2018 Annual American Control Conference (ACC)
June 27–29, 2018. Wisconsin Center, Milwaukee, USA

# Performance management via MPC for Web services in cloud

Durgesh Singh[1], Joydeep Mukherjee[2], Saikrishna PS[3], Ramkrishna Pasumarthy[1] and Diwakar Krishnamurthy[2]

*Abstract*— **Web services are increasingly being deployed on cloud platforms. Due to their interactive nature, Web services need to ensure fast response times to their end users. Unfortunately, the performance of a Web service can suffer due to a sudden surge in incoming traffic. Furthermore, a cloud-based service can also incur performance degradation due to interference, i.e., contention among services in the cloud platform for shared resources. Such issues motivate the need for automated runtime performance management solutions that ensure response time goals are continuously met. This paper explores a control theoretic approach called Model Predictive Control (MPC) for realizing such a solution. MPC is based on an optimization formulation, which lends itself well to expressing multiple constraints related to response time performance and the amount of resources, e.g., number of virtual machines (VMs), available to a Web service. We outline the design and operation of an MPC controller that governs the scale out and scale in of VMs while adhering to operator-specified thresholds for mean response time and the number of VMs available. Using a realistic Web service testbed, we show that the controller is able to satisfy the specified response time constraint even when the service is subjected to workload surges and interference.**

## I. INTRODUCTION

Cloud computing has been widely adopted by large companies in the past few years. The notion of unlimited computing resources as well as the convenience offered by the pay-as-you-use model has attracted many major Web applications, such as Netflix and Pinterest, to be hosted on cloud platforms such as Amazon Web Service's (AWS) Elastic Compute Cloud (EC2) and Google Compute Engine.

One of the key elements in cloud computing is server virtualization. To realize resource usage efficiencies, a cloud provider typically co-locates multiple virtual machines (VMs) belonging to different cloud subscribers on the same physical machine (PM). Server virtualization is usually done by a software layer, implemented on the PM operating system, called the hypervisor [19]. The hypervisor assigns each VM a certain share of the processing and input-output capacity of the PM. Server virtualization can facilitate cost effectiveness and can leverage on techniques such as live VM migration to dynamically redistribute workloads.

Unfortunately, virtualization can have adverse effects on performance. Specifically, virtualized systems can suffer from a phenomenon called performance interference where VMs on a given PM can interfere with each other's performance. This occurs when VMs contend for shared PM resources such as processor cores [10], network interface [20], and physical memory [14]. Management activities initiated by the cloud provider such as scheduling or migrating a new VM on a PM can also cause background noise, which can lead to performance deterioration in VMs hosted on the PM [23]. Furthermore, performance degradation in a VM can also happen because of unexpected surges in the incoming workload. When performance degradation due to interference or workload surges is detected, additional VMs need to be provisioned to maintain desired performance.

Such performance issues can be especially challenging for cloud-based Web services where end users expect fast response times. While it is possible to over provision VMs to mitigate unpredictable response times due to interference or workload surges, such a strategy can incur unacceptably high costs. A cloud provider needs systematic runtime VM provisioning techniques that can ensure response time targets are met for a subscriber's Web service while adhering to constraints placed on the numbers of VMs that can be provisioned to that service. In particular, such a technique should be able to address the performance degradation effect of workload surges and interference in a robust manner.

Recently, control design methods based on the Model Predictive Control (MPC) concept [7] have found wide acceptance in industrial applications and academia owing to its capability of designing high performance control systems that can operate without expert intervention for long periods of time. MPC refers to an algorithm that utilizes an explicit process model to predict the future response of a given plant. It optimizes the output response of a plant over a finite horizon in an iterative manner based on a set of constraints and a given cost function. In the current context, MPC which is based on an optimization formulation, lends itself well to expressing multiple constraints related to response time performance and the number of VMs available for provisioning. MPC has also been shown to generate highly accurate control decisions in response to sudden external disturbances to the plant [11]. These factors motivate us to study the use of MPC for VM scaling in the cloud.

This paper describes how the idea of MPC can be applied for performance assurance of cloud-based Web services. Specifically, an empirical dynamical model for the cloud computing system is identified from test data. The model describes the relationship of a Web service's response time to the CPU utilization of the PM hosting the service, the workload, and the number of VMs assigned to the service.

