Automated Control in Cloud Computing: Challenges and Opportunities

Harold C. Lim † Shivnath Babu † Jeffrey S. Chase † Sujay S. Parekh ‡
Duke University † IBM T.J. Watson Research Center ‡
Durham, NC, USA Hawthorne, NY, USA
{harold, shivnath, chase}@cs.duke.edu sujay@us.ibm.com

ABSTRACT

With advances in virtualization technology, virtual machine services offered by cloud utility providers are becoming increasingly powerful, anchoring the ecosystem of cloud services. Virtual computing services are attractive in part because they enable customers to acquire and release computing resources for guest applications adaptively in response to load surges and other dynamic behaviors. "Elastic" cloud computing APIs present a natural opportunity for feedback controllers to automate this adaptive resource provisioning, and many recent works have explored feedback control policies for a variety of network services under various assumptions.

This paper addresses the challenge of building an effective controller as a customer add-on outside of the cloud utility service itself. Such external controllers must function within the constraints of the utility service APIs. It is important to consider techniques for effective feedback control using cloud APIs, as well as how to design those APIs to enable more effective control. As one example, we explore *proportional thresholding*, a policy enhancement for feedback controllers that enables stable control across a wide range of guest cluster sizes using the coarse-grained control offered by popular virtual compute cloud services.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies, Modeling techniques, Performance attributes; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.2 [**Software Engineering**]: Management

General Terms

Management, Measurement, Performance

Keywords

Automated control, feedback control, cloud computing, data center

1. INTRODUCTION

Cloud computing derives from a long history of research and development on various approaches to IT outsourcing, in which customers draw from a utility provider's pool of capacity on a pay-as-you-go basis as an alternative to operating their own infrastructure. Some approaches target specific classes of applications. For example, the market for turnkey Application Service Providers (ASPs) is active in some sectors (e.g., *salesforce.com*), and MapReduce/Hadoop middleware is widely used for data-parallel cluster computing clouds.

This paper focuses on cloud computing infrastructure services. Typically the customer or *guest* selects or controls the software for virtual server instances obtained from one or more utility resource *providers*. The resource providers own hosting substrate resources (e.g., servers and storage), and offer them for lease to the guests. The guests may in turn use their private "slices" of leased resources to run software that provides a service to a dynamic community of clients. Virtual computing cloud infrastructure is a common and flexible example of the *utility computing* paradigm: recent offerings include Amazon Elastic Compute Cloud (EC2) [1], Aptana Cloud [3], and Joyent [6]. Advances in virtualization technologies, such as platform support (e.g., Intel's VT extensions) and hypervisor software (e.g., Xen [9] and VMware [8]), have made it easier for resource providers to adopt this paradigm and offer shared pools of hosting server resources as a service to guests.

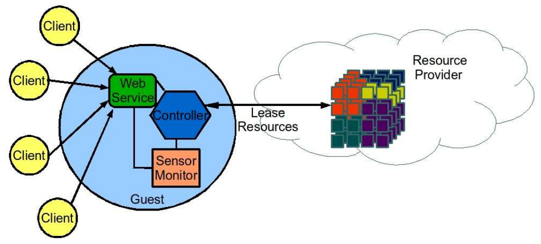


Figure 1: A guest, with a control system, running a web service on a dynamic slice of leased server resources.

Consider a simple motivating scenario in which a small startup company runs a web application service, e.g., a Tomcat [2] server cluster that serves dynamic content to clients (see Figure 1). Rather than purchasing its own infrastructure to run its service, the company leases a slice of resources from a cloud hosting provider to reduce capital and operating costs. The application is horizontally scalable: it can grow to serve higher request loads by adding more servers. The hosting provider is modeled on Amazon's EC2 service [1]. It bills its guests on a per-instance hour basis for active virtual machines, and offers an API with a fixed set of sizing choices for each virtual machine instance: small, large, and extra large. The cloud API also offers support for zones to guide the placement of VM instances in the network.

