles the dynamic QoS sensitivity better when the fluctuation of measured QoS increases, whereas the linear S-ARMAX produces less error when such fluctuation decreases. This is a useful conclusion as it implies that to better handle the dynamic QoS sensitivity, we shall also adaptively determine the best techniques to train QoS function *f* at runtime.

To provide more detailed view of accuracy when using the proposed modeling approach, Fig. 2-4 illustrate the total of 616 valid measurements of the actual QoS and predicated values produced by S-ANN and S-ARMAX. More precisely, Fig. 2 demonstrates the trends for response time. Although the figure shows that error tends to increase for some of the peak points, it is obvious that both models still produce good prediction even for *D1* (from interval 1 to 310), where the QoS trend highly fluctuates. Particularly, S-ANN performs better in terms of predicting change points. On the other hands, both models obtain excellent fit after interval 310 when we switch to *D2*. Similar observation occurs in Fig. 3, which illustrates the trends for throughput. In particular, across all the intervals, our models have good prediction even at the peak and trough for both *D1* (from interval 1 to 310) and *D2* (interval 310 onwards). In addition, predictions for change points in the curve are acceptable. The trends for availability are shown in Fig. 4. Note that the entire trends only have 525 intervals; this is because the availability has constant value (100%) before the first unavailable data point. As a result, our *primitives selector* identifies that no primitives correlate with such QoS as no fluctuation occurs. Figure 4 clearly indicates that both models predict well for *D1* (from interval 1 to 175). However, after we switch to *D2* at interval 175, the S-ANN's prediction tends to be worse than S-ARMAX. This is because the availability trend has extremely tiny fluctuation for *D1,* while the cloud dynamics scenario, which we are simulating hugely affect availability due to continuous migration of database services. We observe that when training the ANN model with stable data (e.g., interval 1-174) followed by sudden fluctuation (at interval 175) can easily cause over-fitting. This eventually influences the ANN's prediction accuracy. Nevertheless, we can see that the S-ANN is adaptive enough to correct itself; the prediction becomes better and more stable from interval 230. On the other hand, S-ARMAX obtains perfect prediction fit for availability in all the intervals.

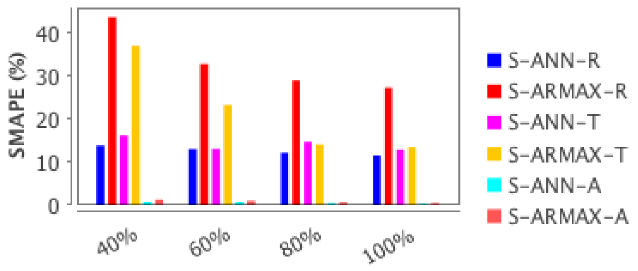


Fig. 5. Sensitivity of accuracy to training data size (x-axis is data %)

C. Sensitivity to Training Data Size

To understand the sensitivity of model accuracy to the training data size, we divide all the measured intervals into 70% as training data and the rest 30% as testing data. All the considered inputs and outputs remain the same as those described in section IV-A. Both S-ANN and S-ARMAX are trained based on a portion of the training data, ranging from 40% to 100%; symmetric uncertainty is applied on each portion of the training data to determine the relevant primitives matrix. Finally, we use the resulting models to predict the whole testing data and record their SMAPE. As shown in Fig. 5, the accuracy of S-ARMAX models for predicting throughput (S-ARMAX-T) and response time (S-ARMAX-R) improve dramatically as the size of training data increases: from 44% to 27% for response time and 37% to 14% for throughput. However, they suffer from relatively high error when there is not sufficient training data. In contrast, the S-ANN models for throughput (S-ANN-T) and response time (S-ANN-R) improve their accuracy gradually: from 14% to 12% for response time and 16% to 13% for throughput. Such improvement is less significant, however. We conclude that S-ANN is less sensitive to the training data size. The accuracy of both models for availability (S-ANN-A and S-ARMAX-A) records similar observations: for S-ARMAX is from 1.3% to 0.5% and for S-ANN is from 0.8% to 0.4%. Ultimately, it is evident that both models improve their accuracy as the size of training data increase. In particular, S-ANN produces acceptable accuracy even under limited data.

