QoS Guarantees and Service Differentiation for Dynamic Cloud Applications

Jia Rao, Yudi Wei, Jiayu Gong, Cheng-Zhong Xu
Department of Electrical & Computer Engineering
Wayne State University, Detroit, Michigan 48202
{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 challenging, especially when application-level performance is considered as the measured output. Application-level performance is highly dependent on the characteristics of workload and sensitive to cloud dynamics. To address these 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 representative server workloads show that STFC outperforms popular controllers such as Kalman filter, ARMA and, Adaptive PI in the control of CPU, memory, and disk bandwidth resources under both static and dynamic workloads. Further results with multiple control objectives and service classes demonstrate the effectiveness of DynaQoS in performance-power control and service differentiation.

1 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 requirement. 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 the 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* [6].

There are many control approaches that have been applied with success to resource allocation in physical servers; see [1], [16], [22], [9], [11] for examples. Recent studies have focused on the application of control approaches for the allocation of virtualized resources in clouds [12], [31], [8], [19], [18]. 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 non-uniformity in cloud resources, the opportunistic use of additional market-based resources(e.g., Amazon spot instances [2]) or even the rogue behavior of malicious users [36].

Many existing work used indirect metrics such as workload arrival rate [12], [31] and CPU utilization [8], [19], instead of application-level performance as the measured output. These work relied on the assumption that there are always static relationships between the metrics and the high-level performance. 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 2, 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, application performance metrics such as response time are good measures of

client-perceived QoS. However, these metrics usually 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 [8], [18] applied adaptive control approaches based on model approximation. However, these approaches pose limitations on how fast the workload and the system behavior can change [37]. In [34], 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 multiple resource allocations in virtualized environments by introducing an extra self-tuning 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 have built a cloud testbed based on a Xen environment. We conducted experiments to dynamically control the allocation of CPU, memory, and disk bandwidth resources to three representative cloud applications. These applications include a cluster-based E-Commerce website (TPC-W [29]), an in-memory key-value store (Memcached [17]), and a video streaming server (Darwin streaming server [5]). 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 model approximation: Kalman filter and ARMA controllers. Experimental results show that, STFC outperformed the closest competitor by up to 28% and 61% 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.

In summary, this paper makes the following contributions:

1) Proposing a novel framework, DynaQoS, to provide QoS-guaranteed automatic resource management to cloud applications. The framework also includes the support for multiple-objective control and service differentiation.
2) Proposing a self-tuning fuzzy controller with adaptive control of output magnitude and flexible rule selection.
3) Implementing the DynaQoS framework and demonstrating its effectiveness on a Xen-based cloud testbed.

The rest of this paper is organized as follows. Section 2 discusses the design objectives and the challenges in automatic cloud resource management. Section 3 and Section 4 elaborate the key designs and implementation of DynaQoS, respectively. Section 5 gives experimental results. Related work is presented in Section 6. We conclude this paper in Section 7.

2 Background and Motivation

In this section, we review the design objectives of a resource controller and discuss the challenges in a cloud environment.

2.1 Design Objectives

For cloud users, leasing virtual servers from cloud is advantageous over buying physical machines only if they leverage the elasticity of cloud and dynamically maintained the virtual resources to a level that matches actual application demands [3]. Feedback control is a promising method for automatic resource allocation. A resource controller should achieve diverse control objectives, maintain high resource utilization, while still guarantee application-level QoS and realize service differentiation. In the following, we outline the design objectives of a resource controller:

- **Precise control of multiple user-defined metrics**. The controller should transparently translate user-defined high-level control metrics to low-level resource requirements and allocate resources in a way that minimizes the error between the measured output and the desired output. These metrics include but are not limited to performance metrics (e.g., response time and throughput), expenditure metrics (e.g., dollars per hour), and energy consumption metrics (e.g., Joule per hour). Moreover, the controller should also support the simultaneous control of multiple metrics if necessary. For example, a cloud user may request a response time target of 1 second together with a power budget of 250 watt.
- **QoS guarantee and differentiation**. The controller should ensure that application-level QoS is guaranteed in the presence of dynamic resource control. It requires that the controller be responsive to QoS violations and stable during oscillations. When resources are constrained, the controller should provide differentiated services to different service classes.
- **High resource utilization**. Besides meeting the stated control and QoS objectives, the resource controller should also maintain the utilization of resources at high levels. This avoids the waste in idle resources and leads to savings in the leasing cost.

2.2 The Challenges

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 adjustments in the resource allocation. However, the determination of the system model in a dynamic cloud environment is not trivial. Workload and cloud dynamics make the identification of system models difficult. In the following, we discuss the causes of dynamic capacity and show that the uncertainties affect application modeling.

2.2.1 Performance interference

In a cloud environment, multiple cloud users share the same infrastructure. Although server virtualization helps realizing performance isolation to some extent, VMs from different cloud users may still have chances to interfere with each other. We show that the involvement of the centralized virtualization layer in CPU, memory and I/O device virtualization causes uncertainties in VM resource allocation [20]. Rogue applications may deprive the resources in the hypervisor and incur significant performance degradation to other applications. For some applications, their performance is dependent on the characteristics of co-hosting VMs. For example, the actual CPU performance relies on the memory intensity of co-running workloads that share the last-level cache and the actual disk performance depends on the sequentiality of other jobs. Thus, application models obtained offline are likely to be inaccurate online with interference from other applications.

2.2.2 The non-uniformity of cloud resources

In this subsection, we show that even without resource contention from others, cloud users may still see variations in the capacity of their applications due to the inherent non-uniformity of cloud hardware.

CPU. In [3], the authors showed that time-sharing of CPU resources in multiple VMs can provide much more predictable performance than I/O sharing. As the number of cores embedded on a single socket increases, heterogeneous CPU architecture and on-chip hardware hyperthreading has gained popularity in modern CPU design. Despite their benefits in low energy consumption and good scalability, these designs lead to non-uniform CPU performance. "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 non-uniformities in VM CPU scheduling. Thus, even allocated with the same amount of CPU time, the actual CPU capacity of a VM changes over time. In Figure 1, we draw the performance differences of a set of micro and macro-benchmarks due to the underlying non-uniformity of cloud resources. The testbed was a Dell server with 12 CPU cores with hyperthreading enabled.

