Runtime Performance Management for Cloud Applications with Adaptive Controllers

Cornel Barna
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, Canada
cornel@cse.yorku.ca

Marin Litoiu
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, Canada
mlitoiu@yorku.ca

Marios Fokaefs
Dept. of Comp. and Soft. Eng.
Polytechnique Montreal
Montreal, Canada
marios.fokaefs@polymtl.ca

Mark Shtern
Dept. of El. Eng. and Comp. Sci.
York University
Toronto, Canada
mark@cse.yorku.ca

Joe Wigglesworth
IBM Toronto Lab
IBM Canada Ltd.
Markham, Canada
wiggles@ca.ib.com

ABSTRACT

Adaptability is an expected property of modern software systems in order to cope with changes in the environment by self-adjusting their structure and behaviour. Robustness is a crucial component of adaptability and it refers to the ability of the systems to deal with uncertainty, i.e. perturbations or unmodelled system dynamics that can affect the quality of the adaptation. Cost is another important property to ensure that resources are used prudently and frugally, whenever possible. Engineering robust and cost-effective adaptive systems can be accomplished using a control theory approach. In this paper, we show how to implement a model identification adaptive controller (MIAC) using a combination of performance and control models and how such a system satisfies the goals for robustness and cost-effectiveness. The controller we employ is multi-input, meaning that it can issue a variety of commands to adapt the system and multi-output, meaning it can regulate multiple performance indicators simultaneously. We show that such a solution can account for uncertainty and modelling errors and efficiently adapt a web application with multiple tiers of functionality spanning multiple layers of deployment, software and virtual machines, on Amazon EC2, an actual cloud environment.

CCS CONCEPTS

• **Mathematics of computing → Kalman filters and hidden Markov models**; • **Software and its engineering → Cloud computing**; **Software performance**; *System administration*;

KEYWORDS

software, adaptive systems, performance modelling, performance optimization, cost, cloud computing, control theory, linear quadratic regulator

ACM Reference Format:
Cornel Barna, Marin Litoiu, Marios Fokaefs, Mark Shtern, and Joe Wigglesworth. 2018. Runtime Performance Management for Cloud Applications with Adaptive Controllers. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3184407.3184438

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184438

1 INTRODUCTION

Novel technologies, like cloud computing and resource virtualization have allowed for the better management of computation resources of software systems leading to the need for self-managing and autonomic systems [2], which eventually bore the field of self-adaptive software systems. In essence, a self-adaptive system senses the changes in the operating conditions and in the environment and adjusts its structure and behaviour to meet its goals in the presence of those changes. A reference MAPE architecture [14], which consists of Monitoring, Analysis, Planning and Execution components, allows the design and the implementation of an Autonomic Manager that regulates the performance of the software system against fluctuating incoming traffic.

While there has been major theoretical progress in the field, there are still substantial challenges in designing and implementing self-adaptive systems and eventually limited engineering solutions, which can guarantee the degree of automation and robustness expected from such an autonomic manager. Control theory has long been a popular choice for autonomous systems, especially in the domain of physical systems, including manufacturing, automotive and aerospace industries. However, it is only recently that it has been explored as an alternative for self-adaptive software systems and its popularity has not yet reached high levels. This is probably due to the complexity of designing such systems. Nevertheless, the benefits in robustness and cost-effectiveness can potentially outweigh the effort for designing control systems.

Our work aims to demonstrate and clarify the process of developing an automated controller for software applications deployed

on cloud infrastructure. We discuss the key components of the controller, namely the performance model and the actual control. As far as the model is concerned, we discuss the representation of the cloud deployment as a *layered queuing network*, which very accurately captures the non-linear nature of the software performance. We use a *layered queuing model* (LQM) as the performance model and we employ Kalman filters as the parameter estimator. In contrast with the model, we opt for a simpler, yet as robust as required, linear controller. We detail the construction of this controller through the linearization of LQM. More importantly, we discuss the benefits of *multi-linearization*, which constitutes a novel contribution, at frequent intervals as the system progresses and its conditions change during runtime. The linear controller of choice is called *linear quadratic controller* (LQR). The synergy between LQM, Kalman and LQR results in a *model identification adaptive controller* (MIAC) architecture, which eventually constitutes our autonomic management system (AMS).

The remainder of the paper is organized as follows: Section 2 presents the proposed architecture and the development process. Section 3 presents the experimental results that validate our approach. Section 4 discusses the key challenges we had to overcome and the key lessons we learned in the process of developing a MIAC controller. Section 5 introduces the background and related work. The conclusions are presented in Section 6.

2 MODEL IDENTIFICATION ADAPTIVE CONTROLLER

To construct a management system based on a MIAC architecture, we need two key components; the performance model and the controller. In this section, we first describe the layered queuing model (LQM) [10] used to capture the performance of the subject software system. We present how the model is designed to operate under uncertainty so that it allows the controller to be robust against perturbations. Second, we describe the process to design a linear quadratic regulator (LQR) [2] as our controller. We discuss how LQR consumes the state of the system through the performance model, and how the latter has to first be linearized. Necessary steps are also discussed so that the controller can handle dynamic and volatile environments through an adaptive recalibration cycle.