[1]Durgesh Singh and Ramkrishna Pasumarthy are with Department of Electrical Engineering, IIT Madras, India. 257durgesh@gmail.com, ramkrishna@ee.iitm.ac.in

[2]Joydeep Mukherjee and Diwakar Krishnamurthy are with the Department of ECE, University of Calgary, Canada. jmukherj@ucalgary.ca, dkrishna@ucalgary.ca

[3]Saikrishna PS is with Department of Electrical Engineering, IIT Tirupati, India. psskrishna@iittp.ac.in

978-1-5386-5428-6/$31.00 ©2018 AACC

Since the PM CPU utilization depends on both workload surges and interference, the model captures the impact of these two characteristics on response time. The output of MPC provides the optimal, i.e., sufficient, number of VM instances to be provisioned for the incoming workload as well as interference so that the mean response time is always below a predefined value.

We use a realistic benchmark Web service system, RUBiS [4], deployed on our cloud testbed, to validate our approach. We present results that provide insights on how the length of the future horizon over which MPC performs its optimization impacts controller performance. We also show that our approach is able to handle workload surges and interference while adhering to operator specified constraints for mean response time and number of VMs. To the best of our knowledge, we are not aware of other studies which have used MPC for VM scaling in an environment characterized by interference and constraints on the number of VMs.

## II. Related Work

The traditional non-control theory approach for similar kinds of problems includes the use of queuing theory. The papers [12], [5] and [22] uses a queuing model which determines the amount of resources to be allocated. While these techniques can be effective, the kinds of models used in these studies are not meant to capture a system's transient behaviour over fine timescales.

Control theory offers a formal mechanism to manage the performance of a system by considering its dynamics at fine timescales. Several studies use classical control approaches such as PID control [8], [26]. Since vertical scaling may not be feasible in many cloud platforms, we focus on the more common approach of horizontal scaling, i.e., scaling by adding more VMs to a service. Furthermore, unlike this study, we explicitly consider the impact of interference, which is common in cloud platforms.

Beyond traditional PID controllers, others have applied more advanced control theoretic techniques for systems performance management. For example, several studies [17], [25] use fuzzy control for vertical scaling, some have used gain scheduling control [18], and minimum variance control [8]. While these techniques have been shown to be effective, they focus on vertical scaling, do not consider interference, and require considerable domain expertise to construct the underlying system models. In our earlier work [21], we used fuzzy control for horizontal scaling of virtualized Web services. However, that work did not consider interference and only used microbenchmark workloads in contrast to the more realistic RUBiS system evaluated in this paper.

Optimal control strategies [13] are a natural fit to the problem we are studying since they allow cost functions and constraints to be accommodated in an explicit manner. Nathuji et al. [15] propose a solution that combines a linear control theoretic model with an optimizer. The model captures the impact of VM resource configurations on VM response times. Similar to our work, this work leverages optimization and considers the impact of interference. However, unlike our work, it focuses on vertical scaling. Furthermore, it only focuses on optimizing the system's control over a single sampling period into the future. We show in Section VI that MPC's ability to optimize over multiple sampling periods can be advantageous.

Wang et al. [24] apply MPC, for vertical scaling of cloud services. They implement an MPC controller that employs a fuzzy model in combination with a genetic algorithm based optimizer. While this controller performs significantly good, the authors note that the control actions cannot be performed too frequently due to the overhead of the non-linear optimization. In contrast to this work, we focus on horizontal scaling and consider interference. Furthermore, due to our use of a linear model, our approach can enable more agile control decisions.

## III. Problem Formulation

### A. Model of Target System

A state-space model, which describes the complete evolution of the system over time, is constructed first. The states of the system are the mean physical core utilization over all VMs denoted as $CPU(k)$, where $k$ is the sampling instant, and the mean request response time over all VMs denoted by $RES(k)$. These are represented by a state vector $x(k) = [CPU(k) \;\; RES(k)]^\top$. They are selected since they can reflect the impact of workload fluctuations and interference. The exogenous input, i.e., disturbance, for our model is the request rate encountered by the Web service $w(k)$ measured in requests per second. The control input is the number of virtual machines active in the system denoted by $u(k)$. The output $y(k)$ which we will be interested in is the mean response time over all VMs. As the behavior of response time is stochastic and non-linear in nature, but for modeling we have linearly approximated it at higher workload region which proves to be an good approximation as it can be inferred from validation plot provided in Section V-A.