1) **Micro-benchmark**. We used a simple matrix multiplication program and measured its performance in terms of cycles per instruction (CPI). We tested with the program running alone (i.e., non-HT) and with a infinite loop keeping the sibling core busy (i.e., HT).
2) **Macro-benchmark**. We used a MapReduce job that classifies approximately 20000 documents into 20 newsgroups using Bayes Networks and measured its execution time. Since the benchmark is a multi-threaded program, *HT* allows threads to be scheduled on hyperthreads while *non-HT* ensures that no hyperthreads are used.

Figure 1(a) and Figure 1(b) show that the non-uniformity of CPU performance results in 40% and 38% performance differences in the micro and macro-benchmarks, respectively.

Memory. With the advances in multicore architecture, memory access also becomes non-uniform. New multicore systems increasingly use the non-uniform memory access (NUMA) architecture for scalability considerations. In such systems, a processor can access its own memory (i.e., local memory) or other processors' memory (i.e., remote memory) through high-speed processor interconnects. As applications become more memory-hungry, VMs with tens of Gigabyte memory is not rare. Application memory is inevitably composed of a mix of local and remote memory making the throughput and latency dependent on access locations. We used the following benchmarks to show how much the non-uniform memory access can affect application performance on a Dell NUMA machine:

1) **Micro-benchmark**. We used the STREAM [28] benchmark to measure the absolute memory bandwidth of local and remote memory.
2) **Macro-benchmark**. We selected one of the most memory-intensive programs, i.e. `mcf`, in the SPEC-CPU2006 [27] benchmark and measured its execution time on local and remote memory.

Figure 1(a) shows that there is 14% discrepancy in peak bandwidth of local and remote memory. This causes an execution time difference of 23% to `mcf` in Figure 1(b).

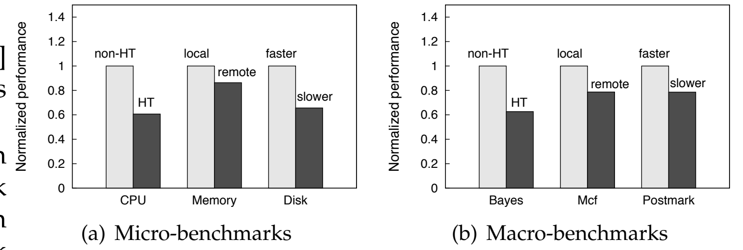
Disk. Disk access has been non-uniform since the introduction of storage techniques such as Zone-Bit-Recording (ZBR). It stores more sectors per track on outer tracks than inner tracks (i.e., slower), resulting in lower latency and higher throughput in the outer tracks (i.e., faster). As a consequence, VMs configured with the same size of virtual disks may see different disk performance. We created two VMs each with a 10GB virtual disk and put the disk image files on two partitions that are 200GB away from each other. We ran the following benchmarks:

1) **Micro-benchmark**. We ran the command `dd` inside VMs to measure the absolute disk throughput

when sequentially reading 2GB data.

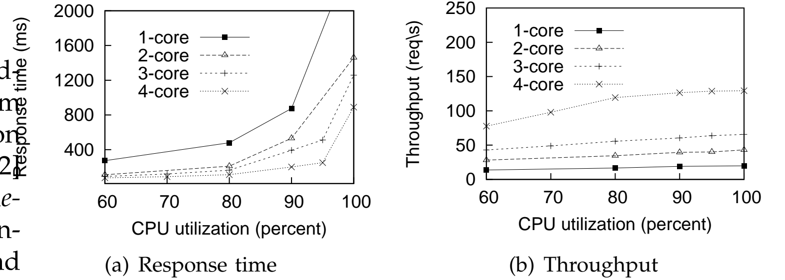
2) **Macro-benchmark**. We used the PostMark [26] benchmark and measured its performance in terms of execution time.

As shown in Figure 1(a), the different locations on the disk incurred 35% performance variations on peak throughput. Although the discrepancies in storage can sometimes be hidden by computation, macro-benchmark PostMark still shows a non-negligible performance difference (22% in Figure 1(b)).



(a) Micro-benchmarks (b) Macro-benchmarks

Fig. 1. The non-uniformity of cloud resources.



(a) Response time (b) Throughput

Fig. 2. Different resource to performance relationship due to dynamic capacity.

2.2.3 Opportunistic use of variable resources

Besides uncertainties from the underlying cloud hardware, dynamics in VMs' capacity can also come from market-based accesses to additional resources. 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. 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) would be a cost-effective way for time-varying workloads.

Next, we study how the non-determinism in capacity can pose challenges in modeling resource to application performance. Figure 2 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 equivalent to a capacity of 4-core. As shown in Figure 2, the relationship between application performance and CPU utilization changes with capacity. When CPU utilization is 80%, both response time (Figure 2(a)) and throughput (Figure 2(b)) show as large as 150% variations. 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 will suffer poor performance.

In summary, these challenges motivated us to develop a model-free and online resource control method that deals with complex resource to performance relationship and dynamic capacity. We propose a novel fuzzy control-based framework, namely DynaQoS, for the management of VM resources.

3 THE DYNAQOS FRAMEWORK

In this section, we present the design of DynaQoS, a prototype of fuzzy control-based resource allocation for cloud applications. For a specific cloud application, DynaQoS dynamically allocates the constrained resource to meet a set of predefined control objectives.