2.1 Performance Model

We opt to use a similarly non-linear model, a *layered queuing model* (LQM) to capture the system's performance. According to LQM [10], the functional tiers of the system are represented as queues, introducing delays to the service time of the overall system, and the nested nature of the infrastructure (software, VMs) is represented as layers. The model orchestrates the layered queues to evaluate the state of the system. The state of the system can be any quality index representing its performance, e.g., CPU utilization or response time, and it is determined mathematically as a set of functions of the incoming workload (w), the topology of the system's infrastructure (u) and the demand of the service requests in terms of resources for each queue (d). Formally, the discrete time model is $y(t) = LQM(w, u, d)$, where $y(t)$ is the vector of the system's output for a given moment in time t. Assuming that the demands of the system are known (by measurement or estimation), we can use the model to estimate the output under any combination of workload and infrastructure. It is important to note here that while u represents the topology of the system for the LQM, it also represents the input of the controller, i.e., the commands, which are also scaling actions upon the system's infrastructure.

2.1.1 Modelling under uncertainties and volatility. When an application is deployed on cloud, its performance is affected by the dynamics of the cloud infrastructure. Only certain cloud components (such as VMs) are visible to the application modeller, others, including hardware or resource management services, are not visible. However, these invisible components may be the cause for delays, which affect the application performance. Although such delays are reflected in the measured performance metrics, their source cannot always be identified [19, 20] and, thus, they are considered *uncertainties*.

There can be two types of uncertainties that may affect the accuracy of the application model in cloud and therefore the efficiency of the performance controllers we design: parametric uncertainties and unmodelled dynamics. The parametric uncertainties refer to both parameters of the model (such as service times, number or probabilities of calls between different components of the software, communication delays, etc.) or the intensity and mix of the workloads. The unmodelled dynamics refer to structural deficiencies of the model, that is missing components and queues that we have not knowledge of. The latter are very important in cloud where we do not have complete knowledge of the deployment environment.

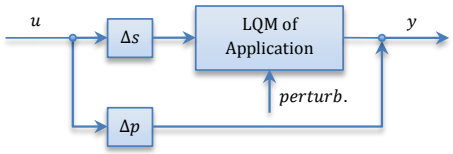


Figure 1: *Extended LQM. Includes structural uncertainties*

To account for unmodelled dynamics, we add two sub-models to extend the application model as seen in Figure 1: a serial sub-model, made of a queuing centre, Δs, and a parallel sub-model, made of another queuing centre, Δp. The serial model Δs will account for the delays in the application requests processing, due to additional proxies in the cloud. The parallel model Δp, will account for speed ups at higher loads, due to possible caching or heterogeneity of the cloud resources. The parameters of those queuing centres, such as service times and visit probabilities (for the parallel model) are unknown at design time and they will be identified at runtime together with other parameters of the model. As a result, the architecture of the LQM model will consider the model for the deployed application in the cloud, as well as these two additional queuing centres.

Parametric uncertainties can occur due to the high volatility of the application's environment, either with respect to users, e.g., changes in volume or type of incoming traffic, or the cloud deployment, e.g., changes in number or type of resources. The model is valid for a specific mix of workload and infrastructure. When the operational conditions change (a change in the topology, the workload intensity or the type of the workload), the model becomes outdated

and needs to be retuned. In the case of the proposed LQM, this tuning happens with the use of Kalman filters. Prebuilding all the models at design time is infeasible because of the large number of possible changes and combinations of changes. Therefore, runtime retuning of the model is more efficient and, therefore, preferable. This gives the adaptive nature to our MIAC architecture.

2.2 Control System

Having captured the performance of the managed application as a non-linear model, we can now proceed to design the controller, which will consume said model. We opted for the feedback controller LQR, where the system is modelled as a set of linear differential equations and the goal is modelled as a quadratic function. A significant advantage of feedback controllers is that they can give an optimal, in terms of efficiency and effectiveness, set of adaptive actions both automatically and fast. Alternatively, we would have to simulate and evaluate every possible combination of states and adaptive actions, possibly over multiple dimensions, in terms of performance parameters and types of resources, before we can find the optimal adjustment.

2.2.1 Linearizing and discretizing the models. Given how LQR is specified, the first step is to linearize LQM and feed the linear models in the controller. We can extract such a linear model, if we observe the behavior of the system around an operational point (OP) $[x_{op}, y_{op}, u_{op}]$, where x generally refers to the state of the system, i.e., the monitored performance metrics, y is the observable output of the system and u is the controller's input commands, i.e., the adaptive actions. If we focus closely to the OP, we can linearly approximate the system's behavior from the non-linear model. We can take points close to the OP in discrete time k by applying the corresponding *deltas* (i.e. small differences). In the case of a cloud-deployed software system, given an OP, the process can be achieved by slightly varying the infrastructure (commands u) with respect to that of the operational point. At this point and within a small range, we can assume that the workload is constant. Thus, by taking model measurements for slightly different commands, we can generate a set of discrete points around the operational point.