The guest company wishes to manage the resources in its slice so as to maximize the return on its investment, while handling the stream of client requests with acceptable service quality. One option is for the guest to simply lease a static set of virtual servers. This approach works only if there is a single "ideal" size for the server set: service quality degrades if the slice is underprovisioned, and the guest pays excess rent if the slice is overprovisioned.

Cloud hosting services offer an opportunity for the customer to monitor the guest application and modulate the slice resources dynamically to balance service quality and cost. The challenge is to provide a general platform and off-the-shelf feedback control policies to automate this dynamic adaptation and take advantage of the natural elasticity of shared resources in cloud computing systems. The control policies must be stable and effective under the wide range of conditions that might be encountered in practice. Adaptive resource provisioning is just one example of the need for feedback-driven application control.

One premise of this paper is that cloud customers should be empowered to operate their own dynamic controllers, outside of the cloud platform itself, or perhaps as extensions to the cloud platform. Starting from this premise, we focus on the problem of building external controllers for dynamic applications hosted on the cloud. We refer to the guest application controllers as *slice controllers*. Our perspective presents challenges for the design of both the cloud platform and the guest slice controllers. From the perspective of the cloud platform, the challenge is to export a sufficient set of well-behaved sensors and actuators to enable control policies to function effectively: Karamanolis et al. [11] suggest that system builders should focus on designing systems that are amenable to feedback control using standard off-the-shelf control policies. On the other hand, the slice controllers must make the best of the sensors and actuators built into the APIs that are actually available, and these APIs may be constrained in various ways to simplify the platform. For example, the APIs tend to provide a coarse granularity of control rather than the continuous or approximately-continuous actuators.

After discussing some of the issues raised by applying previous work in feedback-controlled services to the cloud computing context, we focus in detail on one example to illustrate the impact of the shift in perspective on the design of control policies. We outline the design of *proportional thresholding*, a control technique for systems with coarse-grained actuators and a wide range of actuator values. It modifies an integral controller by using a dynamic target range for the reference signal, instead of a target value as in previous work. The target range adjusts to changes in the accumulated actuator values. We present a preliminary evaluation of an external slice controller for a simple horizontally scalable web service. The prototype service is hosted on a cloud platform based on Automat [18] and the Open Resource Control Architecture (ORCA) [7]), a control framework for dynamic resource leasing in a shared network computing infrastructure. Experiments show that proportional thresholding enhances stability at small cluster sizes while preserving precise control at large cluster sizes.

2. FEEDBACK CONTROL IN A CLOUD COMPUTING INFRASTRUCTURE

There have been a number of works on using feedback control to meet service requirements (e.g., [14, 15, 16, 17]). However, extending their approaches to the context of cloud computing presents new challenges. This section expands on the issues for effective slice controllers in a cloud computing infrastructure.

2.1 Decoupled Control

Cloud hosting platforms export a defined service interface to their customers (guests). The interface provides a useful separation of concerns: the guest is insulated from the details of the underlying physical resources, and the provider is insulated from the details of the application.

Our position is that the controller structure should also reflect this separation of concerns in the functionalities of the controllers. A cloud hosting provider runs its own control system (a *cloud controller*) to arbitrate resource requests, select guest VM placements, and operate its infrastructure to meets its own business objectives. But *application control* should be factored out of the cloud platform and left to the guest. A clean decoupling of application control policy from the cloud platform mechanism is a necessary architectural choice to prevent cloud platforms from becoming brittle as guest demands change.

One way to facilitate this separation is for guests to select or implement their own (optional) slice controllers, outside of the cloud hosting platform. A principled layering offers the best potential for guests to innovate in their control policies and customize their controllers to the needs of their applications, and for the control architecture to scale to large numbers of diverse guests. For example, this structure is common to Amazon EC2 and Eucalyptus [4]. Both of these providers have their own control policy for arbitration, which is encapsulated from guests.