3.1 Design of DynaQoS

As shown in Figure 3, DynaQoS is composed of two layers of controllers. The first layer is a group of self-tuning fuzzy 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 metric, we show an example of such metrics in Section 5. In a cloud environment, more interesting control can be the control of current 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 [10]. The STFC takes the difference between the reference value and the achieved one as well as the change of the error as its inputs 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 resource request. The aggregation of individual requests is based on the weights (gain) of each STFC when determining the final request. The gains are dynamically adjusted according to the control errors 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

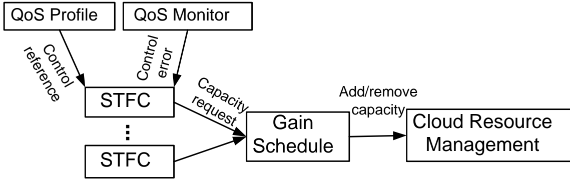


Fig. 3. The structure of the DynaQoS framework.

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.

3.2 The Self-tuning Fuzzy Controller

Due to workload and cloud dynamics, the relationship between allocated capacity and the 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 non-determinism 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 resources and the high-level objectives [7].

Figure 4 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 fuzzy controller implements a static fuzzy control logic and the scaling factor controller together with the output amplifier makes the basic controller adaptive to dynamic server capacity.

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 embedded 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 $[-1, 1]$:

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

Based on these, the controller calculates resource adjustment $\Delta u(k)$ for the next control interval. The 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 of quantified control knowledge about how to adjust the allocation according to $e(k)$ and $\Delta e(k)$. The fuzzification interface converts controller inputs into

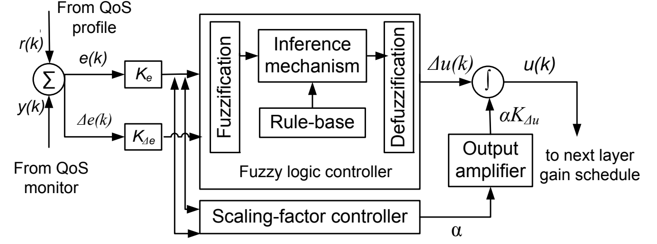


Fig. 4. The structure of the STFC.

TABLE 1
The description of linguistic values.

Linguistic value	Description
NL	negative large
NM	negative medium
NS	negative small
ZE	zero
PS	positive small
PM	positive medium
PL	positive large

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 output. 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

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

3.2.1 *Design of the rule base*

The design objective is to translate human expert's knowledge into a set of rules to control the resource allocation. 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, NM, NS, ZE, PS, PM, and PL. Their meanings are shown in Table 1. They indicate the sign and the size in relation to the other linguistic values.

Figure 5(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 $\Delta e(k)$. That is, in Zone 1, $e(k)$ is positive and $\Delta e(k)$ is negative; in Zone 3, $e(k)$ is negative and $\Delta e(k)$ is positive. In these two zones, it can be observed that the error is self-correcting and the achieved value is moving toward the reference value. Thus, $\Delta u(k)$ needs to be set to keep current trend.

Zone 2 and 4 are characterized with the same signs of $e(k)$ and $\Delta e(k)$. That is, in Zone 2, $e(k)$ is negative and $\Delta e(k)$ is negative; in Zone 4, $e(k)$ is positive and $\Delta e(k)$ is positive. Different from Zone 1 and Zone 3, in these two zones, 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 5(b). For example, when "$e(k)$" and "$\Delta e(k)$" are NL and PS, "$\Delta u(k)$" is set to PM.

3.2.2 Fuzzification, inference and defuzzification

At the heart of a fuzzy controller are the membership functions that quantify the *certainty* (between 0 and 1) that an input fall in the corresponding ranges. We select the "triangle" membership function, which is the most widely used in practice. We set the width and height of the "triangle" membership function to be 2/3 and 1, respectively. See our previous work [34] for design details of the membership function. The fuzzification component translates the inputs into corresponding certainty in numeric values of the membership functions. Let $\mu_m(e(k))$ denote the certainty of $e(k)$ of the mth membership function, and $\mu_n(\Delta e(k))$ the certainty of $\Delta e(k)$ of the nth membership function.

The inference mechanism is to determine which rules should be activated and what are the conclusions. Let $\mu(m,n)$ denote the certainty of $rule(m,n)$. The *and* operation in the premise is calculated via *minimum*:

$$\mu(m,n) = \min\{\mu_m(e(k)), \mu_n(\Delta e(k))\}.$$

Based on the outputs of the inference mechanism, the defuzzification component calculates the fuzzy controller output, which is a combination of multiple control rules, using "center average" method. Let $b(m,n)$ denote the center of membership function of the consequent of $rule(m,n)$. In this case, it is where the membership function reaches its peak. The fuzzy control output is

$$\Delta u(k) = \frac{\sum_{m,n} b(m,n) \cdot \mu(m,n)}{\sum_{m,n} \mu(m,n)}.$$

3.2.3 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 issues need to be addressed:

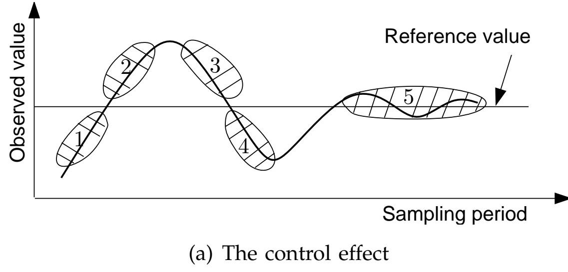
1) When there are abrupt workload or capacity changes, the control should be responsive enough to correct the resource discrepancy within a small number of steps.
2) When there are considerable fluctuations in the control effect, it may be due to two reasons. The fluctuations may come from the inaccuracies of the controller that incurs control overshooting; or it may be due to the process delay [23] of the resource allocation. A process delay is the time between the resource allocation and the actual adjustment effect can be reflected 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 factor α and the output amplifier $K_{\Delta u(k)}$ together determine the magnitude of the allocation adjustment. In our previous work [34], 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 [34] 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 or the current memory size. 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 the $K_e e(k)$ and $K_{\Delta e}\Delta e(k)$, K_e and $K_{\Delta e}$ together are able to determine which membership functions or control rules are to be activated. As shown in Figure 5(b), small values of K_e and $K_{\Delta e}$ tend to activate rules in the center of the rule table, such as the rules in Zone 5; large values are likely to trigger rules such as PL and NL. Observations in the control of real plants suggest that it is often desirable to decrease the



(a) The control effect

"$\Delta u(k)$"		"$\Delta e(k)$"						
		NL	NM	NS	ZE	PS	PM	PL
"$e(k)$"	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. 5. Design of the fuzzy control rules.