Using these equations to find points close to the operational point, we can define a discrete-time linear system described by:

$$\begin{cases} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \end{cases} \tag{1}$$

where A, B, C and D are matrices; y, u and x are vectors. When the matrices are constant in time, the system described by Equation 1 is a linear time invariant (LTI). To find the matrices A, B, C and D, which make up the linear approximation of the system, given the OP and some points around it, we use linear regression. Given that linear regression is simple and efficient, we chose it in order to create models that may be valid for points further than the OP. Other linearization techniques focus too much around the OP. In our case, we can afford to sacrifice some of the accuracy, since according to our method the linear model will at some point have to be updated around a new OP.

2.2.2 Designing the controller. After the linear model is available, the next step is to design the actual controller in terms of the goals to be optimized. More specifically, the goals are defined as a performance index across the state and command variables in form of a quadratic function (Equation 2), with the objective of finding the command u that minimizes this quadratic function subject to the system in Equation 1. The weight, or penalty, matrices Q_x and Q_u penalize the state variations or the cost of adaptive commands, respectively. The construction of the weight matrices depends on the domain on which we apply the controller. Although the matrix Q_u refers to cost, it does not necessarily capture it directly as the monetary cost for resources, set by a provider. More precisely, it declares the preferences of the system's designer on the various commands. Nevertheless, the matrix can be derived directly from the provider's financial costs through some mathematical transformations, thus producing cost-effective adaptations. The specific design method of the penalty matrices remain out of scope for this work, although we validate the impact of choosing different penalty matrices.

$$J = \sum_{0}^{\infty} x^T Q_x x + u^T Q_u u \tag{2}$$

An optimal, feedback controller will find the u that minimizes J. Given the linear model and Equation 2 and assuming that the system is controllable, the steps to synthesize the controller [2] are summarized next. The optimization problem has the following solution:

$$u = -Kx + k_r y_r \tag{3}$$

where x is the system's state as defined earlier, y_r is the goal for the output, K is the feedback gain matrix and k_r is the steady-state factor.

The feedback gain matrix guarantees that the system will remain stable, in the form of $y = 0$, meaning that the output of the system will remain close to the operational point ($y_a(k) = y_{op}$). Since our goal is to bring, in fact, the output towards its desired state (i.e. $y = y_r$), we need another factor, which is k_r.

K is calculated with LQR as:

$$K = -Q_u^{-1} B^T P x$$

where $P \in \mathbb{R}^{n \times n}$ is a positive definite, symmetric matrix that satisfies the *Riccati equation*:

$$PA + A^T P - PBQ_u^{-1} B^T P + Q_x = 0$$

Based on K, we can calculate k_r by solving the following equation:

$$1 = C(A - BK)^{-1} B k_r \tag{4}$$

2.3 Autonomic Management System

With the LQM performance model and the LQR controller constructed, we can move on with building the final autonomic management system for cloud applications, following the MIAC architecture as illustrated in Figure 2. This architecture has the ability to be robust in the face of uncertainty, thanks to the extended LQM performance model(which incorporates the structural uncertainities), and in spite of the dynamic nature of the software system, thanks to the recalibration and relinearization of the LQM when it becomes outdated, so that LQR has always an accurate model to operate with.

The flow of data and control is further presented in Algorithm 1. The algorithm requires as input four sets $\mathcal{W}$, $\mathcal{Y}$, $\mathcal{U}$, and $\mathcal{X}$. $\mathcal{W}$ contains the names for the workload parameters that are to be monitored. $\mathcal{Y}$ contains the outputs of the system, which will determine whether the system operates normally or not. $\mathcal{U}$ contains the set of resources or parameters, e.g. number of threads or servers, which we can change to bring the system back to a healthy state. $\mathcal{X}$ is a subset of measured or estimated performance variables that we use in linearized model Sets $\mathcal{W}$, $\mathcal{Y}$ $\mathcal{U}$ and $\mathcal{X}$ are *nominal* sets, meaning they only specify *what* is to be included, based on which the actual measurement vectors w, y, u, x are monitored or generated by the manager. Apart from these four sets, the algorithm requires as input the original (before runtime calibration) non-linear model, LQM_0.

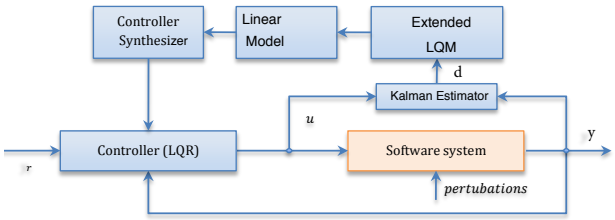


Figure 2: *Model Identification Adaptive Control Architecture.*

Algorithm 1: Model Identification Adaptive Controller (MIAC)

input: $\mathcal{W}$ – the set of workload parameters to be monitored;
$\mathcal{Y}$ – the set of system outputs;
$\mathcal{U}$ – the set of possible commands;
$\mathcal{X}$ – the set of state variables;
LQM_0 – the original non-linear performance model

```
1  while TRUE do
2      for each sampling interval t do
3          Measure system variables → [y_c, u_c, w_c];
4          Estimate performance parameters(Kalman) → [d_c];
5          Update Extended LQM
6          if linear model not accurate– its outputs deviates from those of
             LQM; then
7              Set [x_op, y_op, u_op] = [x_c, y_c, u_c];
8              Linearize LQM → [A, B, C, D];
9              Synthesize LQR for linear model → [K, k_r];
10     Controller produces adaptive commands → Δu ;
```