D. Efficiency

Finally, we examine the efficiency to train our sensitivity-aware QoS models with all the training data set. We observed that the time taken to calculate systematic uncertainty of a service-instance is rather trivial in our case. The assessed model training time is calculated as the average of 20 trainings against the *response time* data in all intervals. To better evaluate the efficiency for both S-ANN and S-ARMAX, we distinguish two cases: (i) the training time when changes occur in relevant primitives matrix; this requires extra computation in finding the proper number of order and hidden neuron, and (ii) the case without such changes. From Fig. 6, we can clearly see that the training of S-ANN with changes in the matrix produces relatively large overhead (5.48 secs). This is due to the complexity of ANN where the most appropriate number of hidden neuron tends to be high and varying in the 20 trainings. Nevertheless, the trainings with a known number of hidden neuron can be completed in around 0.334 sec. On the other hand, the training of S-ARMAX with change in matrix can be completed in 0.113 sec; the overhead is marginal and almost similar to training without such change. This is because the structure of ARMAX is simpler. In addition, we observed that in most of the 20 trainings, setting the number of order as small as 2 could result in the best accuracy. The S-ARMAX can be trained within around 0.112 sec with known number of order. To conclude, both models have training overhead less than 10 seconds. In particular, S-ARMAX is relatively more efficient in all cases. Therefore, the training overhead of our models is negligible within the sampling interval of 30 secs.

E. Discussion

Our modeling approach runs periodically in order to capture dynamic QoS sensitivity. Determine the frequency of modeling is a complex task. If the modeling is too frequent, this may entail large demand on resources for computing the model. In addition, model training may not be completed within its interval. In contrast, too low frequency may fail to capture the actual and evolving diversity of QoS. Consequently, arriving on the right frequency encompasses a trade-off between efficiency and accuracy. In our experiments, we have analyzed the workload characteristics and have empirically decided on the frequency level. We believe that the same approach can be applied in a practical scenario, where each of the *middleware instances* can have its own frequency for modeling, as long as their responsible service-instances do not functionally interact with each others.

We observed that when Dom0 suffers contention, the performance of our approach could become worse. In our experiment, we have eliminated this by determining the proper amount of provision for Dom0 offline. In the real-world cases, it is still possible to follow the same approach. More precisely, the cloud provider can specify the required computational resources for Dom0 in relation to the total number of service-instances on each PM type. This can be achieved offline by running dummy applications or using historical data. Such decision may influence the VM to PM allocation strategy, which is out of the scope of this work.

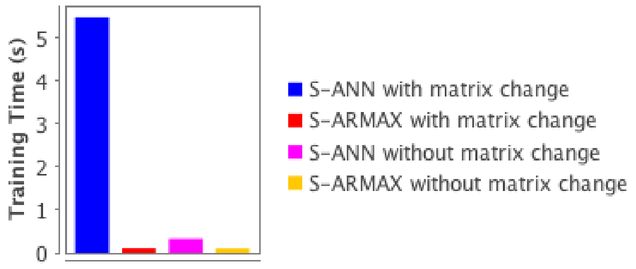


Fig. 6. Training time of S-ANN and SARMAX

Another worth observation to mention is that the historical data in *data collector* could become extremely large as time goes by. Therefore a “forgotten strategy” is desired when there is no need to take too much data into account. To achieve such goal, one could set a threshold to the maximum number of historical intervals to be recorded. Once such threshold is exceeded, the *QoS function trainer* can apply cross-validation to examine if dropping data from the oldest intervals would affect the model accuracy. If the reduction in accuracy is less than 1% error then such data can be removed.