Algorithm 1 The gain scheduling algorithm.

1: **Input** $\Delta u_1(k), \ldots, \Delta u_n(k)$, $e_1(k), \ldots, e_n(k)$;
2: **Output** $\Delta u(k)$;
3: **Initialize** $w_1, \ldots, w_n$ to all zeros;
4: **for** n iterations **do**
5: $\quad w_i = \frac{|e_i(k)|}{\sum_{j=1}^{n} |e_j(k)|}$;
6: **end for**
7: $\Delta u(k) = \sum_{i=1}^{n} \Delta u_i(k) \cdot w_i$;
8: return $\Delta u(k)$;

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 information into consideration. In Figure 5(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 stabilizes, if the achieved control deviates from the reference value, K_e will quickly restores to a larger value by 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 helps mitigate process delays by generating a sequence of small or zero actuations for more stable control.

3.3 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. 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. Algorithm 1 takes individual STFCs' outputs $\Delta u_1(k), \ldots, \Delta u_n(k)$ and the corresponding errors $e_1(k), \ldots, e_n(k)$ as inputs and outputs the synthesized adjustment $\Delta u(k)$. We assume that there always exists a control solution for the multiple-objective control problem. The weights are dynamically changed based on individual STFCs' control errors. In the extreme case, the multiple-objective control degrades to a single-objective control, if one objective generates near zero errors.

3.4 Realizing service differentiation

Service differentiation is desirable when the aggregated resource demand of multiple service classes is beyond the limit of available 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 exceed 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. Finally, complications in cloud resource scheduling and performance interference also contribute to the variation of capacity. For example, results in Figure 1(b) show up to 38% variations in application performance due to scheduling dynamics; the authors in [36] also demonstrated possible CPU cycle stealing between cloud users.

To provide QoS guarantees, we consider 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 will not be starved for resources by 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 of tracking 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 capacity is limited, the basic class would release the resource first.

3.5 Scalability considerations

The scalability of DynaQoS depends on the complexity of the fuzzy control problem and the scale of the cloud application. To limit the control complexity, we confine the problem to controlling a single bottleneck resource and leave the identification of the constrained resource to cloud users. In practice, the number of control objectives and number of service classes are also limited. As will be shown in Section 5.2 and Section 5.3 that DynaQoS works effectively with two control objectives and two classes. Therefore, by carefully defining the control problem, we avoid the scalability issues.

DynaQoS assumes that there always exists a front-end node in cloud applications, from which it queries information for making control decisions. While the collection of such information is dependent on the actual implementation of the cloud application, DynaQoS decides how resources are distributed to VMs. For scalability, DynaQoS applies the resource adjustment to a cloud application by uniformly adjusting resources on individual VMs. Given the number of online VMs, DynaQoS calculates the resource allocation on each VM and multicasts the request to the servers hosting these VMs. This design simplifies the interactions between DynaQoS and the VMs that receive resources, thus improves scalability.

4 System Implementation

4.1 Cloud applications

TPC-W is an E-Commerce benchmark that models after an online book store. 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. We empirically determined that the DB tier was the bottleneck tier under the browsing workload. To generate dynamic workloads with varying CPU demands, we modified the TPC-W client generator to throw different number of requests at different times. The performance of TPC-W was measured by the request response time. We configured the front-end VM with 8 core and 4 GB memory and the DB VMs each with 2 GB memory. DynaQoS managed the CPU allocation to the DB VMs.

Memcached is a in-memory key-value store. It caches data objects such as results from database calls and page rendering. Its performance depends heavily on the size of memory it uses for caching objects. We configured Memcached (version 1.4.10) to remove cached data if memory is exhausted. When VM memory size is not large enough, cached data may be retrieved from disk storage resulting in degraded performance. We wrote a Memcached client emulator to generate random `GET` (i.e., the retrieve operation) requests with configurable working set size (WSS). The larger the WSS, the more memory is required, and vice versa. Before the retrieval of objects, we populated Memcached with approximately 1GB data. We measured Memcached's performance by its throughput (i.e., request per second). We used a single VM with 4 cores to host Memcached and DynaQoS managed the memory allocation to the VM.

Darwin streaming server is a multimedia server that streams hinted video contents using various protocols. Depending on the size of the video files, the streaming workload can be either CPU-intensive or disk I/O-intensive. We configured the Darwin server to stream 15 Quicktime movies with a total size of 30GB. Thus, its performance is bottlenecked by disk bandwidth. We used the `StreamingLoadTool` shipped with the Darwin server as the client emulator and modified it to generate dynamic workload by periodically changing the number of active movie sessions. The more sessions, the more disk bandwidth is needed. We measured Darwin server's performance in terms of client-side movie play rate (i.e., KBit per second). The Darwin VM was set to 4 core and 2 GB memory. The disk bandwidth was controlled by DynaQoS.

4.2 Testbed

Our testbed consists of virtual hosts, multiple clients, and a NFS server. 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 in a total capacity of 24 logical CPUs. The front-end and back-end DB VMs were hosted on separate machines. We used a number of client machines each with 8 cores and 8 GB memory to generate workloads for TPC-W, Memcached, and Darwin streaming server. The NFS server used a RAID5 partition to serve the VM disk images. We used Xen version 4.0.2 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 with a Gigabit Ethernet.

4.3 Implementation of DynaQoS

QoS monitor. We measured application-level performance at the client side of each cloud application. Generally, we modified the workload generators to maintain logs of finished requests. We wrote utility programs to parse the logs and calculate the average performance for

every control interval. We used response time, throughput, and play rate as the control outputs for TPC-W, Memcached, and Darwin streaming server, respectively.