In step 1 of the algorithm, and illustrated in Figure 2, it can be seen that the monitoring and control of the system by MIAC is a closed loop. Step 2 is the iteration over time. At each iteration, we maintain the model synchronized through Kalman calibration. In step 3, we measure the current state of the system, where vector y_c contains the current measurements for the system's outputs, vector u_c is the current configuration of the system, for example, its virtual resources and their topology, and vector w_c is the current state of the workload, e.g., arrival rate. In step 4, the *Kalman Estimator* estimates the new performance parameters, with which it updates the extended LQM, in step 5. In step 6, we check whether the current linear model is still accurate, after the update of LQM. If not, then, we activate the upper loop of Figure 2. We, first, use the current state of the system, as extracted in step 3, to define a new operational point in step 7 (as discussed in Section 2.2.1), where vector x_c includes measurements for the system's current state as they come from the monitoring service. In step 8, the updated LQM is linearized, a process which is detailed later, where A, B, C, and D are the coefficient matrices for the linear model. Based on the linear model, in step 9, we design an optimal controller, as described in Section 2.2.2. The controller is now ready to operate in the lower loop of Figure 2. By monitoring the deployed system and comparing its behavior against a set of goals y_r, in step 10, the controller can issue a set of commands u to rectify any problematic situations.

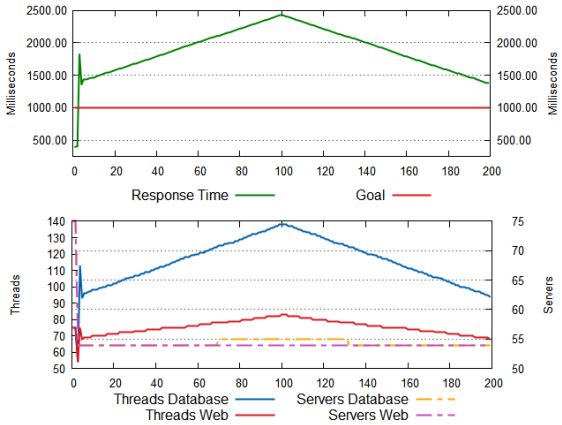
Notice that the upper loop of Figure 2 is activated only when the linear model diverges too much from the actual system (step 3 of Algorithm 1). If the linear model is accurate, then only the lower loop is executed. This will ensure that the controller is always valid for the current state of the system. A second observation is that when the output of the system is the same as the goal (i.e., $y = y_r$), then the command set produced by the controller Δu is empty; this result is guaranteed by the mathematical definition of the controller.

3 EXPERIMENTAL STUDIES

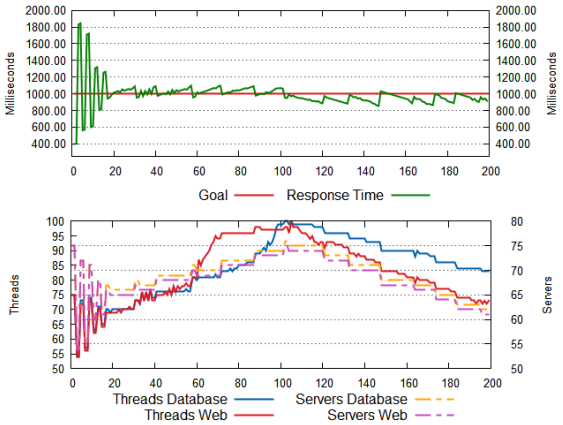
To validate the MIAC manager we constructed in the previous section, we have conducted a series of experiments. For all experiments, we deployed a *bookstore* application, developed using J2EE technology, on multiple Linux (Ubuntu) virtual machines on the Amazon EC2 cloud. The application performs various SQL commands (select, insert, update). We developed an LQM model for this setting to capture the performance of the application. The efficiency of the controller was first validated on the model itself and later on an actual deployment in Amazon. Any change in the controlled system is reflected in the model. In the initial topology, the database server (MySql) was deployed on one instance, the web application server (Tomcat) was deployed on two instances, while a fourth instance was acting as a load balancer (Apache 2) to distribute the incoming web requests between the application servers.

As input to our algorithm and in order to design the controller, we need to define the set of commands $\mathcal{U}$, the set of monitored state parameters $\mathcal{X}$ and the set of controlled system outputs $\mathcal{Y}$. We define the command points for our system as $\mathcal{U} = [S_d, T_d, S_w, T_w]$, where T_d is the number of threads for database servers and T_w for web servers; S_d is the number of database servers and S_w is the number of web servers. The state vector, x, contains the response time of the web application, the same as the output vector, y (see Equation 1). For the rest of our experiments, when we refer to specific values of these vectors, we will note them with their lower case representation, i.e. u, x and y.

