2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

KPI-agnostic Control for Fine-Grained Vertical Elasticity

Ewnetu Bayuh Lakew*, Alessandro Vittorio Papadopoulos†, Martina Maggio‡, Cristian Klein*, Erik Elmroth*
*Department of Computing Science, UmeåUniversity, Sweden.
†IDT, Mälardalen University, Sweden.
‡Department of Automatic Control, Lund University, Sweden.
Email: *{ewnetu,cklein,elmroth}@cs.umu.se, †alessandro.papadopoulos@mdh.se, ‡martina@control.lth.se.

***Abstract*—Applications hosted in the cloud have become indispensable in several contexts, with their performance often being key to business operation and their running costs needing to be minimized. To minimize running costs, most modern virtualization technologies such as Linux Containers, Xen, and KVM offer powerful resource control primitives for individual provisioning – that enable adding or removing of fraction of cores and/or megabytes of memory for as short as few seconds. Despite the technology being ready, there is a lack of proper techniques for fine-grained resource allocation, because there is an inherent challenge in determining the correct composition of resources an application needs, with varying workload, to ensure deterministic performance.**

This paper presents a control-based approach for the management of multiple resources, accounting for the resource consumption, together with the application performance, enabling *fine-grained vertical elasticity*. The control strategy ensures that the application meets the target performance indicators, consuming as less resources as possible. We carried out an extensive set of experiments using different applications – interactive with response-time requirements, as well as non-interactive with throughput desires – by varying the workload mixes of each application over time. The results demonstrate that our solution precisely provides guaranteed performance while at the same time avoiding both resource over- and under-provisioning.

I. Introduction

Vertical elasticity is about increasing or decreasing the amount of resources allocated to an application. For example, for a Virtual Machine (VM) CPU cores or memory can be added or removed. The latter is enabled by a technique called memory ballooning [10]. Similarly, Platform as a Service (PaaS) providers may change the amount of resources allocated to hosted applications using container-specific technology, such as cgroups [12]. In contrast to horizontal elasticity, which is about starting or stopping whole VMs, vertical elasticity features lower overhead and finer granularity control, allowing to allocate fractions of CPU cores and megabytes of memory for intervals as short as few seconds [21].

There are several reasons to pursue vertical elasticity. First, decreasing the granularity at which resources are allocated, both in amount and time, is known to lead to increased resource utilization efficiency [7]. Second, for cost efficiency reasons, hardware becomes increasingly dense, featuring many CPU cores and DRAM modules. Applications that only require a fraction of such a server need to be carefully co-located [26] to ensure efficient server utilization and avoid resource sprawling [38]. Academia and industry are already envisioning a form of hardware packing, called *Rack-Scale Computing*, in which the boundary of a traditional server will encompass all resources inside a rack [13].

The decrease in granularity not only brings great opportunities in increasing efficiency, but also several challenges. First, a vertical elasticity controller needs to react quickly and accurately to fluctuation in resource requirements of an application. Second, scaling needs to be performed independently in several resource dimensions, e.g., CPU and memory. Third, each application may target a different Key Performance Indicator (KPI), such as average response time, tail response time or throughput. Translating target KPIs to required resources is difficult, due to the non-linearity and the time variability of the relationship [26] and the run-time uncertainty about the actual performance delivered by the shared hardware, a phenomenon called *noisy neighbor* [22].

Previous approaches only partially dealt with these issues. They either targeted a single KPI [33], [14], a single resource dimension [18], [9], [27] and/or required expert knowledge to guide the vertical elasticity controller towards which resource dimension to act upon [16]. In this paper, we propose a KPI-agnostic methodology to design vertical elasticity controller for multiple resource dimensions. Our approach builds on well-established control-theoretical tools, such as system identification and model predictive control.

The contribution of this paper is twofold.