The target system dynamics in state space form is given as follows:

$$\begin{aligned} x(k+1) &= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} x(k) + \begin{bmatrix} b_{11} \\ b_{21} \end{bmatrix} u(k) + \begin{bmatrix} e_{11} \\ e_{21} \end{bmatrix} w(k) \\ y(k) &= \begin{bmatrix} 0 & 1 \end{bmatrix} x(k) \end{aligned} \quad (1)$$

In Equation 1, the matrices that weight the state, input, and exogenous input are denoted by $A$, $B$, and $E$, respectively. All entries of these matrices, i.e., $a_{ij}$, $b_{ij}$ and $e_{ij}$; $i, j \in \{1,2\}$, are to be identified using linear regression as explained in Section V-A.

*Model Assumptions:* The proposed state space model is developed with the following assumptions and considerations:

- We consider CPU bound workloads and interference for this study. Accordingly, we monitor and collect the mean PM per-core utilization imposed by each VM.
- We are interested in accurately capturing system dynamics in the operating regions corresponding to high physical core utilizations, i.e., 40% to 90%. This is because response time violations are significant in these regions.

- There exists a threshold for the rate of incoming requests above which requests cannot be served by our system using the maximum number of VMs specified. The incoming request rate is never allowed to go beyond this threshold to avoid system saturation and failure due to overload.

### B. The Control Problem

As mentioned previously, the control objective is to guarantee mean response time below a certain threshold by utilizing a minimum number of VMs. Also, there is a constraint on the maximum number of VMs for serving requests. We explore MPC due to its ability to accommodate such constraints.

The optimal control problem of MPC in a discrete time system [6] is defined as follows:

$$\begin{aligned} \min_{u} \; f(x,u) = \sum_{i=0}^{N_p-1} \Big[ & (x_i - x_{ref,i})^T Q_x (x_i - x_{ref,i}) \\ & + (u_i - u_{ref,i})^T Q_u (u_i - u_{ref,i}) \Big] \\ & + (x_{N_p} - x_{ref,N_p})^T S (x_{N_p} - x_{ref,N_p}) \end{aligned} \tag{2}$$

subject to

$$\begin{aligned} u_{min} \leq u_i \leq u_{max} \\ y_{min} \leq C x_i \leq y_{max} \end{aligned} \tag{3}$$

In Equation 2, $x_0$ is [1] the initial state value at sample 0 and is known, i.e., measured, $x_i$ is predicted state at the $i^{th}$ sample, $x_{ref,i}$ is reference value of the state at the $i^{th}$ sample, $u_i$ is the control input at the $i^{th}$ state, $u_{ref,i}$ is reference value of the control input at the $i^{th}$ state, $x_{N_p}$ is the terminal state, i.e., the predicted value of states at the last sample instant of the prediction horizon, and $x_{ref,N_P}$ is the reference value of the terminal state. $Q_x$, $Q_u$, $S$ are weighting matrices for states, control inputs and terminal state, respectively. As shown in Equation 3, constraints are placed on the maximum and minimum values of the input, i.e., number of VMs, and the output, i.e., mean response time. In Equation 3, $C$ is the output matrix that maps the state variables to the output. As shown in Equation 1, due to our choice of state variables and output, the value of this matrix is $\begin{bmatrix} 0 & 1 \end{bmatrix}$.

The $x_{ref,i}$, $u_{ref,i}$, and $x_{ref,N_p}$ values in Equation 2 are tunable parameters that influence the trajectory of the controller. The extent of influence of these parameters can be controlled through their corresponding weighting matrices. In our study, we are interested in regulating only the output state, i.e., the mean response time. Consequently, the state weight is given by $Q_x$=$C^T Q_y C$. The terminal state weight is obtained by solving steady state Riccati equation. In this work, the reference values for the output state and terminal state are set to be the desired response time target. To allow some flexibility with respect to the optimization, $y_{max}$ in Equation 3 is set to be slightly above this reference value. In essence, one can consider the reference value to be a soft limit and $y_{max}$ to be a hard limit on mean response time.