The layered approach requires a cloud hosting API that is sufficiently rich to support these interacting controllers. This separation implies that the guest slice controllers function independently of one another. Moreover, the cloud controller functions without direct knowledge of the application performance metrics, or the impact of allocation and placement choices on the service quality of the guests: it must obtain any knowledge it requires from the slice controller through the cloud hosting API. It is an open question how advanced control policies should interoperate and cooperate across the platform boundary. Note that these controllers are self-interested and are not mutually trusting: the interacting control loops have the structure of an economic negotiation.

The ORCA interface is based on resource lease contracts whose terms are negotiated and expressed through exchanges of property lists. Each lease object is valid for a stated term (interval), and may be renegotiated and extended before it expires. The lease terms define the control interval for active resources (e.g., virtual servers), although additional resources may be leased for the slice at any time. The property lists define the parameters of the negotiation: their structure and interpretation must be agreed by the interacting controllers.

Many of the previous works on feedback-controlled adaptive resource provisioning assume a central controller that combines application control and arbitration policy (e.g., [15, 17, 16]). Urgaonkar et al. [17] use queueing theory to model a multi-tier application and to determine the amount of physical resources needed by the application. Soundararajan et al. [16] present control policies for dynamic provisioning of a database server. These approaches do not transfer directly to cloud environments with decoupled control. Padala et al. [14] is suitable for decoupled control, but requires adjustment for coarse-grained actuators, as discussed below.

2.2 Control Granularity

The control API for the cloud hosting platform is a "tussle boundary". There may be a gap between the sensors and actuators desired by slice controllers and those exposed by the resource providers. Providers may hide useful information to preserve their flexibility, or hide power to simplify the operation of the cloud controller policy and the underlying hosting mechanisms. The slice controller must make the best of whatever sensors and actuators are available, and whatever control intervals and granularity the provider allows.

For example, in a virtualized environment, resource providers may choose not to export the access to hypervisor-level actuators of the cloud computing infrastructure, such as controlling the CPU and memory allocations of a virtual machine and the location of the physical host of a virtual machine. EC2 and Eucalyptus discretize into a small range of predefined sizes, and do not expose these fine-grained actuators.

It is an open question what granularity is required for effective control. Many previous works on feedback control for Internet services have focused on an integrated control loop with fine-grained access to the sensors and actuators of the underlying virtualization platform on a single server. For horizontally scalable clusters, it is necessary to dampen the control loop at small cluster sizes, when the control granularity is coarse relative to the allocated resource (accumulated actuator value). Section 3 discusses a *proportional thresholding* technique for slice controllers to function with the coarse-grained control that is typical of current cloud platforms.

2.3 Other API Constraints

The experiments with proportional thresholding in this paper assume a simple clustered web service. Content is generated by a CPU-bottlenecked middle tier (a Tomcat cluster), with a front-end Apache server that balances request load across the middle tier. The guest runs the Hyperic HQ [5] monitoring system within its slice to obtain a feedback signal of CPU utilization, which is presumed to be accurate. The feedback signal drives the control policy, which modulates the number of virtual machines in the middle tier with knowledge that the middle tier is automatically *request-balanced*.

More realistic scenarios raise additional issues for the cloud platform API.

Sensors.

Since the underlying resources are owned by the resource provider, the slice controller depends on the cloud platform for accurate sensor measurements at the resource level. Choices made by the cloud provider may have unintended effects on the suitability of the measurements for a feedback control policy. For example, consider a virtual machine service based on a work-conserving proportional share CPU scheduler in the hypervisors. CPU utilizations may be reported as a percentage of the VM's assigned share (entitlement), or as a percentage of the CPU resource actually available to it, which may be more than the assigned share if the physical node has surplus resources. A choice made by the hosting provider for more efficient resource usage (a work-conserving scheduler) may result in a noisy sensor measurement.

One solution is for the cloud platform to expose well-specified sensors that are suitable for stable control. Another solution is to design slice controllers that are robust to noisy sensor measurements and/or that rely on application-level measures they can access directly (e.g., request load level, queue lengths). Hyperic HQ provides a range of application-level measures for various applications. This paper uses CPU utilization reported from the Xen hypervisor with the CPU allocation bounded at the share size.