V. Related Work

Emeakaroha et al. [1] propose a framework to manage QoS in the cloud. Their QoS models are static expressions, which handle predefined rules for managing QoS in isolation. Queuing theory has been widely applied for QoS modeling (e.g., [5] and [7]). [7] describe a two-step approach to decompose QoS into hardware CPs. Firstly a profiling mechanism is applied to estimate resource demand for each application. In the next step, a multi-station queuing network is used to analyze the correlation between demand and performance related QoS. Their work is not cloud related however. [5] is cloud specific and focus on analyzing QoS model for each tier of an application using queuing network. All of the aforementioned approaches are static, closed-form QoS models and their effectiveness is restricted to the assumptions of system's internal operations. Their static nature makes these approaches limited in being adaptive and in coping with the dynamic sensitivity of QoS in the cloud.

Feedback control is applied in a semi-dynamic way to address the problem of QoS modeling. Diao et al. [8] present a Multiple-Input and Multiple-Output (MIMO) feedback based approach, which has focused on runtime performance modeling on how it correlates with the primitives. The research discussed in [2] focuses on linear MIMO modeling of performance in the cloud. These approaches consider VM-level interference whereas our approach takes dynamic service-level interference into account.

[9] and [10] apply linear ARMAX regression to express correlation between performance and primitives for VM-based applications. They have not considered the problem of QoS interference and cloud dynamics; henceforth their use of ARMAX and the resulting model is different to ours. Other machine learning techniques (e.g., ANN [3,4,6], SVM [3], TreeBased [12] and nonlinear [11]) have been applied for performance modeling in the cloud. Particularly, [3] and [6]

conduct off-line model training and their experiments are based on highly restricted and controlled environment, which limits the evaluation of model accuracy. Unlike our models, their models deal with a fixed predefined set of primitives. Their dynamicity is limited as these models are trained with the provision values of these primitives. Furthermore, they assume homogenous QoS model for an application and its replicas.

Although not designed for cloud, [20] propose a hybrid, fine-grained performance modeling where linear AR is used to predict demand of primitives and Kalman filter is applied to tune the actual model. However, they do not handle QoS interference and cloud dynamics explicitly. In addition, they do not consider software CPs. [21] realize QoS modeling based on change-point detection techniques. Nevertheless, they do not intend to discuss dynamic QoS sensitivity.

All aforementioned QoS modeling solutions are semi-dynamic because they only tend to dynamically model *how* hardware CPs correlate with QoS, but their treatments for *which* and *when* have been static. In addition, they focus on performance QoS only and do not consider software CPs. As a result, their handling of dynamic QoS sensitivity are implicit.

VI. Conclusion and Future Work

We have proposed a novel self-adaptive, sensitivity-aware QoS modeling approach grounded on symmetric uncertainty and two machine learning techniques, ARMAX and ANN to reach two formulations of the QoS model. The approach can capture the dynamics of QoS sensitivity by determining *which*, *when* and *how* primitives correlate with QoS at runtime. In addition, we cater for QoS granularity, QoS interferences and cloud dynamics. Our approach considers fine-grained model as well as both software and hardware CPs. We have implemented our approach as a middleware solution that adaptively creates fine-grained QoS models. We have experimentally evaluated our approach with respect to accuracy, sensitivity to data size and efficiency using the *RUBiS* benchmark and the FIFA 1998 workload trends. The results reveal that our approach is effective and produces better accuracy as when compared with the conventional models in various cases. Experiments also imply that the proposed S-ANN tends to be more accurate than S-ARMAX, when the fluctuation of QoS increases. Therefore, one of the future extensions is to adaptively determine the best technique to train QoS model at each interval.

The implication of self-adaptive QoS modeling and its dynamic sensitivity analysis to adaptation in the cloud are vast: the model can assist autonomic software agents in predicting causes of probable risks leading to violations; working on appropriate mitigation strategies and/or even planning for optimal QoS design and online adaptation strategies. Moreover, it can assist problems related to QoS self-management, self-adaptation, resource utilization and/or elastic provision. In future papers, we will report on novel applications benefiting from the proposed modeling approach.

References

[1] V.C. Emeakaroha et al, "Low level metrics to high level SLAs -LoM2HiS framework: bridging the gap between monitored metrics and SLA parameters in cloud environments," *in Proc. of the High Performance Computing and Simulation Conference*, 2010.

[2] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," *in 5th conference on Computer systems*, New York, USA, pp. 237–250, 2010.