[1]The discrete time sample is indicated as a subscript for the sake of clarity.

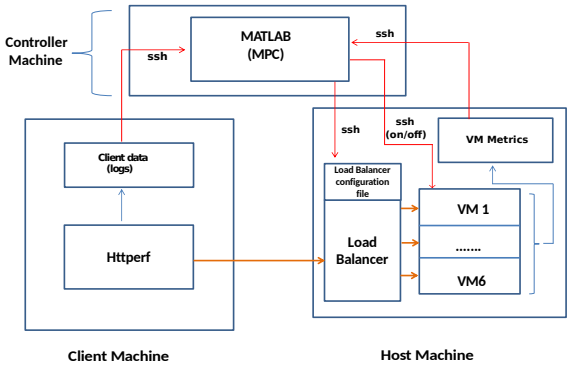


Fig. 1. Experimental setup

## IV. Experimental Setup

Figure 1 shows the experimental setup of the system. The server (host) machine is used to host the Web service VMs. It has 32 GB of physical memory and has 2 sockets with each socket containing 4 cores clocked at 3.2 GHz. We use the VirtualBox [3] virtualization software (version 5.1.6_Ubuntur110634) on this machine to create and manage the VMs. In our experiments, we execute up to 3 VMs on each socket resulting in a maximum of 6 VMs. Each VM is allocated one physical core and 2 GB of memory. Each VM instance hosts RUBiS [4], a Web service that emulates a system where users can bid and buy items. RUBiS is deployed on the Apache Web server. The server machine also executes the HAproxy [1] load balancer, which uses a round robin policy to distribute incoming requests equally among the VMs active in the system. To facilitate control, we sample the mean physical processor per-core utilizations of each of the VMs every 2 seconds, i.e., the sampling period.

The client machine has 16 GB of physical memory and 4 cores clocked at 3.2 GHz. This machine executes the httperf [9] workload generator, which submits a synthetic workload to the RUBiS VMs hosted on the server. We focus on the default RUBiS browsing mix workload for this study. We use the instrumentation provided by httperf to record mean response times for the RUBiS transactions over each sampling period.

A third machine is used as controller in which MATLAB [2] is used to implement MPC. This machine receives the mean response time from the client machine and the mean core utilization from the server machine. These are in turn provided as inputs to the controller. All three machines are connected to each other using a dedicated 1 Gbps fast Ethernet switch. For one of our experiments, we emulate performance interference on the server machine by executing background "noise" processes that consume the machine's processor cores. Specifically, we execute a lighttpd Web server on each VM to emulate interference. The lighttpd server hosts a PHP script. Every time the script is invoked by a HTTP request, it consumes a CPU service time that is exponentially distributed with a mean of 0.02 seconds. We can control the mean CPU consumption of the lighttpd

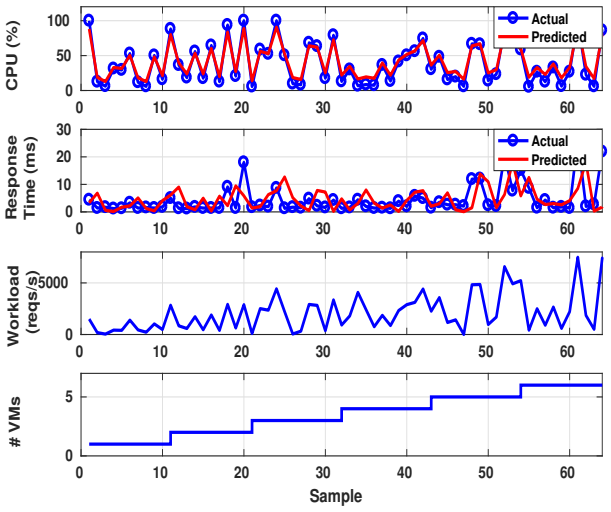


Fig. 2. Validation plot on test data.

server, and hence the degree of performance interference experienced by the RUBiS VMs, by controlling the rate at which the script is invoked. To prevent the background noise from completely depriving the RUBiS VMs of resources, we bound the maximum mean per-core utilization of the lighttpd server to 60%.