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 control outputs 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 needed. For service differentiation in Section 5.3, we only considered the service differentiation in TPC-W and two classes: *Premium* and *Basic*. They both have three levels of SLO specified in terms of response times, {1s, 5s, 10s}, and with adaptation thresholds: 10 and 30 violations, respectively.

Self-tuning fuzzy controller. STFC has been implemented as a set of 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.

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. 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.

Memory size of the Memcached VM is adjusted by the Xen privilege management interface `xm mem-set` at a granularity of 1 MB. To avoid the out of memory (OOM) error, we set the minimum memory size to be 256 MB. For I/O bandwidth of the Darwin VM, we used command `lsof` to correlate the VMs virtual disks to processes and change the corresponding processes' bandwidth allocation via the Linux device mapper driver `dm-ioband` [25].

5 Experimental Results

5.1 Comparing STFC to other popular control methods

We design experiments to study the efficacy of DynaQoS in resource allocation under both static and dynamic workloads. For comparison, we have also implemented three popular controllers within the DynaQoS framework:

Kalman filer [8] 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 [8] to track the utilization of CPU and allocate CPU resources correspondingly to maintain the utilization to a specified value.

Adaptive proportional integral (PI) [19] controller directly tracks the error of the measured performance and adjusts resource allocations 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) [18] builds a local linear relationship between resource allocation and application performance with recently collected samples. If application performance deviates from the target value, the controller computes the allocation that corrects the error based on the obtained model. We empirically determine that a second-order ARMA model with a window size of 20 generates the most control accuracy.

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

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

where $r(k)$ is the control objective and $e(k)$ is the error. The smaller the $R(e)$, the more the measured output 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 controllers as:

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

Figure 6(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. We selected the set point of the controllers to be 1 second except that we followed [8] and set the Kalman filter's set point to be 90% CPU utilization, which translates to approximately a 1-second response time under the capacity of 16 cores. The virtual cluster had 4 DB VMs each with 4 VCPUs and its initial capacity was set to 6 cores (a cap of 600). From Figure 6(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.

For the control of memory size and disk bandwidth, we used similar controller settings for STFC, Adaptive PI, and ARMA. As for Kalman filter, it is hard to set thresholds for disk bandwidth and memory utilizations because these concepts are usually ill-defined. We set Kalman filter to directly work with application-level performance. The set points of Darwin server and Memcached were set to 120 KBit/s and 800 req/s, respectively. Figure 6(b) shows that STFC outperformed all controllers by successfully maintaining the play rate close to the target. The closest competitor in this test was

also Adaptive PI because it is similar to STFC in nature. It is expected that, in Figure 6(c), all the controllers have much larger deviations from the target. The reason is that when memory is insufficient, the guest operating systems incur considerable background disk swapping which can last several control intervals. This delayed effect spanning over multiple control intervals affect controllers' performance significantly.

We are also interested in the adaptability of the controllers under dynamic workloads. Specifically, we study how the controllers perform in response to SLO violations and if they can maintain resource utilization at high levels. Figure 7, Figure 8, and Figure 9 plot the performance and resource allocations of TPC-W, Darwin streaming server, and Memcached in a 90-minute period, respectively. The intensities of the workloads were increased every 30 intervals (i.e., 15 minutes) with the period between the $60th$ and the $90th$ interval being the traffic peak. After the peak, the workloads dropped every 30 intervals until reaching the original traffic levels.

For the TPC-W benchmark, as shown in Figure 7, ARMA performed worst among the controllers with a large number of SLO violations. Kalman filter was not responsive to workload changes and failed to bring the response time back to the target before the workload changed again. Both of STFC and adaptive-PI successfully maintained the response times around the target. Figure 7 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. An examination of the CPU allocations reveals that Kalman filter was not responsive to workload changes and ARMA was too sensitive to the dynamics. It confirms that controllers based on local model approximation impose limitations on how fast workloads can change. Both STFC and Adaptive PI effectively expanded the virtual cluster as traffic increased and shrunk it when the spike left, resulting in high utilizations.

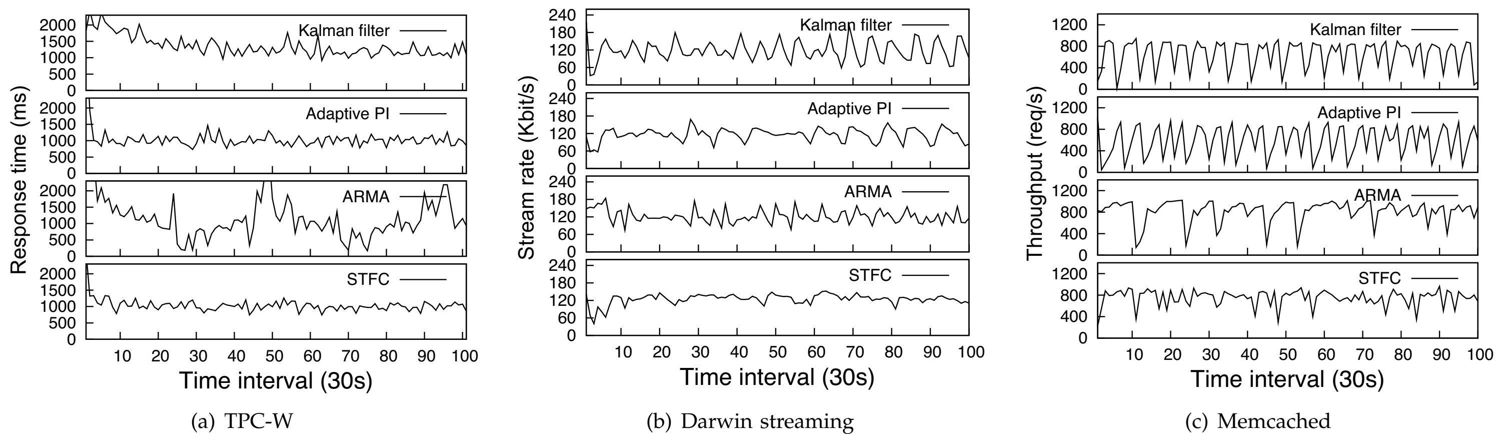
For the Darwin streaming server and Memcached workloads, as shown in Figure 8 and Figure 9, Kalman filter and ARMA had expected poor performance. Both STFC and adaptive-PI do not assume any models of underlying system, and were able to adjust the disk bandwidth and memory properly. In Figure 8 and Figure 9, we also find that STFC maintains more stable allocations during the period between the workload changes. This explains the more stable control performance of STFC in these two tests, especially in the Memcached workload. We attribute the more stable control of STFC to the flexible rule selection mechanism as it triggers control actuations with small or zero magnitude during oscillations. In Figure 10, we quantitatively compare STFC with other three controllers using the *PerfDiff* metric. In all three tests, STFC outperformed the closest competitor by 17% (TPC-W, Adaptive PI) and 37% (TPC-W, Adaptive PI) in static and dynamic workloads, respectively. In workloads that are more nonlinear (e.g., Memcached), STFC is more advantageous defeating other controllers by at least 28% and 61% in static and dynamic workloads.