Actuators.

Horizontally scalable network services use a variety of means to direct the flow of requests to servers. Request routing solutions may involve programmable network elements installed by the hosting provider. Cloud platform APIs will need to expose suitable actuators for the slice controller policy to configure and adapt request routing or other programmable network elements.

There are many related research questions involving internal sensor/actuator constraints.

- How much internal knowledge does the slice controller need, in order to be effective?
- How much internal control needs to be exposed for the slice controller to be effective?
- How do we integrate the internal guest constraints to the guest control policy?
  - In a multi-tier application, how do we design an effective control that deals with tier interactions?
  - How do we factor into the control policy the network elements and constraints of the guest service, such as request routing?
  - How do we design an effective control policy that takes into consideration stateful guest services?

3. PROPORTIONAL THRESHOLDING

This section presents proportional thresholding to illustrate the challenges of coarse-grained control and the means to address it in a slice controller for a cloud hosting platform. Our control approach is similar to Padala et al. [14], which dynamically adjusts the CPU entitlement of a virtual machine to meet Quality of Service (QoS) goals by empirically modeling the relationship of CPU entitlement and utilization to tune the parameters of an integral control. The question is how to adapt the control policy for the case when fine-grained actuators for adjusting CPU entitlements are not available, e.g., the slice controller can only request changes to the number of virtual machines in a cluster. Resizing the number of virtual machines changes the capacity of the slice in coarse discrete increments. At small cluster sizes this may cause the control system to oscillate around a target CPU utilization.

Other works have considered the oscillation problem and instead, use static thresholding for their control policy (e.g., [17, 16]). In this policy, rather than having only one target goal, the goal is turned into a *target range*, defined by a high and low threshold. Thus, the system is considered on target when the sensor measurement falls inside the target range. Urgaonkar et al. [17] use this idea to allocate physical servers for a multi-tier Internet application. Only when the observed rate differs from the predicted rate by an application-defined range does the server resize. Their policy releases resources only when they are needed by other applications, which is difficult to achieve when the application controllers and cloud controller are decoupled. Similarly, Soundararajan et al. [16] present control policies for dynamic replication of database servers based on static thresholding with a target range. Their control policies are in steady state only when the average latency of their database servers is inside the low and high threshold.

Static thresholding is simple to use, however, it may not be robust to scale. Consider the motivating scenario, mentioned in Section 1, of a guest running as a web service host. Since the Tomcat cluster is request-balanced, going from 1 to 2 machines can increase capacity by 100% but going from 100 to 101 machines increases capacity by not more than 1%. The relative effect of adding a fixed-sized resource is not constant, so using static threshold values may not be appropriate.

3.1 Design of Proportional Thresholding

Proportional thresholding addresses the problems outlined in the previous section. This control policy modifies an integral control by using a dynamic target range, instead of a single target value. Moreover, the dynamic target range decreases as the accumulated actuator values increases. *Proportional thresholding* is particularly important for sensors with large dynamic range and fixed coarse-grained actuators, such as using horizontal scaling to handle flash crowds, while maintaining a certain CPU utilization target. Specifically in our motivating scenario, the impact of a constant change in the actuator value is dependent on the current server set. For example, in a request balanced cluster, this behavior is illustrated in Figure 2, where we plot the effect of increasing the size of a Tomcat cluster to the CPU utilization while maintaining a fixed workload.

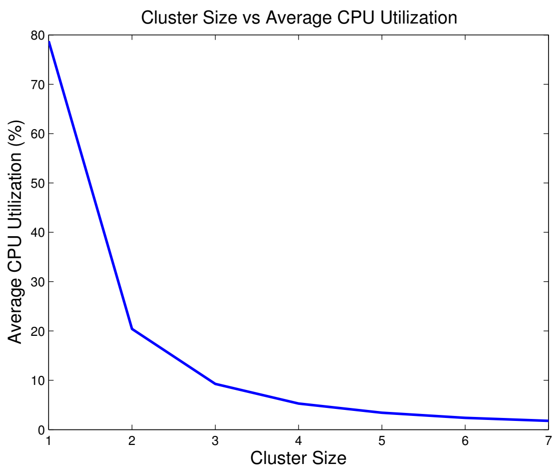


Figure 2: The effect of increasing the size of a Tomcat cluster to the CPU Utilization while maintaining a fixed workload.

