DynaQoS: Model-free Self-tuning Fuzzy Control of Virtualized Resources for QoS Provisioning

Jia Rao, Yudi Wei, Jiayu Gong and Cheng-Zhong Xu
Department of Electrical and Computer Engineering
Wayne State University
Detroit, Michigan, USA
{jrao, ydwei, jygong, czxu}@wayne.edu

***Abstract*—Cloud elasticity allows dynamic resource provisioning in concert with actual application demands. Feedback control approaches have been applied with success to resource allocation in physical servers. However, cloud dynamics make the design of an accurate and stable resource controller more challenging, especially when response time is considered as the measured output. Response time is highly dependent on the characteristics of workload and sensitive to cloud dynamics. To address the challenges, we extend a self-tuning fuzzy control (STFC) approach, originally developed for response time assurance in web servers to resource allocation in virtualized environments. We introduce mechanisms for adaptive output amplification and flexible rule selection in the STFC approach for better adaptability and stability. Based on the STFC, we further design a two-layer QoS provisioning framework, DynaQoS, that supports adaptive multi-objective resource allocation and service differentiation. We implement a prototype of DynaQoS on a Xen-based cloud testbed. Experimental results on an E-Commerce benchmark show that STFC outperforms popular controllers such as Kalman filter, ARMA and adaptive PI by at least 16% and 37% under both static and dynamic workloads, respectively. Further results with multiple control objectives and service classes demonstrate the effectiveness of DynaQoS in performance-power control and service differentiation.**

I. Introduction

As server virtualization grows increasingly popular and mature, hosting enterprise applications in a cloud has become an attractive solution for scalability and cost-efficiency. Applications running within virtual machines (VM) have on-demand access to compute resources in response to increased application loads. On the other hand, virtual resources can be maintained at a minimal level during off-peak periods in order to reduce cost. Thus, virtual machines should be dynamically provisioned to match actual application demands, rather than the peak one. However, these demands are difficult to estimate due to time-varying and diverse workload. More importantly, client-perceived quality-of-service (QoS) should still be maintained in the presence of background dynamic resource provisioning. These observations call for an effective approach that automates resource allocation for cloud users.

Regulatory control is a promising method for resource allocation, in which a feedback controller enforces service-level objectives (SLO) while minimizing the resources required. More importantly, if properly designed, this type of control can provide predictable performance with theoretical stability guarantees. In general, a feedback controller applies the *control input* to a target system in order to regulate the *measured output* to the value of a *desired output* [5].

There are many control approaches that have been applied with success to resource allocation in physical servers; see [1], [12], [18], [8], [10] for examples. Recent studies have focused on the application of control approaches for the allocation of virtualized resources in clouds [11], [24], [7], [14], [13]. The cloud adds new challenges to the QoS-oriented resource allocation, in addition to workload dynamics. Different from physical servers, a virtual server may see a varying capacity in the cloud. The dynamics in the capacity can be due to the uncertainties in resource scheduling, opportunistic use of additional market-based resources(e.g. Amazon spot instances [2]) or even the rogue behavior of malicious users [29].

Many existing work used indirect metrics such as workload arrival rate [11], [24] and CPU utilization [7], [14], instead of response time as the measured output. These work relied on the assumption that there are always static relationships between the metrics and response time. The relationships are usually determined either by industry practice or offline testing. Although easier to control, the use of indirect metrics may not be effective in a dynamic cloud environment. In Section II, we show that when the CPU utilization is 80%, the response times of an E-Commerce benchmark can have as large as 150% variations with different capacities. Therefore, with dynamic capacity, resource utilization is not readily translated to application-level performance and models obtained under one capacity setting are likely to be inaccurate for other settings. In practice, response time is a good measure of client-perceived QoS. However, response times behave nonlinearly with respect to resource allocations and are highly dependent on the characteristics of workload, as well as server capacity. This nonlinearity poses challenges to design a stable and accurate controller.

To address the issue of the lack of an accurate server model, the work in [7], [13] applied adaptive control approaches based on model approximation. However, these approaches pose limitations on how fast the workload and the system behavior can change [30]. In [27], we developed a two-layer self-tuning fuzzy control (STFC) approach for QoS assurance in web servers with respect to response time. In this paper, we extend the STFC approach to resource allocation in virtualized environments by introducing an extra self-tuning

978-1-4577-0103-0/11$26.00

output amplification and flexible rule selection mechanism. In comparison with other popular controllers, STFC shows better adaptability and stability. Based on the STFC, we further design a two-layer QoS provisioning framework, DynaQoS, that supports adaptive multi-objective resource allocation and service differentiation.

To evaluate the performance of STFC and the DynaQoS framework, we built a cloud testbed based on a Xen environment. We conducted experiments to allocate CPU resources to VM clusters running the TPC-W [22] E-Commerce benchmark. For comparison with STFC, we also implemented three popular controllers within the DynaQoS framework: a model-independent adaptive PI controller and two controllers based on local models approximation: Kalman filter and ARMA controllers. Experimental results show that, STFC outperformed the closest competitor by 16% and 37% under static and dynamic workloads, respectively. The output amplification reduces the settling time to 3 control intervals and the flexible rule selection improves the stability. Further results on simultaneous control of performance and power show that, DynaQoS was able to find a balance between conflicting objectives. In service differentiation, DynaQoS guaranteed the performance of the premium class and provided better service to the basic class outperforming a popular differentiation policy.

The rest of this paper is organized as follows. Section II discusses the challenges in automatic cloud resource management. Section III and Section IV elaborate the key designs and implementation of DynaQoS, respectively. Section V gives experimental results. Related work is presented in Section VI. We conclude this paper in Section VII.

II. Background and Motivation

To build a resource controller realizing a high-level objective, a mathematical model that captures the relationship between the allocated resource and the high-level metric is necessary. Given the model, any deviation of the high-level metric from the desired value can be corrected by applying adjustment in the resource allocation. However, the determination of the system model in a dynamic cloud environment is not trivial. Workload and cloud dynamics can possibly render prior system models invalid and result in poor control performance.