[3] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao and K. Dutta, "Modeling virtualized applications using machine learning techniques," *in Proc. of the 8th Conference on Virtual Execution Environments*, pp. 3-14, 2012.

[4] G.Kousiouris et al, "Translation of application-level terms to resource-level attributes across the Cloud stack layers," *In IEEE Symposium on Computers and Communications*, pp. 153-160, 2011.

[5] R.N. Calheiros, R. Ranjan, and R. Buyya, "Virtual machine provisioning based on analytical performance and QoS in cloud computing environments," *in Proc. of the 40th Conference on Parallel Processing*, Taipei, Taiwan, pp. 295–304, 2011.

[6] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao, "Application performance modeling in a virtualized environment," *In 16th Symposium on High Performance Computer Architecture*, January 2010.

[7] Y. Chen, S. Iyer, X. Liu, D. Milojicic and A. Sahai, "SLA decomposition: translating service level objectives to system level thresholds," *in the 4th Conference on Autonomic Computing*, 2007.

[8] Y. Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury, "MIMO control of an apache web server: modeling and controller design," *In Proc. of American Control Conference*, 2002.

[9] P. Padala et al, "Automated control of multiple virtualized resources," *In Proc. of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems*, pp. 13-26, 2009.

[10] Q. Zhu and G. Agrawal, "Resource provisioning with budget constraints for adaptive applications in cloud environments," *in Proc. of the 19th ACM Symposium on High Performance Distributed Computing*, 2010.

[11] R.C. Chiang and H.H. Huang, "Tracon: interference-aware scheduling for data-intensive applications in virtualized environments," *in Proc. of 2011 Conference for High Performance Computing, Networking, Storage and Analysi*s, New York, USA, pp. 1-12, 2011.

[12] P. Xiong et al, "Intelligent management of virtualized resources for database systems in cloud environment," *In 27th Conference on Data Engineering*, 2011.

[13] W. S. Sarle, Neural networks and statistical models, 1994.

[14] Rice University Bidding Systems, http://rubis.ow2.org/.

[15] Xen: a virtual machine monitor, http://xen.xensource.com/.

[16] I.H. Witten, E. Frank, Data mining: practical machine learning tools and techniques, Morgan Kaufmann: Los Altos, CA, 2005.

[17] B.Widrow and D. Samuel, Stearns: adaptive signal processing, Prentice Hall, 1985

[18] M.Riedmiller and H.Braun, "RPROP-a fast adaptive learning algorithm," *in Proc. of ISCIS VII,* University, 1992.

[19] G.Box, G.M. Jenkins and G.C. Reinsel, Time series analysis: forecasting and control, third edition. Prentice-Hall, 1994.

[20] T. Zheng, M. Litoiu, and M. Woodside, "Integrated estimation and tracking of performance model parameters with autoregressive trends," *In Proc. of 2nd conference on Performance engineering*, 2011.

[21] P. Bodık et al, "Statistical machine learning makes automatic control practical for internet datacenters," *In Proc. of the 2009 conference on Hot topics in cloud computing*, 2009.

[22] Y. Zhang, G. Huang, X. Liu, and H. Mei, "Integrating resource consumption and allocation for infrastructure resources on-demand," *In Proc. of 3rd IEEE Conference on Cloud Computing*, 2010.

[23] J.Li et al, "Profit-based experimental analysis of IaaS cloud performance: impact of software resource allocation," *In Proc. of the 9th Conference on Service Computing*, 2012.

[24] M. Arlitt and T. Jin, "A workload characterization study of the 1998 world cup web site," *IEEE Network*, 14(3), pp. 30 –37, May 2000.

[25] Google App Engine, http://code.google.com/appengine/

[26] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/

[27] Tomcat, http://tomcat.apache.org/

[28] JBoss, http://www.jboss.org/

[29] R. Picard and D. Cook. "Cross-validation of regression models" *Journal of the American Statistical Association***,** 79(387), pp. 575–583, 1984.

[30] Apache Mathematics Library, http://commons.apache.org/math/

[31] Encog Framework, http://www.heatonresearch.com/encog