Similar to Padala et al. [14], the control policy uses an integral control because it eliminates steady state errors. An integral control is defined by

$$u_{k+1} = u_k + K_i \times (y_{ref} - y_k), \tag{1}$$

where u_{k+1} is the new actuator value, u_k is the current actuator value, K_i is the integral gain parameter, y_{ref} is the target sensor measurement, and y_k is the current sensor measurement. The integral gain parameter K_i can be estimated empirically [15]. To avoid the problem of having oscillations due to the coarse-grained actuator, we then define y_h and y_l as the high and low target sensor measurements, which defines the target range. The modified integral control is as follows:

$$u_{k+1} = \begin{cases} u_k + K_i \times (y_h - y_k) & \text{if } y_h < y_k \\ u_k + K_i \times (y_l - y_k) & \text{if } y_l > y_k \\ u_k & \text{otherwise} \end{cases}. \tag{2}$$

This way, similar to static thresholding, a change in the actuator value only occurs when the sensor measurement is outside the target range. More specifically, the actuator increases value only when it is above the high target and decreases in value when it goes below the low target.

In *proportional thresholding*, the target range used by the controller should be able to change dynamically depending on the accumulated actuator values. This addresses the problem with static thresholds, which does not give an effective control, in terms of ensuring high resource utilization and meeting client demands. With static thresholds, the behavior in Figure 2 can potentially lead to poor resource utilization. It should be noted that this behavior is not restricted to horizontal scaling.

The dynamic target range should capture the property of being resource-efficient. Since the relative effect of the increment becomes finer as the number of allocated resources, the target range should narrow for more precise control as the number of allocated resources increases. This means that our modified integral control (Equation 2) should have the property of converging to the standard integral control (Equation 1) as the accumulated actuator values goes to infinity. In order to achieve this behavior and assuming we set $y_h = y_{ref}$, proportional thresholding adjusts y_l depending on the number of actuator values accumulated and at the same time have the following behavior: $\lim_{x\to\infty} y_l = y_{ref}$, where x is the accumulated actuator values. In the next section, we describe our prototype control system and how we formulate the control parameters, specifically K_i and the equation for y_l for a specific actuator by empirically modeling the behavior of an actuator and sensor measurements.

4. EXAMPLE

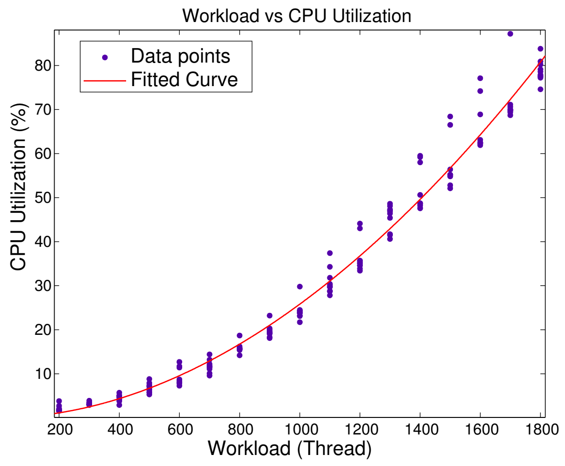


Figure 3: The CPU utilization of a Tomcat server under various workload.

4.1 Prototype Control System

Like our motivating scenario, our guest serves as a web service host by using leased virtual machines to form a Tomcat cluster. Moreover, as mentioned in Section 2, our prototype assumes that there is a front-end Apache web server that balances the distribution of requests across all cluster nodes.