1) Uncertainties in resource scheduling: In [3], the authors showed that time-sharing of CPU resources in multiple VMs can provide much more predictable performance than I/O sharing. With advances in multi-core technologies, modern processors are able to embed a number of CPU cores on a single socket. To achieve thread-level parallelism with lower energy cost, heterogeneous CPU architecture and on-chip hardware hyperthreading has gained popularity in modern CPU design. Despite their benefits, they pose significant challenges in VM resource management. "Big" cores are more powerful than "small" cores and hardware threads have distinct performance dependent on whether their sibling threads are executing or not. Current Virtual Machine Monitors such as VMware and Xen, do not consider the underlying architectural differences in VM CPU scheduling. Cloud users may observe different CPU capacities when scheduled with "big" or "small" cores; or with hardware threads from busy or idle cores.

Consider a virtual cluster consisting of 4 VMs executing a mapreduce job to classify approximately 20000 documents into 20 different newsgroups on on a DELL server with 12 CPU cores. Each physical core has two hardware threads which can be scheduled simultaneously. The default CPU scheduling in the Xen hypervisor, referred to *CPU_UNPIN*, allows the two threads from the same core to be scheduled together. In comparison, we experimented with another scheduling scheme, *CPU_PIN*, which ensures that no hardware threads from the same core are scheduled at the same time. It guarantees that each scheduled hardware thread gets the full processing capacity on a core. The experimental results shows that the *CPU_PIN* scheduling reduced the execution time by as large as 37% (reduced from 918.3 second to 668.5 second). This reveals a significant variation of CPU capacity under the same nominal resource allocation.

2) Opportunistic use of variable resources: Besides the uncertainties underlying cloud systems, the dynamics in VMs' capacity can also come from market-based accesses to additional compute capacity. Amazon Elastic Compute Cloud (EC2) provides *Spot Instances* [2] as a complementation to *On-demand Instances* and *Reserved Instances*. Different from the other two, Spot Instances make use of unused Amazon EC2 capacity and are charged a much lower spot price. Cloud users bid on spare capacity and run Spot Instances as long as their bids exceed the spot price. Spot price changes with the supply and demand and the instances whose owner's bids are below current spot price will be terminated. If hosted applications are resilient to nondeterministic capacity additions and removals, mixing reserved capacity (i.e. on-demand or reserved instances) with transient capacity (i.e. spot instances) will be a cost-effective way for time-varying workload and limited budget.

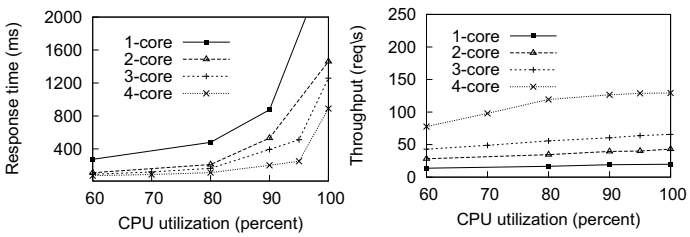
However, the nondeterminism in compute capacity poses significant challenges in modeling resource to application performance. Figure 1 plots the application performance of TPC-W against the resource utilization (i.e. CPU utilization) under different capacities. We threw 500 shopping clients to the TPC-W virtual cluster and created different levels of capacities by adding or removing VMs from the virtual cluster. For example, a total number of 4 VMs, each with one core, is equavilant to a capacity of 4-core. As shown in Figure 1, the relationship between application performance and the CPU utilization changes with capacity. When the CPU utilization is 80%, both response time (Figure 1(a)) and throughput (Figure 1(b)) show as large as 150% variations with different capacities. With dynamic capacity, resource utilization is not readily translated to application performance. System models obtained under one capacity setting are likely to be inaccurate for other settings. Without an accurate system identification, control-based resource allocation suffers poor performance.

III. The DynaQoS Framework

In this section, we present the design of DynaQoS, a prototype of the fuzzy control-based VM resource allocation.

A. Design of DynaQoS

As shown in Figure 2, DynaQoS is composed of two layers of controllers. The first layer is a group of self-tuning fuzzy



(a) Response time (b) Throughput

Fig. 1. Different resource-performance relationship due to dynamic capacity.

controllers (STFC) that control individual objectives. During each control interval, a STFC queries the corresponding QoS profile manager for the reference value of the controlled metric. A QoS monitor periodically reports the achieved value of the metric. The metrics to be controlled can be conventional application-level performance metrics such as response time or throughput; or any user-defined high-level metrics, we show an example of such metrics in Section V. In a cloud environment, more interesting control can be the control of leasing expenses (based on variable resource prices) towards a target of leasing budget, or the control of VM level power consumption below a per VM budget [9]. The STFC takes the difference between the reference value and the achieved one as well as the change of the error as its input and outputs a resource request to the second layer gain scheduler.

When there are multiple control objectives, the second layer gain scheduler aggregates the resource requests from individual STFCs and forms a unified one to be submitted to the cloud resource management API. The aggregation of individual requests is based on the weights (gain) of each STFC in the determination of the final request. The gains are dynamically adjusted according to the control error of STFCs. Service differentiation is necessary if multiple service classes exist and the aggregated resource demand is beyond the available capacity. We define multi-level objectives in the QoS profile manager for each service class. If resource contention is detected and it can not be resolved for a certain number of control intervals, the class with lowest priority modifies its control objective to the next level.

B. The Self-tuning Fuzzy Controller

Due to workload and cloud dynamics, the relationship between allocated capacity and received service quality exhibits considerable nonlinearities. The relationship can often be linearized at fixed operating points. It is well known that the linear approximation of a nonlinear system is accurate only within the neighborhood of the operating point. Abrupt changes in workload traffics and the nondeterminism in VM capacity can possibly make the simple linearization inappropriate. Instead of modeling the system in mathematical equations, fuzzy control employs the control rules of conditional linguistic statements on the relationship of allocated resource and the high-level objectives [6]. The fuzzy control rules are able to embed human expert's experiences and the rule base is easily updated by adding new knowledge. There are works that applied fuzzy control to QoS guarantees in web server [27] and computer networks [4] with success.

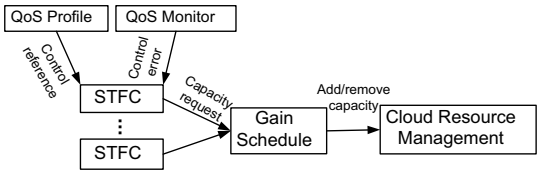


Fig. 2. The structure of the DynaQoS framework.