5.2 Scheduling multiple objectives

In the previous experiment, we set the control objectives precisely at predefined values. In many problems, relaxing the "point" objectives to some suboptimal "regions" is also acceptable. This observation makes simultaneous control of multiple objectives possible. DynaQoS applies gain scheduling to balance the trade-off between conflicting objectives. In this experiment, we study the simultaneous control of two objectives, application-level performance and power within the DynaQoS framework. Since memory and disk allocation has little effect on the power consumption, we focus on the control of the TPC-W benchmark for its CPU allocation.

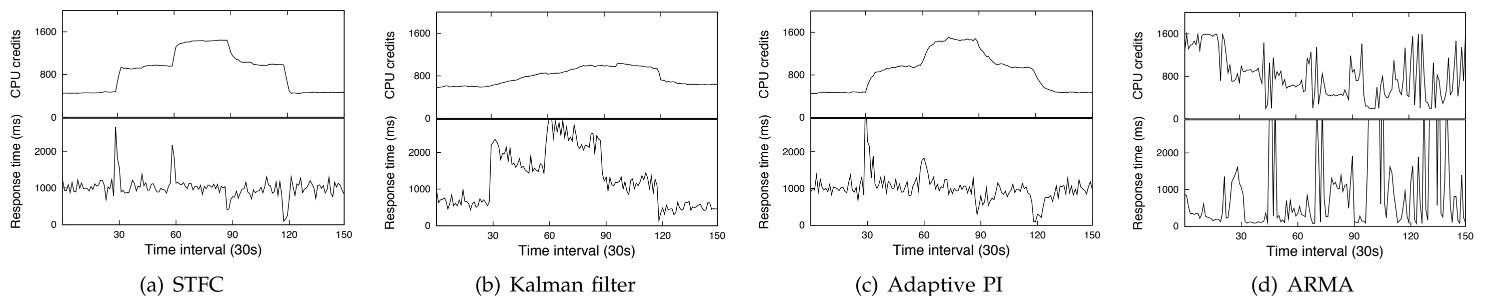
We assume that individual cloud users are allocated power budgets to limit the power consumption of their applications. We approximate power budgeting for each cloud application by dedicating the physical host to one user and measuring the system-wide power. There are existing work focusing on the VM-level power measurement [10] and the techniques can be used by DynaQoS to support multiple user power budgeting. 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 CPU resource is used, the the larger the power consumption is. The set points were set to 1 second and 250 watt for response time and power, respectively.

Figure 11 plots the response time and power consumption during the control. Before the $30th$ interval, there existed a balance point between application performance and power consumption. DynaQoS successfully identified the balance point and stabilize the response time and power consumption at approximately $800ms$ and $190w$, respectively. Starting from 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 Figure 11, we can see that the combined power consumption immediately exceeded the budget. 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 gave more weight to the corresponding STFC. With the settings of the objectives, DynaQoS successfully brought response time and power close to their targets with stable performance. Once background jobs ended, DynaQoS returned back to the ($800ms$, $190w$) balance point.



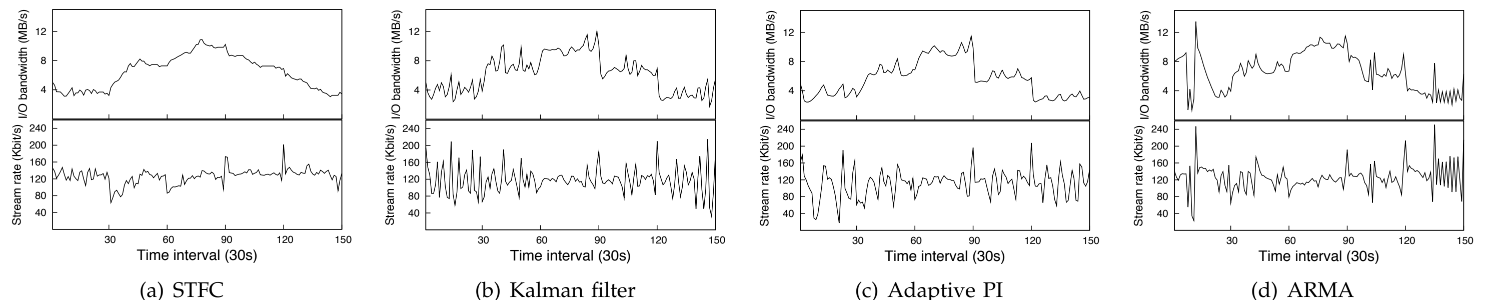
(a) TPC-W (b) Darwin streaming (c) Memcached

Fig. 6. Performance comparison of STFC, Kalman filer, Adaptive PI and ARMA under static workloads.



(a) STFC (b) Kalman filter (c) Adaptive PI (d) ARMA

Fig. 7. Performance comparison of STFC, Kalman filer, Adaptive PI and ARMA under dynamic TPC-W workloads.