We construct the LQM using the OPERA tool [16] to track the behavior of the system. In every iteration, the LQM is retuned using Kalman filters to recalculate the model parameters, so that the model remains accurate.



(a) *Without re-linearization.*



(b) *With re-linearization.*

Figure 3: *Behaviour of the system when the goal for Response Time was set to 1000 ms and the workload monotonically increases / decreases.*

To linearize the LQM, we have used the `LinearRegression` function from the package `optim` [1] in Octave to calculate the matrices A and B from Equation 1.

We have also chosen $C = [1]$ and $D = 0$. The reason for this is because we picked response time to represent both the state and the output of the system. From Equation 1, we have that $y(k) = Cx(k) + Du(k)$. Given that $y = x$, it is derived that $C = [1]$ and $D = 0$.

For the LQR, we have used the implementation offered by Octave's package `control` [18]. The parameters for the `lqr` function were the matrices A and B identified during linearization and the following weight matrices Q_x and Q_u:

$$Q_x = [1]; \quad Q_u = \begin{bmatrix} 100\,000 & 0 & 0 & 0 \\ 0 & 1\,000 & 0 & 0 \\ 0 & 0 & 100\,000 & 0 \\ 0 & 0 & 0 & 1\,500 \end{bmatrix} \tag{5}$$

The `lqr` function calculated the matrix K as discussed in Section 2.2.2. To calculate k_r, we assumed that all of its four components were equal and applied the Equation 4:

$$k_r = \frac{1}{b_1 + b_2 + b_3 + b_4} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \times (1 - (A - BK))$$

where b_i are the components of B:

$$B = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

K and k_r form our controller, and allow us to calculate a command by applying Equation 3.

For the first experiment (Figure 3), we have set the goal for the response time to be 1000 ms. As for the workload, we start with 18000 users, linearly increasing up to 23000 and then linearly decreasing again down to 18000 users. Figure 3a shows the behavior of the system when using a conventional LQM, liniarized only at the beginning of the experiment, thus creating only one controller (K and k_r from Equation 3), while Figure 3b shows the behavior when the LQM is relinearized every time its error (the difference between the measured response time and the estimated one using the linear model) exceeds the threshold of 100 ms.

The experiment shows that when we linearize often (we have built 31 linear models for the whole duration of the experiment), we manage to stabilize the system and maintain the response time close to the desired value. Also, the value of J (Equation 2) in this case was approximately 21×10^6, which is significantly smaller than the value obtained with a single linearization: 17×10^9. Considering that the goal of the controller is to minimize J, this shows that multiple linearization is a significantly better model. Figure 3a shows that the controller fails to maintain the response time close to the goal when the workload fluctuates.

Running more experiments with different goals (at 300 ms, 500 ms and 700 ms) produced similar results: the relinearization of the LQM model enabled the controller(s) to maintain a response time close to the desired goal, while doing a single linearization at the beginning of the experiment generated poor results.

In the second experiment, we wanted to evaluate the behaviour of the system in the presence of irregular workload, i.e., when the workload suddenly increases or decreases non-monotonically. The results are summarized in Figure 4. The bottom plot shows the shape of the workload. The goal for the response time in this experiment was set to 700 ms and as it can be seen in the figure, the controller is able to stabilize the system around this goal. In fact, the controller helps the system achieve an average response time of 703 ms (standard deviation of 73.37).

In order to evaluate the influence of the weight matrices Q_x and Q_u on the behaviour of the controller, we have set up an additional experiment with the following matrices:

$$Q_x = [1]; \quad Q_u = \begin{bmatrix} 1\,000 & 0 & 0 & 0 \\ 0 & 100\,000 & 0 & 0 \\ 0 & 0 & 1\,500 & 0 \\ 0 & 0 & 0 & 100\,000 \end{bmatrix} \tag{6}$$

We set the response time goal to be 700 ms, and then increased the workload linearly and then decreased it, also linearly. Figure 5a

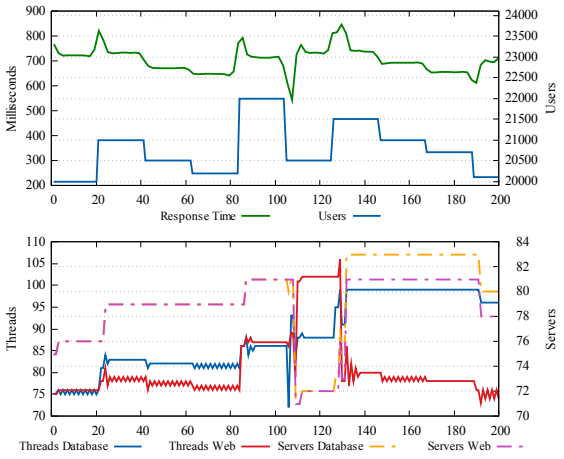


Figure 4: *Behaviour of the system when the goal for Response Time was set to 700 ms, and the workload increasing/decreasing suddenly.*

shows the behaviour of the system when the matrices from Equation 5 (prefer to add servers) are used, while Figure 5b shows the results for matrices from Equation 6 (prefer to add threads).