Figure 3 illustrates the structure of the Self-tuning Fuzzy Controller. It consists of three components, namely the fuzzy logic controller, the scaling-factor controller and the output amplifier. The resource allocated in control interval $k+1$, denoted by $u(k+1)$, is adjusted according to its error $e(k)$ (i.e., the normalized difference between the reference value and the achieved one) and change of error $\Delta e(k)$ in previous control interval k using a set of control rules embeded in the fuzzy logic controller. $e(k)$ and $\Delta e(k)$ are calculated using the reference value $r(k)$ and the observed value $y(k)$. For the stability of the control system, we define the normalized error $e(k)$ in a range of $[-1, 1]$:

$$e(k) = \begin{cases} \frac{r(k)-y(k)}{r(k)} & 0 \le y(k) \le 2r(k); \\ -1 & y(k) > 2r(k). \end{cases}$$

Based on these, the controller calculates resource adjustment $\Delta u(k)$ for next control interval. The calculated resource adjustment is then fed into the next layer gain scheduler.

The fuzzy logic controller contains four building blocks. The actual fuzzy logic is implemented as a set of *If-Then* rules about quantified control knowledge about how to adjust the allocation according to $e(k)$ and $\Delta e(k)$. The fuzzification interface converts controller inputs into certainties in numeric values of the input membership functions. The inference mechanism activates the rule-base and applies fuzzy rules according to the fuzzified inputs and generates the fuzzy conclusions for the defuzzification interface. The defuzzification interface converts fuzzy conclusions into the change of allocation in numeric value.

The STFC is built on the static fuzzy logic controller by adding the self-tuning scaling factors and the output amplifier. There are three scaling factors: input factors K_e and $K_{\Delta e}$, output factor α and output amplifier $K_{\Delta u}$. The change of input scaling factors changes the connection of input values to suitable rules, The change of output scaling factor and the amplifier together adjust the amplitude of the control input. The actual inputs of the fuzzy logic controller are $|K_e|e(k)$ and $|K_{\Delta e}|\Delta e(k)$. Thus, the resource allocated to the VM during management interval $k+1$ is

$$u(k+1) = u(k) + \alpha|K_{\Delta u}|\Delta u(k) = \int \alpha K_{\Delta u}\Delta u(k)dk.$$

1) Design of the rule base: The design objective is to translate human expert's knowledge into a set of control rules to control the resource allocation without a model of the dynamic cloud environment. In the fuzzy logic controller, the control rules are defined using linguistic variables. For brevity, linguistic variables "$e(k)$", "$\Delta e(k)$", and "$\Delta u(k)$" are used to describe $e(k)$, $\Delta e(k)$, and $\Delta u(k)$, respectively. The linguistic variables assume linguistic values NL(negative

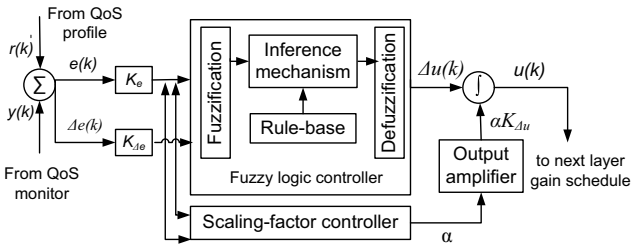
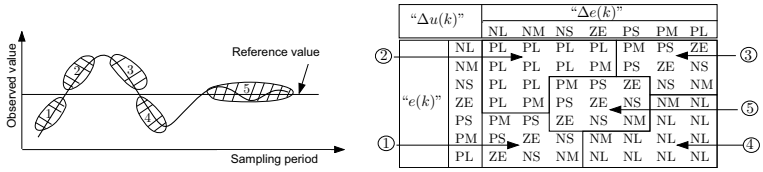


Fig. 3. The structure of the STFC.



(a) The control effect

"Δu(k)" \ "Δe(k)"	NL	NM	NS	ZE	PS	PM	PL
NL	PL	PL	PL	PL	PM	PS	ZE
NM	PL	PL	PL	PM	PS	ZE	NS
NS	PL	PL	PM	PS	ZE	NS	NM
ZE	PL	PM	PS	ZE	NS	NM	NL
PS	PM	PS	ZE	NS	NM	NL	NL
PM	PS	ZE	NS	NM	NL	NL	NL
PL	ZE	NS	NM	NL	NL	NL	NL

(b) The rule table

Fig. 4. Design of the fuzzy control rules.

large), NM (negative medium), NS (negative small), ZE (zero), PS (positive small), PM (positive medium), and PL (positive large).

Figure 4(a) gives an simple illustration of typical control effect. In this figure, we identify five zones with different characteristics. Zone 1 and 3 are characterized with opposite signs of $e(k)$ and $\Delta e(k)$, in which the error is self-correcting and the achieved value is moving toward the reference value. Thus, $\Delta u(k)$ needs to be set either to speed up or to slow down current trend. Zone 2 and 4 are characterized with the same signs of $e(k)$ and $\Delta e(k)$, in which the error is not self-correcting and the achieved value is moving away from the reference value. Therefore, $\Delta u(k)$ should be set to reverse current trend. Zone 5 is characterized with rather small magnitudes of $e(k)$ and $\Delta e(k)$. Therefore, the system is at a steady state and $\Delta u(k)$ should be set to maintain current state and correct small deviations from the reference value.The resulted control rules are summarized in Figure 4(b). For example, when "$e(k)$" and "$\Delta e(k)$" are NL and PS, "$\Delta u(k)$" is set to PM.

2) Fuzzification, inference and defuzzification: We take the same design for the membership function and inference mechanism from our previous work; see [27] for details.