(a) STFC (b) Kalman filter (c) Adaptive PI (d) ARMA

Fig. 8. Performance comparison of STFC, Kalman filer, Adaptive PI and ARMA under dynamic Darwin streaming workloads.

5.3 Service differentiation

In this section, we investigate the effectiveness of DynaQoS in providing differentiated services to two client classes, *Premium* and *Basic*. Due to limitation of space, we only study service differentiation in TPC-W workload. We dedicated a virtual cluster to each of the client class, and adjusted the CPU resource to realize differentiated services. In TPC-W, 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 DB clusters. The tomcat balancers dispatch the requests within the virtual cluster in a round-robin manner.

DynaQoS applies target adaptation if resource contention is detected. We compare the target adaptation of DynaQoS to a strict differentiation policy (STRICT), which guarantees CPU allocation of the premium class and provides best-effort service to the basic class. To prevent resource starvation of the basic class, we reserved 2-core's capacity to the basic class. 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. The client concurrency levels were set to 200 browsing clients for both classes resulting 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 total demand is

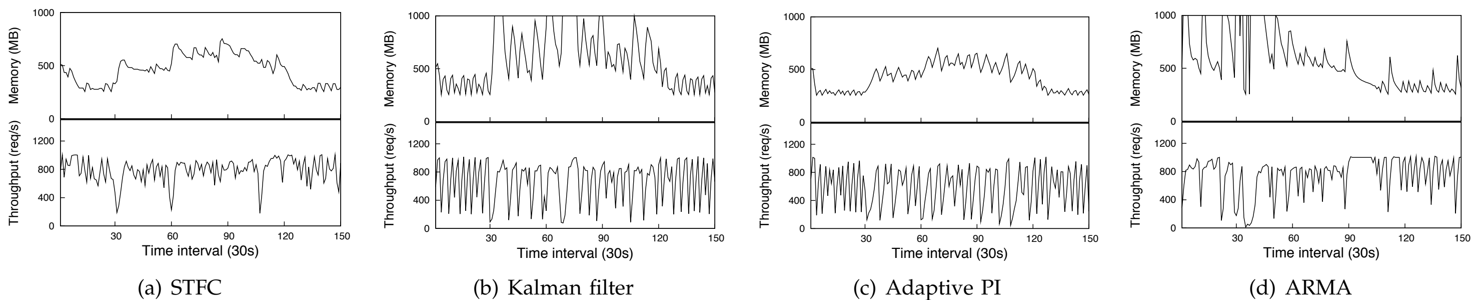


Fig. 9. Performance comparison of STFC, Kalman filer, Adaptive PI and ARMA under dynamic Memcached workloads.

below the physical capacity. We emulated a 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 possible CPU usage of the two classes from 20 to 12 CPUs. As discussed in Section 2.2.2, the capacity change due to the hardware non-uniformity is possible in current cloud platforms. More importantly, the change in the actual CPU capacity does not reflect in nominal CPU allocations, which poses challenges for service differentiation.

In Figure 12, we compare DynaQoS with the STRICT policy. We implemented two variations of the STRICT policy, one with the knowledge of the new capacity, one without. As shown in Figure 12(a), DynaQoS was able to detect the resource contention at the $32th$ interval because the premium class had saw 10 serious violations in response time. It triggered the target adaptation of the basic class 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. Without the knowledge of the new capacity, the STRICT policy allocates CPU resources according to the nominal capacity, 24 CPU in the host. As shown in Figure 12(b), the lack of actual capacity information resulted in no service differentiation to the premium class.

In Figure 12(c), we observe that with the new capacity information (i.e., 12 CPU), STRICT was able to guarantee the performance of the premium class. But the basic class suffered a 10 second response time compared with 5 second in DynaQoS. An examination of the actual CPU allocation in DynaQoS 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 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 were 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 enforces that the total allocation is not beyond 12 CPU

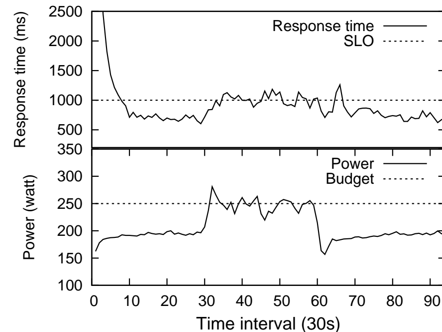


Fig. 11. Simultaneous control of performance and power.

and the basic class only got 2 CPU capacity or whatever was left by the premium class. The non-work-conserving mode in STRICT wasted some CPU time which can otherwise be used by the basic class.

5.4 Overhead

The overhead of DynaQoS comes from the computation required for making control decisions and the communication incurred due to DynaQoS in the original cloud application. The computation of fuzzy control decisions mainly consists of input fuzzification, looking up the rule table, and output defuzzification. The activation of these functions only incurred less than 5% CPU utilization in `dom0`, thus it poses negligible overhead on application performance.

When the VMs of cloud applications spread over multiple hosts, DynaQoS requires additional communication between these machines for collecting feedback information and transmitting resource allocation requests. To study the communication overhead, we deployed an instance of TPC-W with 128 DB VMs on 16 physical machines. Every control interval (i.e., 30 second), DynaQoS collected the resource utilization on each host [1] and multicasted the calculated new resource allocation. To isolate the overhead of DynaQoS from performance

1. In our testbed, we collect feedback information on application performance from client side and collect the information on power consumption from each host. To generalize the overhead study on other system implementations, where information on performance or other metrics may be collected from each host, we use the resource utilization collection overhead to approximate the general cases.