In both situations, the system was stable and the goal has been maintained. Using the matrices from Equation 5, the value for J (Equation 2) was approximately 27×10^7; the utilization of the weights from Equation 6 resulted in a value for J of almost half, in the range of 14×10^7. This experiment shows that the choice of the penalty matrices affects the adaptation strategies, but performance-wise LQR remains unaffected.

In the last experiment, we deployed our topology on the Amazon EC2 cloud along with the MIAC system. We used small instances (1 CPU, 2 GB RAM and 160 GB hard drive) for the web cluster and the database. We used the regular workload (linearly increasing and then decreasing) starting with 35 users reaching a maximum of about 85. We set the goal for response time at 300 ms. We bounded the number of VMs in the web cluster between 1 and 16. We consider relatively small clusters in order to keep the cost of the experiment low, given that it is conducted with an actual public cloud provider. In addition, given that our database is a single-node MySQL server, we did not consider scaling this cluster. Therefore, it was removed from the set of available commands of the controller. To simplify our experiments, we also excluded web and database threads from consideration due to this reason. Therefore, only the web cluster was scaled. We monitored the system every minute. After every scaling action, we enforced 6 minutes of scaling inactivity to allow for the elastic operation to finish and the system to stabilize itself.

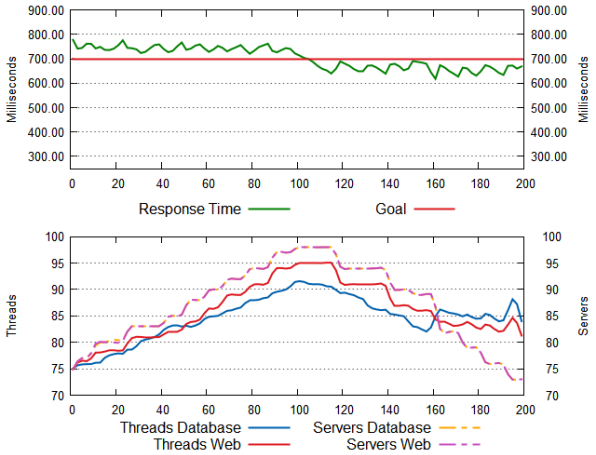
As it can be seen in Figure 6, MIAC was able to maintain the response time close to the set goal even in a real deployment, further strengthening our argument for a robust controller. In fact, the average error was 3.5 ms, with a standard deviation of 84.09. Naturally, it cannot be expected for the controller to keep the response time exactly on the goal. One reason is that for small clusters, the addition or deletion of a single VM can have a great impact to the performance. Nevertheless, response time was eventually kept at the desired levels. In addition, as the figure shows for the response time, the recalculation and calibration of the LQM model resulted in very accurate estimations, with the average error between the measured metrics (m) and the estimated ones (e) from the model to be 0.55 ms (standard deviation of 13.43).

4 LESSONS LEARNED AND CHALLENGES

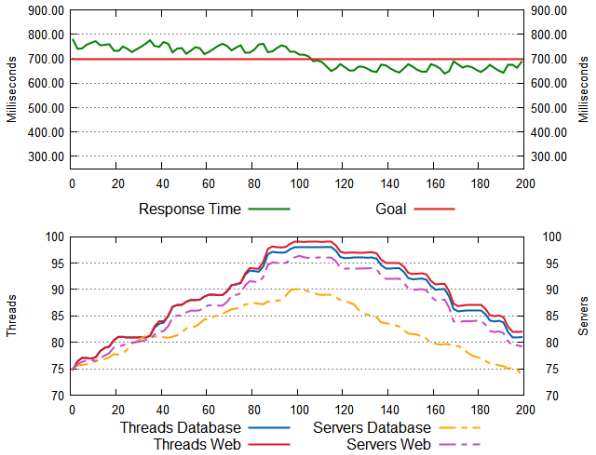
In this paper, we present our process for building a controller to manage the performance parameters of software systems. The task on its own is not trivial and requires a solid background on the mathematical foundations of control theory and the performance model used by the controller. For this reason, in this paper, we also provide all the theoretical details on the LQM model and the LQR controller we built. Although the effort may seem too much compared to other less complex techniques, the benefits can possibly outweigh the extra effort [9]. More specifically, as we have shown the adaptive nature of our controller through the multilinearization process make the controller be more resilient and robust against perturbations coming from the system or its deployment environment. Additionally, thanks to the runtime adaptive mechanisms of both the model and the controller, this extra effort is required only once during the design of the controller, and afterwards the management system requires little to no maintenance, unless the system itself changes.