## V. MPC Controller

### A. System Identification

We leverage standard techniques proposed in literature [16] to identify the system model. Specifically, to cover a large space of the control input we have followed two approaches. First, the number of VM instances is varied in a sinusoidal manner while RUBiS is subjected to a randomly chosen deterministic request rate. The maximum rate is chosen so as to avoid system saturation. The amplitude of the sine wave is selected such that it covers the range of the possible values of the control input. Second, the request rate is linearly increased first and then decreased. For every request rate $r$, the number of VM instances $VM_r$ is selected randomly from the range $MinVM_r$ to 6, where $MinVM_r$ is the minimum number of VMs needed to avoid system saturation at that rate and is obtained by hit and trail method.

The state space model matrices of Equation 1 after parameter estimation using linear regression are as follows:

$$A = \begin{bmatrix} 0.7309 & 0.0009 \\ 1.1524 & 0.6761 \end{bmatrix} B = \begin{bmatrix} 0.59 \\ -37.18 \end{bmatrix} E = \begin{bmatrix} 0.0008 \\ 0.0383 \end{bmatrix}$$

Figure 2 shows the validation plot, i.e. measured data vs model predicted data, based on a subset of the system identification experiments. In this figure, the red line indicates model predictions whereas the blue line shows the corresponding measurements. As shown in the figure, we vary the request rate randomly to observe the changes in the state variables. As mentioned previously, we focus on building a linear model that provides accurate predictions for higher per-core CPU utilization values. This is reflected in the figure since response time and utilization predictions closely match in this operating region. In some regions, the model overestimates response times. The overall coefficient of variation, i.e., $R^2$, values for utilization and response time are 0.91 and 0.58, and the RMSE value are 5.56 and 4.07, respectively.

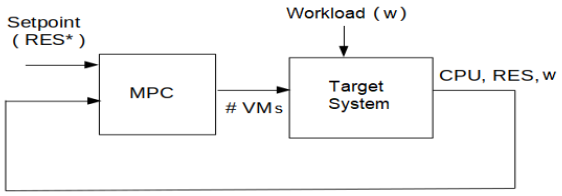


Fig. 3. MPC implementation scheme on the target system

### B. Controller Implementation

Figure 3 shows the MPC implementation scheme on the target system. The target system has two state variables CPU utilization ($CPU$) and mean response time ($RES$) as well as an exogenous input ($w$), as explained previously. As shown in the figure, the values of these states and input are available as inputs to the MPC controller at every sampling instant.

As discussed in Section III, MPC requires reference values to be specified for defining the reference trajectory. A reference response time $RES^*$, i.e., set point ($x_{ref,i}$), of 10 ms is defined for the mean response time ($RES$) state variable. This value is usually decided based on Service Level Agreements (SLA) the service operator has with end users. The controller attempts to achieve this set point within a specified prediction horizon. Reference values for the control input, i.e., $u_{ref,i}$, need to be specified as well. Settings close to the maximum number of VMs in our setup, i.e., 6, will lead to over provisioning while settings close to 1 increase the likelihood of set point violations. The control references are set to 3 as a trade-off between these extremes.

Finally, we discuss the MPC constraints. We place a constraint that $RES$ not exceed 15 ms. As mentioned previously, this value is intentionally chosen to be higher than the set point to provide the controller some flexibility in identifying feasible solutions. To reflect our experiment setup, the MPC optimization also constrains the number of VMs to be in the range 1 to 6.

The tunable parameters of MPC are typically arrived at by creating workloads similar to those used for system identification and observing controller performance under various settings for these parameters. Using this process, we choose the prediction horizon $N_p$ to be 20, as discussed in Section VI. Furthermore, following this process the settings used for the state, control, and terminal state are $Q_x = 1$, and, $Q_u = 0.5$, respectively. The value of $Q_s = 5$ is obtained by solving Riccati equation. Finally, as mentioned previously, we select a sampling interval of 2 seconds.