3) Design of the self-tuning controller: The fuzzy logic controller only defines the basic control rules according to the inputs of $e(k)$ and $\Delta e(k)$. It outputs the sign and magnitude of the allocation adjustment $\Delta u(k)$. With cloud dynamics, there could be a lot of fluctuations in the control effect. To achieve accurate, responsive and stable control, the following practical issues should be addressed:

1) When there are abrupt workload or capacity changes, the control should be responsive enough to correct the resource discrepency within a small number of steps.
2) When there are considerable fluctuations in the control effect, it may be due to two reasons. The fluctuations may come from the inaccuracies of the controller that incurs control overshooting; or it may be due to the process delay [19] of resource allocation. A process delay is the time between the change of resource allocation and the actual adjustment effect can be observed in application performance. Both problems can be alleviated by decreasing the control magnitude or prolonging the control interval to stabilize the control effect.

To address the above issues, we design the self-tuning controller to have adaptive output magnitude and flexible control rules. The self-tuning features are realized by dynamically changing the input, output scaling factors and the output amplifier. The output scaling facotr α and the output amplifier $K_{\Delta u(k)}$ together determine the magnitude of the allocation adjustment. In our previous work [27], we used another level of fuzzy controller to adjust the output scaling factor α. However, the output $\Delta u(k)$ of the fuzzy logic controller is within the range of $[-1, 1]$. The change of α has limited effect on the magnitude of the control output. To overcome abrupt workload and capacity changes, the magnitude needs to be changed dynamically based on current conditions. We preserve the adaptive controller of α as in [27] and add a self-tuning output amplifier. The amplifier implements heuristic control knowledge as follows:

$$K_{\Delta u(k)} = |\frac{c}{2} \cdot e(k)|,$$

where c is the current allocation for a specific resource. For example, c can be the cap value of the CPU allocation in a Xen platform. The amplifier follows a heuristic rule that the maximum resource adjustment should not exceed half of current capacity for stability and should be proportional to the control error for adaptability. Note that the direction of the adjustment is still determined by the fuzzy logic.

To address the problem of process delay and control inaccuracies, fuzzy control rules also need to be tuned based on current conditions. Recall that the actual inputs of the fuzzy logic are $K_e e(k)$ and $K_{\Delta e}\Delta e(k)$, K_e and $K_{\Delta e}$ together determine which membership functions or control rules are to be activated. As shown in Figure 4(b), small values of K_e and $K_{\Delta e}$ activate rules in the center of the rule table, such as the rules in Zone 5; large values are likely to trigger rules like PL and NL. Observations in the control of real plants suggest that it is often desirable to decrease the control magnitude during fluctuations. Thus, we define K_e and $K_{\Delta e}$ as:

$$K_e(k+1) = (1-\gamma)K_e(k) + \gamma e(k),$$

$$K_{\Delta e}(k+1) = (1-\gamma)K_{\Delta e}(k) - \gamma \Delta e(k),$$

where γ is a discount factor that gives more weight on the observance of recent $e(k)$ and $\Delta e(k)$ while still taking history experiences into consideration. We empirically set γ to 0.8 in the experiments. In Figure 4(a), we can see that, during fluctuations the trajectory of control is likely to follow Zone 1 $\rightarrow$ Zone 2 $\rightarrow$ Zone 3 $\rightarrow$ Zone 4. If the pattern is repeated many times, fluctuations exist and $e(k)$ shows as a series of positive and negative values. Gradually, K_e would converge to a small value close to zero, which triggers rules with small or zero magnitude. When the control effect stabilized, if the achieved control deviates from the reference value, K_e will quickly restore to a larger value accumulating $e(k)$ with same signs. The self-tuning scheme works similarly for $\Delta e(k)$ except that $\Delta e(k)$ has the same sign during fluctuations and a

subtraction is used to compensate consecutive $\Delta e(k)$. The self-tuning of the control rules also helps mitigate process delays by generating a sequence of small or zero actuations for more stable control.

C. Scheduling multiple objectives

There exist many control problems in which the consideration of multiple objectives is required, and these objectives may conflict with each other. In cloud computing, a cloud user may want to keep the application level response time and throughput in certain ranges that satisfy SLA objectives. However, the user may be simultaneously required to maintain the power consumption of his or her applications to be below a specified bound. The *Gain schedule* component in the DynaQoS framework implements a weighted scheduling algorithm that synthesizes the outputs from individual STFCs with different objectives. The resulted output is the final resource adjustment request submitted to the cloud resource management. Given individual STFC's outputs $\Delta u_1(k), \ldots, \Delta u_n(k)$ and the corresponding errors $e_1(k), \ldots, e_n(k)$ as inputs, the synthesized adjustment $\Delta u(k)$ is defined as

$$\Delta u(k) = \sum_{i=1}^{n} \Delta u_i(k) \cdot w_i,$$

where $w_i = \frac{|e_i(k)|}{\sum_{j=1}^{n} |e_j(k)|}$.

We assume that there always exists a control solution for the multiple-objective control problem. The gain scheduling algorithm depends on the careful selection of the reference values by the cloud user. If a control solution exists, the algorithm applies dynamic weights to individual STFCs based on their control errors. In the extreme case, the multiple-objective control degrades to a single-objective control, if one objective is satisfied generating near zero control errors.

D. Realizing service differentiation

Service differentiation is desirable when the aggregated resource demand of multiple service classes is beyond the limit of allocated resources. Although cloud systems allow prompt allocation of resources in response to the increase in client traffic, there are still cases that the total demand can temporally exceeds available capacity. First, the cloud user who owns the cloud application may run out of budget preventing him adding more capacity during a spike load. Second, applications running on the market-based cloud resources may see capacity fluctuations due to the supply and demand of the dynamic capacity. For example, Amazon EC2 users may choose to host applications on a cluster of VMs containing both reserved and spot instances. The spot instances will be terminated if the spot prices exceed the users' bids resulting in a reduction in the total capacity. Finally, complications in cloud resource scheduling and performance interference also contribute to the variation of capacity. For example, results in Section II show approximately 40% variations in application performance due to scheduling dynamics; the authors in [29] also demonstrated possible CPU cycle stealing between cloud users.