We use ORCA as the underlying architecture and resource leasing mechanism. It is a software toolkit developed at Duke University that allows guests to lease resources from a resource substrate. It is also a service-oriented infrastructure and the architecture provides resource leasing abstractions for the guests. Using virtualization, ORCA enables guests to share a common pool of resources [7]. Furthermore, to emulate the API provided by Amazon EC2, we only expose similar API functionalities to the control system. We use Automat [18] for our control interface. It is a programmable hosting center toolkit integrated with ORCA that supports external slice controllers. Similar to the motivating scenario, our slice controller performs horizontal scaling. We also use Hyperic HQ to gather the CPU utilization of all leased virtual machines. The sensor measurement used by our control system is then the average CPU utilization of all allocated virtual machines, filtered by an exponential moving average filter.

4.2 Control Parameters

For the parameter K_i of our control policy, we use the value $K_i = -.07$, which is estimated offline. This value is derived by using linear regression to model the relationship between the CPU utilization and the cluster size under a synthetic heavy workload: $y_{k+1} = .8819y_k - .5892u_k$. Using the Z-transform of this model, we estimate the settling time and maximum overshoot corresponding to a range of K_i values. We then use a K_i that gives a settling time of 15.12 sample intervals and maximum overshoot of 0.0002544%.

Since the Tomcat cluster is request-balanced, we empirically measured the CPU utilization of a single machine under various workload to formulate the equation for y_l (see Figure 3). From these data, we find the best-fit curve, which in this case is

$$\texttt{CPU} = (3.869 \times 10^{-5}) \times \texttt{workload}^{1.947}. \tag{3}$$

Given a target CPU utilization, y_h, we can then use the equation to get the estimated average workload for each machine applied to the system:

$$\texttt{workload}_{est} = (\frac{y_h}{3.869 \times 10^{-5}})^{\frac{1}{1.947}}. \tag{4}$$

The idea is that we are interested in finding the lowest threshold value, such that the minimum amount of resources is used while still satisfying client demands. $\texttt{workload}_{est}$ tells us that any average workload greater than $\texttt{workload}_{est}$ will result in a CPU utilization of greater than y_h. This means that we only want to reduce the number of virtual machines if the resulting average workload is less than or equal to $\texttt{workload}_{est}$. The total workload from $\texttt{workload}_{est}$ when the number of machines is reduced by 1 is given by

$$\texttt{workload}_{tot} = \texttt{workload}_{est} \times (\texttt{currVM} - 1). \tag{5}$$

Using $\texttt{workload}_{tot}$ to solve for the average workload of the current number of machines gives the lowest average workload that is also greater than or equal to the average workload of the number of machines reduced by 1:

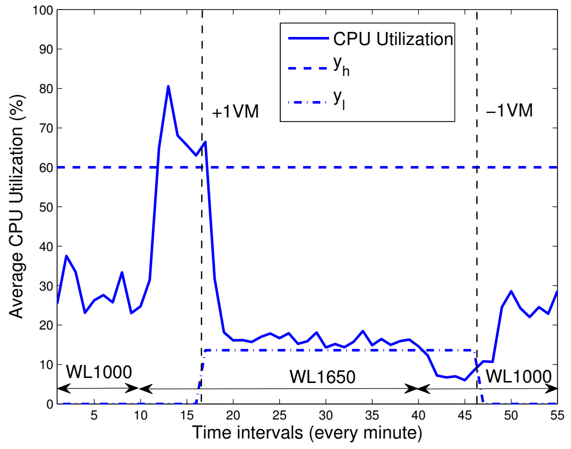
$$\texttt{workload}_{low} = \texttt{workload}_{est} \times \frac{\texttt{currVM} - 1}{\texttt{currVM}}. \tag{6}$$

We then calculate the y_l by applying $\texttt{workload}_{low}$ to Equation 3.