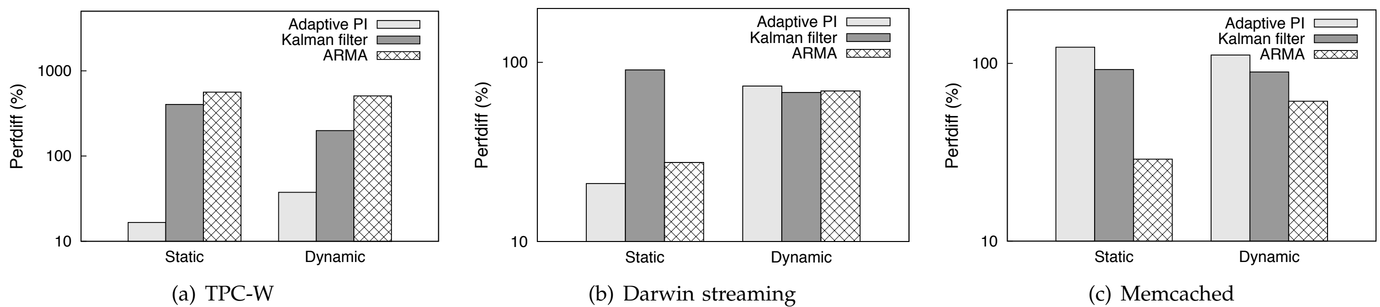


Fig. 10. Performance comparison of STFC and other controllers in relative deviation under various workloads.

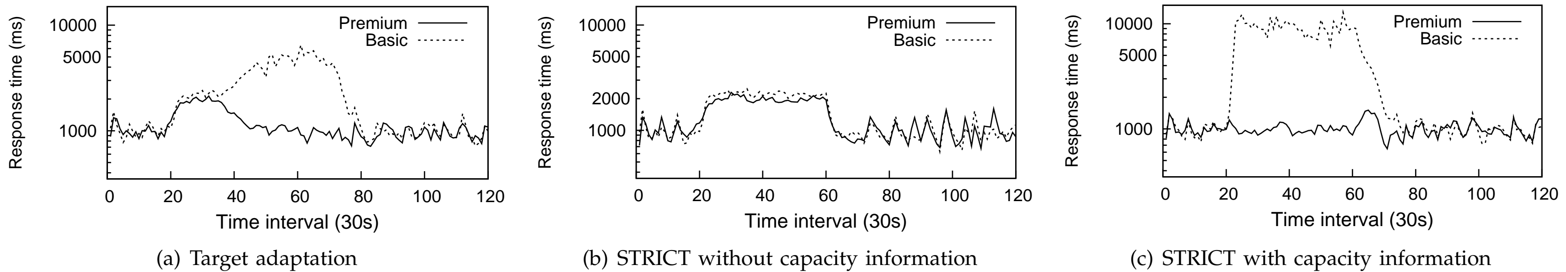


Fig. 12. Service differentiation with different policies.

degradations incurred due to inaccurate control decisions, we disabled the resource reconfiguration on each host. Experiments show that the infrequent information collection and resource request transmitting incurs no more than 2% degradation on TPC-W performance. Thus, we conclude that DynaQoS causes insignificant overhead to the original cloud applications.

6 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 [30], the authors assumed a $G/G/1$ queuing model to guide the resource allocation. 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], [16], [22]. Because the behavior of a web server changes continuously, the performance of the linear feedback control is limited. More recent work applied adaptive control [9], [11] and machine learning [21], [24] to address the issue of the lack of an accurate server model. Although these approaches provide better performance than non-adaptive feedback control approaches, they did not address the problem of process delay in resource allocation.

Our previous work [34] used an adaptive fuzzy control approach without the assumption of a server model to address the process delay in resource allocation. In this work, DynaQoS improves our eQoS work [34] in several ways. First, we introduce adaptive output amplification and flexible rule selection in STFC to explicitly address the issue of dynamic capacity. Second, DynaQoS includes a gain scheduler for the simultaneous control of multiple objectives. Finally, DynaQoS realizes service differentiation by assigning different control targets to different classes.

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. [19] proposed an adaptive proportional controller to regulate the CPU utilization to 80%. Kalyvianaki et al. [8] 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 utilization and application-level performance. DynaQoS directly regulates application performance metrics to desired values.

More work falls into the category of model-based optimization. System models are determined either by system identification [33], [32] or moving average-based local linearization [12], [31], [18]. The authors in [33], [32] obtained the model parameters by applying least squares method to offline collected data. The work in [12], [31] employed Kalman filters in the construction of an request arrival rate model. Padala et al. [18] 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. [35] adopted a fuzzy control approach to map application profiles to resource demands. The fuzzy controller is updated using clustering algorithms. Lama and Zhou [13] proposed a similar adaptive fuzzy controller by learning its structure and parameters online. Our work is different because STFC directly operates on the control error and the change of the error. Thus, it avoids the computation of fuzzy rules for adaptability. The authors in [14], [15] share the similar idea of performance and power control as ours, but we also consider the control of different resources and service differentiation. Rao et al. [20] used reinforcement learning for autonomous resource allocation with discrete steps. In contrast, DynaQoS is capable of allocating resource in a much finer granularity.

7 Conclusion

In this paper, we have proposed a novel 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 representative cloud applications show that the fuzzy controller outperformed three popular controllers for CPU, memory, and disk bandwidth allocation. DynaQoS also demonstrated its effectiveness in the simultaneous control of performance and power and service differentiation.

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] Darwin streaming server. http://dss.macosforge.org/.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

[13] P. Lama and X. Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *MASCOTS*, 2010.

[14] P. Lama and X. Zhou. PERFUME: Power and performance guarantee with fuzzy mimo control in virtualized servers. In *IWQoS*, 2011.

[15] P. Lama and X. Zhou. Ninepin: Non-invasive and energy efficient performance isolation in virtualized servers. In *DSN*, 2012.

[16] 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.

[17] Memcached. http://memcached.org/.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

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

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

[25] The dm-ioband bandwidth controller. http://sourceforge.net/apps/trac/ioband/wiki/dm-ioband.

[26] The PostMark file system benchmark. http://www.freshports.org/benchmarks/postmark/.

[27] The SPECCPU benchmark. http://www.spec.org/cpu2006/.

[28] The STREAM benchmark. http://www.cs.virginia.edu/stream/.

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

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

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

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

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

[34] 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.

[35] 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.

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

[37] 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.