To provide QoS guarantees, we consider the service differentiation to be initiated by individual service classes. When resource contentions are detected, the service class with a lower priority would adapt its SLO (e.g. a response time target) to a lower level. By setting different control objectives, the premium class will receive more resources than the basic class while the basic class avoids starvation maintaining a degraded level of service. We enforce strict priorities between classes. That is the class with a higher priority adapts to a lower level only when the lower priority classes have reached their minimum service levels. To detect resource contentions, DynaQoS follows a simple heuristic rules tracking the statistics of the control performance. If DynaQoS sees a predefined number of serious SLO violations (i.e. $\Delta e(k) < 0$ and $|\Delta e(k)| > \epsilon$) for a certain level of class and the resource adjustment did not correct the control errors (i.e. $\Delta u(k) > 0$), classes with lower priorities would start to adapt to a lower level. Classes at different ranks have the tolerance of different numbers of violations, which ensures that clients with lower priorities will always degrade before the high priority clients. For example, the premium class may only tolerate 10 consecutive violations while the basic class can bear up to 30. When the capacity is limited, the basic class would release the resource first.

IV. System Implementation

A. Cloud applications

We selected the TPC-W [22] benchmark as the hosted cloud application for the evaluation of DynaQoS. TPC-W is an E-Commerce benchmark that models after an online book store, which is CPU-intensive and has the database tier as the bottleneck. We employed a three-tier cluster implementation of TPC-W, which consists of an Apache web server (version 1.3.11) and a group of Tomcat (version 5.5.20) application and MySQL (version 5.0.45) database servers. We put the Apache and all the Tomcat servers into one VM forming a unified front-end, and replicated the MySQL server into a number of DB VMs, one MySQL per VM. The DB virtual server farm was further divided into several virtual clusters, each of which was dedicated to a service class. The apache web server accepts and classifies client requests into different classes. It assigns requests from different classes to different DB virtual clusters. We modified the Apache web server to exam the content of the requests and assign different port numbers to different classes. Based on the port number, Apache module `mod_jk` redirects the requests to corresponding tomcat workload balancers which are responsible for individual virtual clusters. The tomcat balancers dispatch the requests within the virtual cluster in a round-robin manner. There may be consistency issues if the requests from a same client session write to different DB VMs. In this paper, we focus on the evaluation of DynaQoS in resource allocations and leave the issues to future work. To avoid consistency problems, we used read primary browsing mix in TPC-W as the client workload.

We empirically determined that the DB tier was the bottleneck tier under the browsing workload and focused on the CPU allocation to the DB clusters. There are two ways to change the allocation to a DB virtual cluster. One is to change the number of DB VMs in a cluster and the tomcat balancer handles the join and leave of cluster members. Another approach is to have a fixed number of DB VMs and change the

CPU allocations to individual VMs. To evaluate DynaQoS in fine-grained resource allocation, we took the second approach.

B. Testbed

Our testbed consists of a virtual server, client and NFS servers. The physical machines for virtual hosting were two DELL servers with two Intel Xeon X5650 CPUs and 32 GB memory. Each CPU has 6 cores with hyperthreading enabled resulting a total capacity of 24 logical CPUs. The front-end and back-end DB VMs were hosted on separate machines. We configured the front-end VM with 8 core and 4 GB memory. The DB VMs, each with 4 core and 2 GB memory, resided on the other machine. We used a number of client machines each with 8 cores and 8 GB memory to generate workload for the TPC-W. The NFS server used a RAID5 partition to serve the VM disk images. We used Xen version 4.0 as our virtualization environment. `dom0` and guest VMs were running Linux kernel 2.6.32 and 2.6.18, respectively. All the severs were connected by Gigabit Ethernet network.

C. Implementation of DynaQoS

QoS monitor. We consider the client-perceived response time as the measure of application-level performance. We modified the TPC-W's workload generator to maintain a log of finished requests. A small utility program parses the log to calculate the average response time for every control interval.

QoS profile manager. Each service class works with a QoS profile manager to determine the control objective. The control objectives are specified in terms of a set of desired response times with different levels. For service differentiation, the profile manager also sets the number of SLO violations that can be tolerated by a class before a target adaptation is initiated. For the service differentiation experiment in Section V-C, we considered two classes: *Premium* and *Basic*. They both have three levels of SLO, {1s, 5s, 10s}, and with adaptation thresholds: 10 and 30 violations, respectively.

Self-tuning fuzzy controller. STFC has been implemented as a set user-level daemons in the virtual host (i.e. `dom0` in a Xen environment). It takes the measured application-level performance (from QoS monitor) and the performance objective (QoS profile manager) as input and outputs the resource adjustment to Xen's management interface. If multiple control objectives exist, two or more STFCs form a unified request. The control interval is set to 30 seconds for all the experiments.

CPU resource allocation. CPU resources are allocated to each DB VM via Xen Credit Scheduler in terms of cap values. A *cap* value represents the upper limit of CPU time can be consumed by a VM in percentage. For a virtual cluster with 4 VMs and each with 4 cores, the CPU allocation can be in the range of $[1, 1600]$. The CPU time is allocated to individual virtual clusters. We assume good load balancing by the Tomcat balancer, thus distribute CPU cap values uniformly within the cluster. All VMs are given the same weight during allocation.

V. Experimental Results

A. Comparing STFC to other popular control methods

Experiments are designed to study the efficacy of DynaQoS in the determination of proper CPU allocations under both static and dynamic workloads. We have also implemented three popular controllers within the DynaQoS framework:

Kalman filer [7] is a data processing method that uses a series of measurement with noises to produce values closer to the true values of the measurement. It is used in [7] to track the utilization of CPU and allocate CPU resources correspondingly to maintain the utilization to a specific value.

Adaptive proportional integral (PI) [14] directly tracks the error of the measured response time and the target and adjusts the CPU allocation to minimize the error. The gains of the proportional and integral parts are set to $|\frac{c}{2} \cdot e(k)|$, similarly as the STFC, to allow adaptive control.

Auto-regressive-moving-average (ARMA) [13] builds a local linear relationship between the allocated CPU resource and the response time with recently collected samples. If response time deviates from the target value, the controller computes the allocation that corrects the error based on the obtained model. The controller is configured to use a second-order ARMA model with a window size of 20.