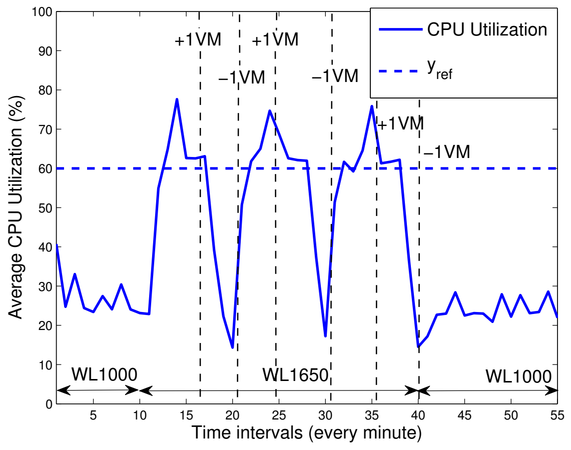
$$y_l = y_h \times (\frac{\texttt{currVM} - 1}{\texttt{currVM}})^{1.947} \tag{7}$$

Equation 7 shows that y_l converges to y_h as the number of virtual machines becomes very large. Moreover, *proportional thresholding* converges to the standard integral control (Equation 1).

In this example, we formulate the control parameter K_i and the equation for y_l offline, which depending on the type of target systems may already be enough. In our case, even though the model may turn out to be not very accurate due to changes in workload, our control system is still able to have an acceptable and stable behavior. One of the benefits of feedback control is that an accurate estimate of the models is not necessary because it can be robust to errors in the parameter estimates. For example, using the $K_i = -.07$ under a lighter workload gives a settling time of 39.96 sample intervals and maximum overshoot of $-.004345\%$, which has a longer convergence time but still results in a stable controller. Furthermore, another reason is that the spectrum of workloads used to formulate y_l encompasses a wide spectrum. As mentioned by Hellerstein et al., a good strategy for modelling systems is to start simple, which means that there is no need to develop a complex model, if a simpler model is already sufficient [10]. Nevertheless, there are target systems that need a complex and accurate model, which requires tuning the control parameters online. Although not specifically for automated control in cloud computing, there have been works that applies adaptive controllers for computing systems [13, 12]. We leave the research problem of integrating the modeling process with the control system as a future work, specifically for a cloud computing infrastructure.



(a) Proportional Thresholding



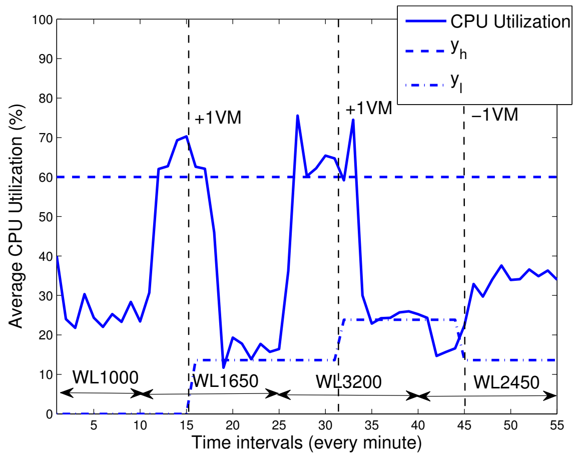
(b) Integral Control

Figure 4: Comparison between proportional thresholding and integral control.

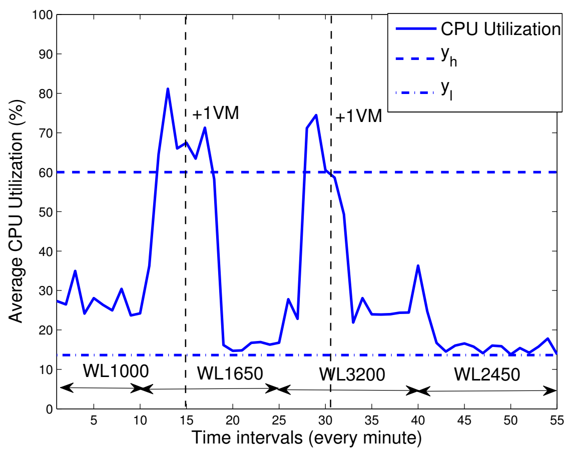
5. EVALUATION

To evaluate our prototype control system, we compared the performance of our prototype using 3 control policies under various synthetic workloads: *proportional thresholding*, static thresholding, and integral control.