## VI. Results

We first consider the choice of MPC parameters, which can influence controller performance. While a detailed sensitivity

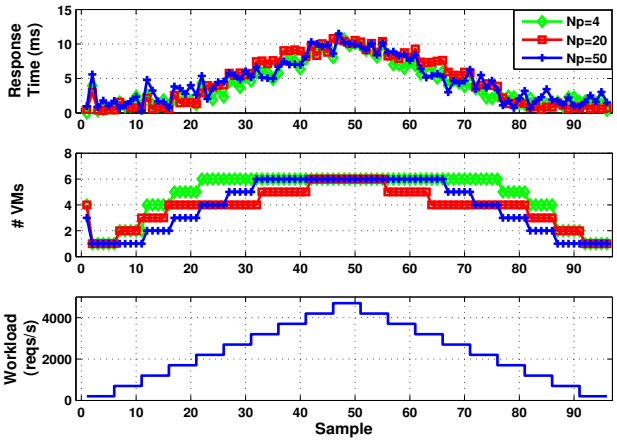


Fig. 4. Sensitivity Analysis of MPC with various prediction horizon ($N_p$)

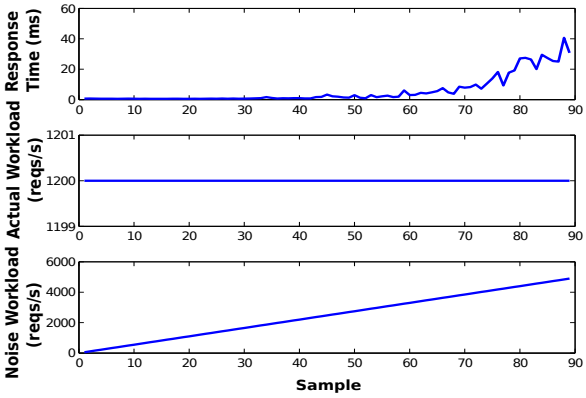


Fig. 5. Effect of interference on RUBiS response time

analysis on these parameters cannot be discussed due to space constraints, we investigate the impact of the prediction horizon $N_p$.

Figure 4 shows controller performance with three different settings for $N_p$ , namely 4, 20, and 50. It can be observed from the plot that the response time set point is satisfied in each case. From the figure, at very low request rates where the model yields pessimistic predictions $N_P = 4$ and $N_p = 20$ causes the controller to use slightly more VMs than actually required. In contrast, $N_p = 50$ is less sensitive suggesting that a longer prediction horizon is robust to model inaccuracies. A different behavior can be observed when the system enters the operating region where the model is more accurate, i.e., a request rate greater than 2200. From the figure, in this region $N_p = 50$ uses slightly more number of VMs than $N_p = 20$. Since we are more interested in this region, we use $N_p = 20$ in our experiments.

The time to compute the control decision is also a key factor in selecting the prediction horizon. The computation time with $N_p = 20$ is approximately 0.45 seconds, which is acceptable given our system dynamics. The computation time can become prohibitively high with very large prediction horizons. For example, the computation time with $N_p = 200$ is only 0.67 seconds. However, it increases to 7.70 seconds with $N_p = 1000$.