To measure the performance of DynaQoS, we define a metric, relative deviation $R(e)$, based on root-mean square error:

$$R(e) = \frac{\sqrt{\sum_{k=1}^{n} e(k)^2/n}}{r(k)}.$$

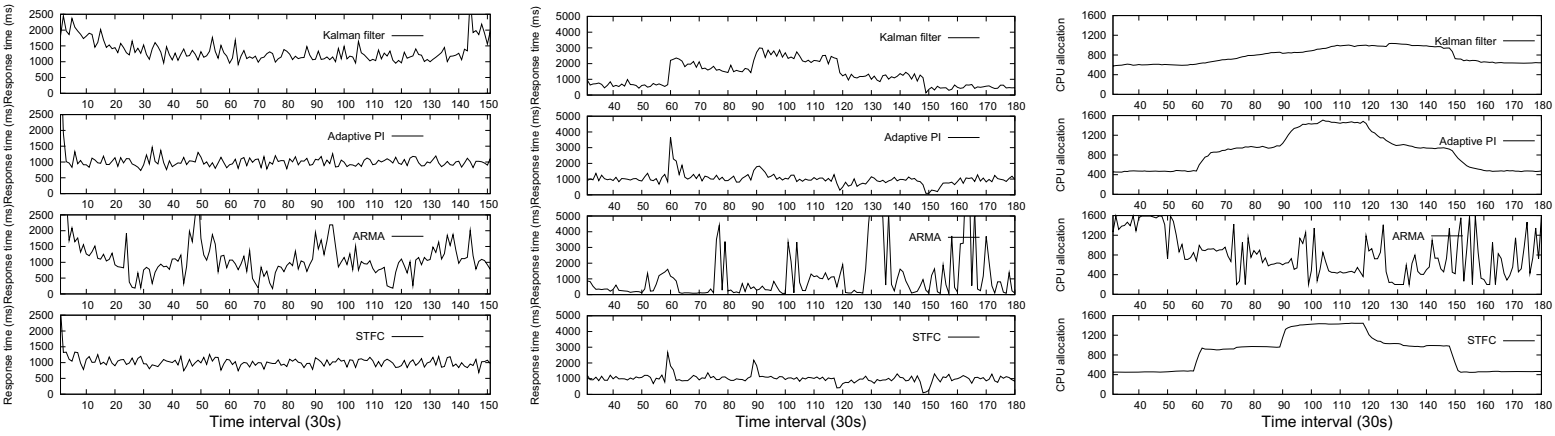
The smaller the $R(e)$, the more the achieved response time concentrates near the target value and better the controller's performance. To compare the performance of different controllers, we take the performance of STFC as a baseline and define the performance difference between STFC and other controller as:

$$PerfDiff = \frac{R(e)_{other} - R(e)_{STFC}}{R(e)_{STFC}}.$$

Response times behave nonlinearly with respect to resource allocations especially when the system is in a busy state. We selected the set point of all the controllers to be 1 second except that we followed the controller in [7] and set the Kalman filter's set point to be 90% CPU utilization, which translates to approximately the 1-second response time under the capacity of 16 cores. In this experiment, we only considered one service class with one virtual cluster. The virtual cluster had 4 DB VMs each with 4 VCPUs and its initial capacity was set to 6 cores (a cap of 600).

Figure 5(a) plots the response times of different control methods with static TPC-W workload. The workload was set to 200 browsing clients, each with a mean think time of 1 second. From Figure 5(a), we observe that, all the control methods except ARMA can bring the response time close to the 1-second target, but with different deviations. ARMA requires a local model to predict the proper CPU allocation, thus whenever a deviation from the target is detected it needs several control intervals to build a new model. Figure V-B draws the performance difference of other controllers relative to STFC. STFC outperformed all other controllers by at least 16% with adaptive-PI as the closest competitor.

We are also interested in the adaptability of the controllers under dynamic workload. We instrumented the workload



(a) Static workload (b) Dynamic workload (c) Resource allocation

Fig. 5. Performance comparison of STFC, Kalman filer, Adaptive-PI and ARMA.

generators of TPC-W to change client traffic levels at runtime. The workload generator reads dynamic traffic levels from a trace file, which models after the real Internet traffic pattern [21]. Figure 5(b) plots the response times in a 90-minute period in which the number of clients was changed every 30 intervals. We started with 100 clients and set the client numbers at the $60th$, $90th$, $120th$ and $150th$ interval to be 200, 300, 200 and 100, respectively. From Figure 5(b), we observe that, ARMA performed worst among the controllers with a large number of SLO violations. Kalman filter was not responsive to the workload change and failed to bring the response time back to the 1-second target before the workload changed again. Both of STFC and adaptive-PI successfully maintained the response times around the target. Figure 5(b) also suggests that STFC is more responsive to the workload change with an average settling time of 3 intervals. In contrast, adaptive-PI had an average settling time of 6 intervals. Figure V-B reveals that STFC outperformed adaptive-PI by 37% in terms of relative deviation. It is expected that Kalman filter and ARMA incurred large deviations.

To better understand the performance of the controllers under dynamic workload, we also plot the actual CPU allocations (i.e. cap values) in Figure 5(c). It shows that Kalman filter is not responsive to the workload change and ARMA is too sensitive to the dynamics. We believe that these two methods can be tuned to fit the system better. However, controllers based on local model approximation impose limitation on how fast workloads can change. Both STFC and adaptive-PI do not assume any models of the underlying system, and were able to adjust the CPU allocations properly. In Figure 5(c), we find that, STFC maintains more stable CPU allocations during the period between the workload changes (e.g. between $60th$ and $90th$ intervals). This explains the more stable control performance of STFC in Figure V-B and is due to the flexible control rule selection in STFC.

B. Scheduling multiple objectives

In the previous experiment, we set the control objective precisely at 1-second response time. In many problems, relaxing the "point" control objectives to some suboptimal "regions" is also acceptable. This observation makes the simultaneous control of multiple objectives feasible and of practical importance. DynaQoS applies gain scheduling to balance the trade-off between conflicting objectives. In this experiment, we study the simultaneous control of conflicting objectives, application performance and power within the DynaQoS framework.

We assume that individual cloud users are allocated a power budget to limit the power consumption of their applications. There are existing work performed VM-level power measurement with success [9] and we believe that VM-level power budgeting is desirable in future data centers. In this experiment, we tested with only one cloud user and consider the system-wide power as the VM's consumption. The system-wide power consumption is measured with a WattsUp Pro power meter. The meter records the power consumption every second and we calculate the average power value for each control interval (i.e. 30 second). The more the CPU resources the smaller the response times but the larger the power consumption. The set points were set to 1 second and 250 watt for the response time and power budget, respectively.