Another challenge in setting up a controller for software performance is understanding the nature of the inputs and their constraints. For example, in public clouds, computations resources (VMs) come in packages predefined by the provider. Therefore, resources may not be added individually, e.g., only memory or only CPU. This detail has to be carefully taken into account when designing the controller, since the impact of the adaptive action can be less fine-grained than the action seemingly is. This is the main reason behind our choosing response time to choose both the state and the output of the system. The addition of a VM may add multiple resources at the same time (CPU, memory, disk), but its exact effect will be definitely measurable on response time. If we were to model the utilization of the individual resources, the addition/removal of a VM would change all utilizations, even if the respective resources had no problem, which could actually affect the whole control process.

Another issue related to the predefined sizes of VMs is that one can add whatever resources the VM offers *at minimum*. For example, if at one point the application needs one additional CPU core to fix its response time, but we assume that the smallest VM available has 2 CPU cores, the controller cannot do anything but add a VM with 2 cores. This problematic behaviour is exacerbated by the fact that our actions are discrete (e.g., add/remove one, two, three VMs), but the command of the controller can be continuous (e.g., add 1.5 VMs). In this case, a meta-decision needs to be taken (do we round up or down?), which can depend on other factors, for example cost or minimal error compared to goal. In any case, this can cause oscillation or overshooting, as it is exemplified in Figure 6, where response time "jumps" when we add or remove a VM. We can mitigate this situation by selecting a good mix of fine-grained and coarse-grained inputs for our controller. In our experiments, the threads play the role of the fine-grained command, while the VM is the coarse-grained solution. An alternative to the coarse-grained VMs can be the use of containers, like Docker, where one



(a) *Using the Q_x and Q_u from Equation 5.*



(b) *Using the Q_x and Q_u from Equation 6.*

Figure 5: *The effect of the weight matrices Q_x and Q_u on the behaviour of the controller.*

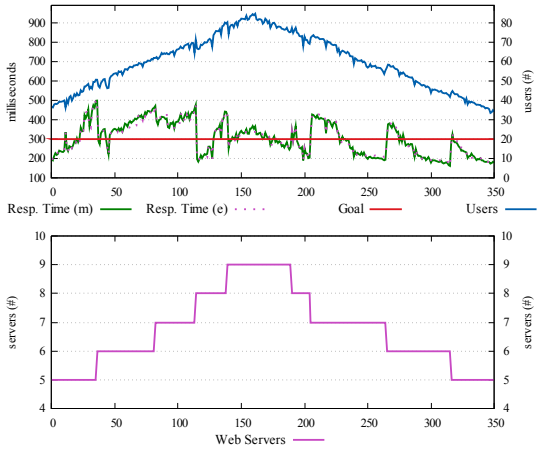


Figure 6: *Behaviour of the system with a real deployment on Amazon EC2. Regular workload was used and the goal was set at 300 ms.*

can slice the host's resources at will and create some more granular than the smallest VM available. Nevertheless, containers may pose challenges on their own like, for example, having one more layer to manage [8] or how container resources can be modelled in a layered queuing model [3].

An important conclusion we drew from the exercise of building MIAC controller is that the effectiveness of a controller implemented using this methodology relies heavily on the capabilities of the model. Resources that cannot be modelled cannot be captured by the controller. In our case, OPERA cannot capture the memory of a software system, so the controller we developed is not able to use memory as a command. In other words, even if our resources where granular enough to allow us to issue commands like increase/decrease the memory, we could not have done so, because the controller would not have been able to asses the impact of the new memory using the model. The key lesson for practitioners here is that *the limitations of the model are transferred to the controller.*

5 BACKGROUND AND RELATED WORK

Feedback loops have been identified as important components in self-adaptive systems [5, 15]. At the conceptual level, feedback loops follow the MAPE architecture [14], but at the implementation levels there are many variations. A prevalent one is based on control theory [4] and has been used for some time. Hellerstein et al. [13] introduced several case studies, where control theory has been used for controlling the threading level, memory allocation or buffer pool sharing in commercial products such as IBM Lotus Notes and IBM DB2.

Although the performance in software is more accurately modelled with non-linear models, many authors use linear models due to their simplicity [7, 12, 13]. Because the models are linear and valid only around the linearization point, controllers designed based on linear models will most likely be valid only around that linearization point and will not be able to handle a large spectrum of perturbations. This is a well known problem in the control of non-linear systems and often control switch approaches [17] are employed to switch between many statically designed controllers. In this paper, we consider the system to be non-linear, represent it with a non-linear model and then linearize it at runtime, around multiple operational points. In this case, the system is modelled as series of linear models and one controller, which is updated multiple times at runtime.

One of the most important reasons for having a model is to study the properties of the system it models and then to design the controller. In control theory, the significant properties are *stability*, *controllability*, *observability*. *Stability* is probably the the main *raison d'être* of control theory. In simple terms, stability means that for bound inputs (commands), the system will produce bound state and output values. Examples of stability studies in control of software and computing systems have been presented in [13].