Figure 6 studies the ability of the controller to track the

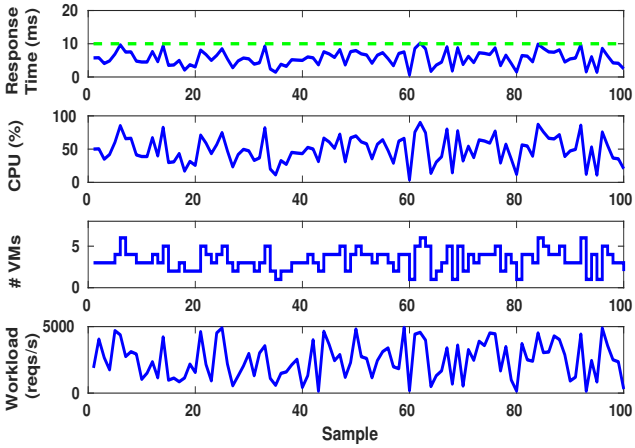


Fig. 6. Impact of workload fluctuations

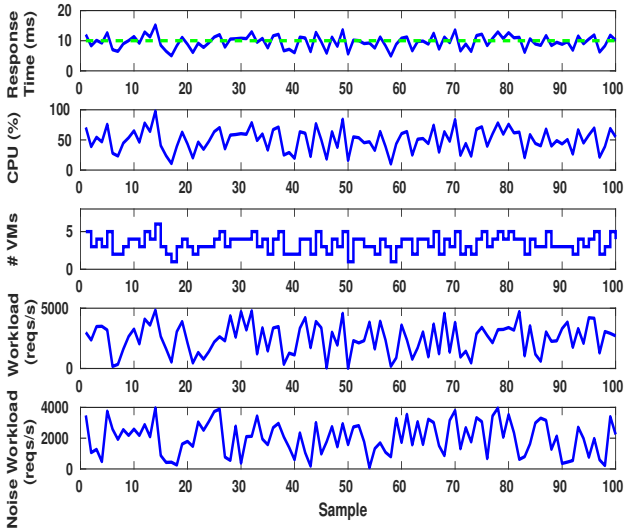


Fig. 7. Impact of interference

set point when the Web service is subjected to more frequent and unpredictable workload fluctuations than that shown in Figure 4. From the figure, the controller is able to handle fluctuations in the request rate by modulating the number of VMs used. The controller guarantees that the response time never go beyond the threshold value of 10 ms. We note that the controller allocates just the right number of additional VMs to achieve this effect instead of over provisioning. Such a strategy is cost effective for systems that bill for VM usage at fine timescales, e.g., the charge per minute of usage policy of Google compute engine.

We now explore the robustness of the MPC technique to interference. As described previously, we emulate performance interference by executing background lighttpd processes that steal processor resources from the RUBiS VMs. Figure 5 shows the effect of this background noise on RUBiS response time. In this experiment the noise is increased by linearly increasing the request rate to the lighttpd servers while the workload to RUBiS is kept constant. From the figure, it can be observed that the response time of RUBiS

increases significantly beyond 4000 requests per second of lighttpd traffic.

To study controller behavior under both workload fluctuations and interference, both the request rate to the RUBiS VMs and its associated lighttpd background noise process are varied randomly. From Figure 7, the response time of the RUBiS VMs increases whenever the combined impact of the RUBiS and lighttpd workloads drives the per-core CPU utilization to very high values. However, the MPC technique is remarkably robust to interference. It is clear from Figure 7 that the set point of 10 ms is violated during periods of considerable interference. However, the controller still makes sure that the mean response time never exceeds the 15 ms hard limit constraint. We note that our system identification did not explicitly capture the impact of interference. However, it is captured implicitly by the model since both the RUBiS VM and the background noise stress the same resource, i.e., the PM's processor cores. However, in a situation when MPC fails to provide a feasible solution due to the constraints, the maximum control input must be implemented so that the response time is satisfied. But, during experimentation we have not encountered this issue.

## VII. Conclusions

This paper investigates the use of MPC to guarantee the performance of a cloud-based Web service. We explore MPC since it can easily accommodate constraints regarding service response time targets and the number of VMs available to the service. To realize a controller that uses MPC, we first build a linear model to capture service dynamics in an operating region, i.e., per-core CPU utilization range, of interest. We construct a controller that uses this model within an optimization formulation to determine the number of VMs to assign to the service at any given sampling instant to satisfy a specified mean response time constraint. Results show that, with the right choice of prediction horizon, MPC is robust to model inaccuracies. They also show that the controller is effective over a wide range of request rates. Furthermore, the controller is robust even in presence of considerable performance interference in the cloud platform.

Future work will consider other constraints. For example, in some environments such as EC2 it might be beneficial to not relinquish VMs immediately since VMs are billed on an hourly or minute basis. Furthermore, many cloud platforms will have VM activation delays and overheads, which need to be considered while making scaling decisions. We will also consider scenarios where there could be performance interference for multiple PM resources. Finally, we will explore MPC controllers that can be deployed by a cloud subscriber. This is a challenging problem since a cloud subscriber cannot access metrics that capture the usage of system resources by competing VMs.