Figure 7 plots the response time and power consumption during the control. Before the $30th$ interval, the cloud application contributed most to the power consumption and there existed a balance point that generating acceptable performance for both objectives. DynaQoS successfully identified the balance point and stabilize the response time and power consumption at approximately $800ms$ and $190w$, respectively. Starting the $30th$ interval, we launched background jobs in the host consuming considerable power. In this way, we emulate the circumstances in which some other jobs belonging to the same cloud user eat a lot of power and the user needs to limit the power usage by the cloud application. From the figure, we can see that the combined power consumption immediately exceeded and budget and DynaQoS was able to contain the consumption within the budget by reducing the CPU allocation to the cloud application. When the response time or the power deviated from the target, DynaQoS give more weight to the corresponding STFC. With current settings of the objectives, DynaQoS was able to brought both response time and power close to their targets with stable performance.

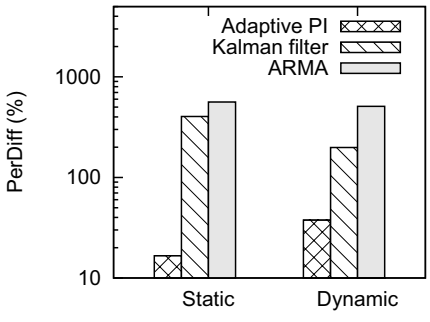


Fig. 6. Performance comparison of STFC and other controllers in relative deviation.

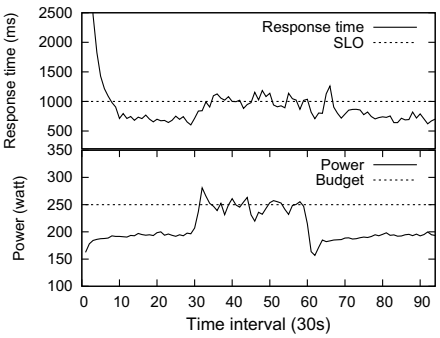


Fig. 7. Simultaneous control of performance and power.

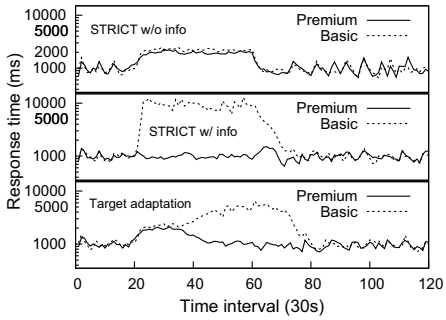


Fig. 8. Service differentiation with different methods.

Once the background jobs ended, DynaQoS returned back to the $(800ms, 190w)$ balance point.

C. Service differentiation

In this section, we investigate the effectiveness of DynaQoS in providing differentiated services to two client classes, *Premium* and *Basic*. DynaQoS applies target adaptation if resource contention is detected. We compare the target adaptation of DynaQoS to a strict differentiation policy (STRICT), that is to guarantee the CPU allocation of the premium class and provide best-effort service to the basic class. To prevent resource starvation of the basic class, we reserve 2-core's capacity to the basic class's virtual cluster. We configured the two classes each with a cluster of 4 DB VMs. Each cluster had 16 VCPUs and can use up to 16 physical CPU resources in theory. The client concurrency levels were set to 200 browsing clients for both classes and resulted in an aggregate CPU demand of approximately 20 CPUs. The server hosting the virtual clusters has a capability of 24 CPUs, thus no service differentiation is needed if the virtual clusters can use the CPU resource freely. We emulated the change in the CPU capacity by restricting the 32 VCPUs of the clusters to the first 12 physical CPUs at the $20th$ interval. This effectively reduced CPU capacity to 12 CPUs. As discussed in Section II-1, the capacity change due to scheduling dynamics is possible in current cloud platforms. More importantly, the change in the actual CPU capacity may not be reflected in the nominal CPU allocations.

In Figure 8, we compare DynaQoS's target adaptation with the STRICT policy. We implemented two variations of the STRICT policy, one with the knowledge of the exact value of the new capacity (*STRICT w/ info*), one without (*STRICT w/o info*). As shown in Figure 8, DynaQoS was able to detect the resource contention at the $32th$ interval because the premium class had seen 10 serious violations in the response time. It triggered the basic class's target adaptation to the next level, 5 second. The performance of both classes stabilized at the $50th$ interval. The premium class succeeded to maintain the 1-second target and the basic class achieved a response time close to its new target. After we increased the capacity at the $60th$ interval, DynaQoS took 10 intervals to detect the change and reset the target of the basic class back to 1 second. In contrast, *STRICT w/o info* policy failed to detect the capacity change and did not enforce service differentiation.

In Figure 8, we also observe that, with the new capacity information (i.e. 12 CPU), *STRICT w/ info* was able to guarantee the performance of the premium class. But the basic class suffered a 10-second response time compared to 5-second in DynaQoS. An examination of the actual CPU allocation during the contention period revealed that the basic class achieved a 5-second level performance because it obtained more CPU resources than in the *STRICT w/ info* policy. During the contention, DynaQoS kept increase the allocation of both classes until the targets were met. The aggregated CPU allocation in terms of cap values can be beyond the actual capacity (12 CPU). It is equivalent to a work-conserving mode but with bounded allocation to the basic class for the purpose of differentiation. Different from DynaQoS, *STRICT w/ info* enforces that the total allocation is not beyond 12 CPU and the basic class only got an allocation of 2 CPU or whatever was left by the premium class. The non-work-conserving mode in *STRICT w/ info* policy wasted some CPU time which can otherwise be used by the basic class.