In Figure 4, we conducted an experiment, where we first applied a workload of 1000 threads, then at the 10^{th} time interval, we increased the workload to 1650 threads, and finally at the 40^{th} time interval the workload goes back to 1000 threads. Figures 4(a), and 4(b) show the behavior of the controller under *proportional thresholding* and integral control, respectively. Note that both figures start with 1 allocated virtual machine. Under *proportional thresholding*, the system does not oscillate, such that the controller allocates 1 virtual machine when the workload is 1000 threads and 2 virtual



(a) Proportional Thresholding



(b) Static Thresholding

Figure 5: Comparison between proportional and static thresholding.

machines when the workload goes up to 1650 threads. In contrast, under integral control, the system oscillates between 1 and 2 virtual machines when the workload goes up to 1650 threads, which may not be desirable since it leads to short-term unpredictability.

In Figure 5, we conducted an experiment where we slowly ramp up the number of threads from 1000 to 1650 to 3200 and then finally decreasing the workload to 2450 threads. Figures 5(a), and 5(b) show the behavior of the controller under *proportional thresholding* and static thresholding, respectively. Like the previous experiment, both figures also start with 1 allocated virtual machine. Figure 5(a) also shows how y_l changes with cluster size, specifically, with 3 virtual machines, y_l has gone up to 23.84%. When the workload drops to 2450 threads, the controller is able to reduce the cluster size to 2 virtual machines and hence, conserve resources. This is in contrast to static thresholding, where the cluster size remains at 3, even though 2 virtual machines are enough to handle the workload.

6. CONCLUSION

We have presented issues that make feedback control in a cloud computing infrastructure different from feedback control of other computer systems: decoupled control, and control granularity. Moreover, we have shown why prior works related to automated control may not work, when used in a cloud computing infrastructure. We have introduced *proportional thresholding*, a new control policy that takes into account the coarse-grained actuators provided by resource providers. Using the actuator constraints similar to Amazon EC2, we have presented a prototype control system that performs better than traditional integral control and static thresholding.

Acknowledgment. This research is supported by the National Science Foundation through CNS-0509408 and CNS-0720829.

7. REFERENCES

[1] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.
[2] Apache Tomcat. `http://tomcat.apache.org/`.
[3] Aptana Cloud. `http://aptana.com/cloud/`.
[4] Eucalyptus. `http://eucalyptus.cs.ucsb.edu/`.
[5] Hyperic HQ Open Source Web Infrastructure Management Software. `http://www.hyperic.com/`.
[6] Joyent. `http://www.joyent.com/`.
[7] Open Resource Control Architecture (ORCA). `http://www.nicl.cod.cs.duke.edu/orca/`.
[8] VMware: Virtualization via Hypervisor, Virtual Machine & Server Consolidation. `http://www.vmware.com/`.
[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of SOSP*, 2003.
[10] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.
[11] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proc. of HOTOS*, 2005.
[12] M. Karlsson, C. Karamanolis, and X. Zhu. An adaptive optimal controller for non-intrusive performance differentiation in computing services. In *Proc. of ICCA*, 2005.
[13] Y. Lu, T. Abdelzaher, and G. Tao. Direct adaptive control of a web cache system. In *Proc. of American Control Conference*, 2003.
[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 *Proc. of EuroSys*, 2007.
[15] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proc. of IM*, 2002.
[16] G. Soundararajan, C. Amza, and A. Goel. Database replication policies for dynamic content applications. In *Proc. of EuroSys*, 2006.
[17] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. of ICAC*, 2005.
[18] A. Yumerefendi, P. Shivam, D. Irwin, P. Gunda, L. Grit, A. Demberel, J. Chase, and S. Babu. Towards an autonomic computing testbed. In *Proc. of HotAC*, 2007.