## References


[1] HAProxy - Reliable, High Performance TCP/HTTP Load Balancer. (2017). http://www.haproxy.org/

[2] MATLAB - MathWorks, https://in.mathworks.com/products/matlab.html

[3] Oracle VM VirtualBox. (2017). https://www.virtualbox.org

[4] RUBiS - Home Page. (2017). http://rubis.ow2.org/

[5] Rodrigo N Calheiros, Rajiv Ranjan, and Rajkumar Buyya. 2011. Virtual machine provisioning based on analytical performance and QoS in cloud computing environments. In Parallel processing (ICPP), 2011 international conference on. IEEE, 295-304.

[6] Eduardo F Camacho and Carlos Bordons Alba. 2013. Model predictive control. Springer Science & Business Media.

[7] Carlos E Garcia, David M Pre, and Manfred Morari. 1989. Model predictive control: theory and practice-a survey. Automatica 25, 3 (1989), 335348.

[8] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. 2004. Feedback control of computing systems. John Wiley & Sons.

[9] httperf. 2017. HTTP performance measurement tool,. (2017). http://www.hpl.hp. com/research/linux/httperf/httperf-man-0.9.pdf

[10] Jeffrey Hu, Kiran Kamath, SEN Saurav, and Sandhya Kunnatur. 2008. Policy- Based Hypervisor Configuration Management. (Sept. 15 2008). US Patent App. 12/210,928.

[11] Mircea Lazar. 2006. Model predictive control of hybrid systems: Stability and robustness. (2006).

[12] Young Choon Lee, Chen Wang, Albert Y Zomaya, and Bing Bing Zhou. 2010. Profit-driven service request scheduling in clouds. In Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on. IEEE, 15-24.

[13] Frank L Lewis, Draguna Vrabie, and Vassilis L Syrmos. 2012. Optimal control. John Wiley & Sons.

[14] Joydeep Mukherjee, Diwakar Krishnamurthy, and Jerry Rolia. 2015. Resource contention detection in virtualized environments. IEEE Transactions on Network and Service Management 12, 2 (2015), 217-231.

[15] Ripal Nathuji, Aman Kansal, and Alireza Ghaarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In Proceedings of the 5th European conference on Computer systems. ACM, 237-250.

[16] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. 1996. Applied linear statistical models. Vol. 4. Irwin Chicago.

[17] Jia Rao, Yudi Wei, Jiayu Gong, and Cheng-Zhong Xu. 2013. QoS guarantees and service differentiation for dynamic cloud applications. IEEE Transactions on Network and Service Management 10, 1 (2013), 43-55.

[18] Penamakuri Sesha Saikrishna, Ramkrishna Pasumarthy, and Nirav P Bhatt. 2017. Identification and Multivariable Gain-Scheduling Control for Cloud Computing Systems. IEEE Transactions on Control Systems Technology 25, 3 (2017), 792-807.

[19] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffn, and Leendert Van Doorn. 2005. Building a MAC-based security architecture for the Xen open-source hypervisor. In Computer security applications conference, 21st Annual. IEEE, 10-pp.

[20] Jorg Schad, Jens Dittrich, and Jorge-Arnulfo Quiane-Ruiz. 2010. Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proceedings of the VLDB Endowment 3, 1-2 (2010), 460-471.

[21] Durgesh Singh, PS Saikrishna, and Ramkrishna Pasumarthy. 2016. Modeling and Performance management of a Virtualized Web-server. In 11th International Workshop on Feedback Computing (Feedback Computing 2016).

[22] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, and Pawan Goyal. 2005. Dynamic provisioning of multi-tier internet applications. In Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on IEEE, 217-228.

[23] GuohuiWang and T. S. Eugene Ng. 2010. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center (INFOCOM).

[24] Lixi Wang, Jing Xu, Hector A Duran-Limon, and Ming Zhao. 2015. QoS-driven cloud resource management through fuzzy model predictive control. In Autonomic Computing (ICAC), 2015 IEEE International Conference on. IEEE, 81-90.

[25] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. 2008. Autonomic resource management in virtualized data centers using fuzzy logicbased approaches. Cluster Computing 11, 3 (2008), 213-227.

[26] Qian Zhu and Gagan Agrawal. 2010. Resource provisioning with budget constraints for adaptive applications in cloud environments. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. ACM, 304-307.