VI. Related Work

Provisioning of QoS guarantees has been an active research topic. Early work focused on provisioning service guarantees or differentiation under fixed capacity. Methods such as queuing-theoretic analysis, traditional feedback control and adaptive control have been studied extensively. In [23], the authors assumed a $G/G/1$ queuing model to guide to resource allocation so that a desired request latency was achieved. However, this approach depends on the parameter estimation of the model, which is difficult to obtain without understanding the system internals. Due to the absence of the knowledge of underlying systems, traditional linear feedback control was applied to control the resource allocation in web servers [1], [12], [18]. Because the behavior of a web server changes continuously, the performance of the linear feedback control is limited. More recent work applied adaptive control [8], [10] and machine learning [17], [20] to address the issue of the lack of an accurate server model. Although these approaches provide better performance than non-adaptive feedback approaches, they did not address the problem of process delay which is inherent in most resource allocation problems. Our previous work [27] used an adaptive fuzzy control approach without the assumption of a server model to explicitly address

the process delay in resource allocation.

With the proliferation of virtualization technologies, the subjects of traditional resource allocation become virtual machines. The resource allocation problems usually come with constraints defined on application-level QoS or system-level power consumption or both. To automate the resource allocation, regulatory control-based and model-based optimization methods have been studied in literature. Padala et al. [14] proposed an adaptive proportional controller to regulate the CPU utilization to 80%. Kalyvianaki et al. [7] used a Kalman filter controller to track the CPU utilization and adaptively maintained the utilization to 60%. Our work does not assume any relationship between the utilization and the application-level performance. DynaQoS directly regulates response times to a desired value.

More work falls into the category of model-based optimization. The system models are determined either by system identification [26], [25] or moving average-based local linearization [11], [24], [13]. The authors in [26], [25] obtained the model parameters by applying least squares method to offline collected data. The work in [11], [24] employed Kalman filters in the construction of an request arrival rate model. Padala et al. [13] applied an ARMA method to build a local model of resource and application-level performance. System identification can be difficult in some complex systems and models obtained may not be applied to a different system. Methods based on local linearization may not be effective under the workload with large and abrupt fluctuations. Our fuzzy control-based approach does not rely on the understanding of underlying systems and deals with nonlinearities.

There are also existing work focusing on model-independent resource allocation. Xu et al. [28] adopted a fuzzy control approach to mapping application profiles to resource demands. The adaptability of the controller is realized by updating fuzzy rules using offline clustering algorithms. Our work is different in that STFC directly operates on the control error and the change of the error. It avoids the expensive computation of fuzzy rules for adaptability. Rao et al. [16], [15] used reinforcement learning for autonomous resource allocation with discrete steps. In contrast, DynaQoS is capable of allocating resource in a much finer granularity.

VII. Conclusion

In this paper, we have proposed a response time-based fuzzy control approach for the allocation of virtualized resources. We develop a self-tuning fuzzy controller with adaptive output amplification and flexible rule selection. Based on the fuzzy controller, we further design a two-layer QoS provisioning framework, DynaQoS, that supports adaptive multi-objective resource allocation and service differentiation. Experiments on a Xen-based cloud testbed and an E-Commerce benchmark show that the fuzzy controller outperformed three popular controllers for CPU resource allocation. DynaQoS also demonstrated its effectiveness in the simultaneous control of performance and power and service differentiation.

Acknowledgment

This work was supported in part by U.S. NSF grants CNS-0702488, CRI-0708232, CNS-0914330, and CCF-1016966.

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13, January 2002.

[2] Amazon Spot Instances. http://aws.amazon.com/ec2/spot-instances.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, EECS Department, University of California, Berkeley, Feb 2009.

[4] C.-L. Chen. Ieee 802.11e edca qos provisioning with dynamic fuzzy control and cross-layer interface. In *ICCCN*, 2007.

[5] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[6] C.-H. Jung, C.-S. Ham, and K.-I. Lee. A real-time self-tuning fuzzy controller through scaling factor adjustment for the steam generator of npp. *Fuzzy Sets Syst.*, 74, 1995.

[7] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *ICAC*, 2009.

[8] A. Kamra, V. Misra, and E. M. Nahum. Yaksha: a self-tuning controller for managing the performance of 3-tiered web sites. In *IWQoS*, 2004.

[9] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *SOCC*, 2010.

[10] M. Karlsson, C. T. Karamanolis, and X. Zhu. Triage: performance isolation and differentiation for storage systems. In *IWQoS*, 2004.

[11] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and performance management of virtualized computing environments via lookahead control. In *ICAC*, 2008.

[12] C. Lu, T. F. Abdelzaher, J. A. Stankovic, and S. H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *RTAS*, 2001.

[13] 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 *EuroSys*, 2009.

[14] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, 2007.

[15] J. Rao, X. Bu, C.-Z. Xu, and K. Wang. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *MASCOTS*, 2011.

[16] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. VCONF: a reinforcement learning approach to virtual machines auto-configuration. In *ICAC*, 2009.

[17] J. Rao and C.-Z. Xu. Online measurement the capacity of multi-tier websites using hardware performance counters. In *ICDCS*, 2008.

[18] P. P. Renu, P. Pradhan, R. Tewari, S. Sahu, A. Ch, and P. Shenoy. An observation-based approach towards self-managing web servers. In *IWQoS*, 2002.

[19] F. G. Shinskey. *Process Control Systems: Application, Design, and Tuning*. McGraw-Hill, 1996.

[20] V. Sundaram and P. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *IWQoS*, 2003.

[21] The ClarkNet Internet traffic trace. http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html.

[22] The Transaction Processing Council (TPC). http://www.tpc.org/tpcw.

[23] B. Urgaonkar and P. Shenoy. Cataclysm: Handling extreme overloads in internet services. In *WWW*, 2004.

[24] R. Wang, D. M. Kusic, and N. Kandasamy. A distributed control framework for performance management of virtualized computing environments. In *ICAC*, 2010.

[25] X. Wang and Y. Wang. Co-con: Coordinated control of power and application performance for virtualized server clusters. In *IWQoS*, 2009.

[26] Y. Wang, X. Wang, M. Chen, and X. Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *RTSS*, 2008.

[27] J. Wei and C.-Z. Xu. eqos: Provisioning of client-perceived end-to-end qos guarantees in web servers. *IEEE Transaction on Computer*, 55, 2006.

[28] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *ICAC*, 2007.

[29] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram. Scheduler vulnerabilities and attacks in cloud computing. *arXiv:1103.0759v1 [cs.DC]*, Mar 2011.

[30] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, 43, 2009.