Observability is the property of the model that allows finding, or at least estimating, the internal state variables of a system from the output variables. This property is important from pragmatic point of view. In a real system, it is impossible, hard or impractical to measure all state variables. On the other hand, the commands and outputs are easier to measure. Examples of how to use observability and how to estimate software performance parameters for applications deployed across multi-tiers and using Kalman filters are presented in [21]. Other authors have used other techniques to estimated runtime model parameters [6] on the same assumption that the system was observable. The concept of controllability (or reachability) describes the possibility of driving the system to a desired state, i.e. to bring its internal state variables to certain values [2]. This property ensures that we can design a controller. We use the concept of observability, in our paper, when we estimate the performance model in cloud. In this aspect, we extend [21] for cloud environments. Our goal is to achieve controllability across a large design space and for any model and controller we design and to make the system stable. Although one can synthesize empirical controllers [11], we follow classic control theory to build optimal controllers [15].

6 CONCLUSIONS

In this work, we demonstrated the construction of a model identification adaptive control architecture as a management system to monitor and maintain applications on cloud environments. Our experiments have shown that the use of control theory in an Adaptive Manager for cloud applications performs exceptionally well and can produce a robust and effective controller. Additionally, the mathematical background of control theory allows us to systematically design and verify such adaptive management systems. This method is capable of operating on a multidimensional level both with respect to the goals that are to be achieved, as well as the adaptive actions. The proposed controller performs better than previous methods thanks to the concept of multilinearization, which allows the controller to readjust itself in order to better monitor the system and produce more efficient adaptive actions. Our experiments on Amazon EC2 showed that our controller is applicable and performant in real settings.

REFERENCES

[1] 2015. Octave Optim Package v1.4.1. http://octave.sourceforge.net/optim/overview.html. (2015).

[2] Karl Johan Aström and Richard M Murray. 2010. *Feedback systems: an introduction for scientists and engineers*. Princeton university press.

[3] Cornel Barna, Hamzeh Khazaei, Marios Fokaefs, and Marin Litoiu. 2017. Delivering Elastic Containerized Cloud Applications to Enable DevOps. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM.

[4] Yuriy Brun, Ron Desmarais, Kurt Geihs, Marin Litoiu, Antonia Lopes, Mary Shaw, and Michael Smit. 2013. A design space for self-adaptive systems. In *Software Engineering for Self-adaptive Systems II*. Springer, 33–50.

[5] R de Lemos, H Giese, HA Müller, M Shaw, J Andersson, L Baresi, B Becker, et al. 2009. Software engineering for self-adaptive systems. In *Dagstuhl Seminar*, Vol. 10431. Springer.

[6] Antonio Filieri, Lars Grunske, and Alberto Leva. 2015. Lightweight adaptive filtering for efficient learning and updating of probabilistic models. *ICSE. IEEE* (2015).

[7] Antonio Filieri, Henry Hoffmann, and Martina Maggio. 2014. Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 299–310. https://doi.org/10.1145/2568225.2568272

[8] Marios Fokaefs, Cornel Barna, Rodrigo Veleda, Marin Litoiu, Joe Wigglesworth, and Radu Mateescu. 2016. Enabling devops for containerized data-intensive applications: an exploratory study. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 138–148.

[9] Marios Fokaefs, Yar Rouf, Cornel Barna, and Marin Litoiu. 2017. Evaluating Adaptation Methods for Cloud Applications: An Empirical Study. In *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. IEEE, 632–639.

[10] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. 2009. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering* 35, 2 (2009), 148–161.

[11] Hamoun Ghanbari, Marin Litoiu, Przemyslaw Pawluk, and Cornel Barna. 2014. Replica Placement in Cloud through Simple Stochastic Model Predictive Control. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*. IEEE, 80–87.

[12] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. 2012. Feedback-based Optimization of a Private Cloud. *Future Generation Computer Systems* 28, 1 (January 2012), 104–111. https://doi.org/10.1016/j.future.2011.05.019

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

[14] IBM. 2005. *An Architectural Blueprint for Autonomic Computing*. Technical Report. IBM. http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[15] Rudolf Kalman. 1959. On the general theory of control systems. *IRE Transactions on Automatic Control* 4, 3 (1959), 110–110.

[16] Marin Litoiu. 2013. Optimization, Performance Evaluation and Resource Allocator (OPERA). (2013). http://www.ceraslabs.com/technologies/opera

[17] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. 2011. A multi-model framework to implement self-managing control systems for QoS management. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 218–227.

[18] Lukas Reichlin. 2015. Octave Control Package v2.3.8. http://octave.sourceforge.net/control/overview.html. (2015).

[19] Joerg 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 (2010).

[20] Michael Smit, Bradley Simmons, and Marin Litoiu. 2013. Distributed, Application-level Monitoring of Heterogeneous Clouds using Stream Processing. *Future Generation Computer Systems* 29, 8 (2013), 2103–2114.

[21] C. Murray Woodside, Tao Zheng, and Marin Litoiu. 2008. Performance Model Estimation and Tracking Using Optimal Filters. *IEEE Transactions on Software Engineering* 34, 3 (2008), 391–406. https://doi.org/10.1109/TSE.2008.30