1) From the theoretical perspective, we propose a methodology to design an application-specific vertical elasticity controller. A rough model of the application behavior is produced off-line using system identification. The model is then used as a foundation to design a model predictive Multiple Input Multiple Output (MIMO) controller. At run-time, the controller collects resource utilization and KPI, and updates the controller behavior to compensate for run-time uncertainty (Section III-A).
2) From the practical standpoint, we evaluate the proposed approach over multiple applications – RUBiS, RUBBoS, Olio, and a video encoding service. In the studied cases, we use two KPIs – response time and

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.71

throughput – and two resource dimensions – CPU and memory. We test the applications with realistic workloads – featuring periodicity as well as spikes – and with synthetic workloads, providing a comprehensive evaluation (Section V).

Results show that the obtained vertical elasticity controller reacts quickly and adjusts CPU and memory allocation accurately, to reach the target KPIs. The remainder of this paper is organized as follows. Section II introduces the related literature, providing an overview of the differences with our approach, that is then detailed in Section III. Our experimental setup is described in Section IV, while Section V shows our results. Finally, Section VI concludes the paper.

II. Related Work

We have seen a lot of advancements in auto-scaling techniques [24] in the last decade. However, the focus has mostly been on horizontal auto-scaling and similar methods [30] which are not directly applicable for vertical auto-scaling due to their coarse-grained approaches. Vertical auto-scaling has, however, gained a lot of attention recently [25], [28], [21], [18], [39].

Delving into details, Kalyvianaki et al. [18] designed a controller using Kalman filtering to control CPU allocation based on the CPU utilization. Baresi et al. [9] use a control-theoretical method based on loop shaping to allocate CPU to the running applications in both containers and VMs, and consequently allocate a fixed amount of memory per core. The authors in [29] proposed a two stage controller to allocate CPU cores for different tiers of a multi-tier application. While the first controller regulates the relative CPU utilization of each tier, the second controller adjusts the allocations in cases of CPU contention. Yazdanov and Fetzer [39] also developed a vertical auto-scaler to allocate the right amount of CPU cores to high priority applications. Their solution is built on top of the Xen Hypervisor using a combination of CPU hot-plugging and tuning virtual CPU power to provide a finer grain control of the physical resources associated to the VM. Spinner et al. [33] proposed a model-based approach that uses the relationship between the CPU cores allocated and the observed application performance to automatically extract and update the model using resource demand estimation techniques. Lakew et al. [21] presented two generic response time performance models, queue length based and inverted response time, which map performance to CPU capacity and provide performance guarantees for interactive applications deployed in the cloud.

Some work has been done on optimal memory size allocation for a given application [27], [14], [15]. Molto et al. [27] presented a mechanism to dynamically adjust the application memory size to achieve efficient memory utilization. The authors in [14], [15] presented control theoretic approach to dynamically adjust application memory to meet response time under workload dynamics.

The authors in [25], [16] considered multi-dimensional resource (e.g., CPU and memory) vector for a single application to meet its performance targets. More precisely, Lu et al. [25] proposed a technique based on limits, reservation and shares to automatically set resource control for both VMs and resource pools to meet performance. Application performance objectives were translated into the appropriate resources, such as memory and CPU. The authors in [16] proposed fuzzy control to coordinate two different controllers each controlling CPU and memory for a single application in order to meet its response time.

However, all the aforementioned approaches either targeted a single KPI (e.g., response time [21], [16], [33], [14], [15], resource utilization [18], [27]), a single resource dimension (e.g., CPU [21], [18], [9], memory [27], [14], [15]), required expert knowledge to guide the vertical elasticity controller towards which resource dimension to act upon [16], and/or may lead to resource under- or over-provisioning during unexpected workload changes [25]. In this paper, we propose a KPI-agnostic methodology to design vertical elasticity controllers meeting the target performance of an application and considering multiple resources.

III. The control system

We consider a cloud infrastructure that hosts multiple services, each with different characteristics, as well as variable and unpredictable workload dynamics. Each service has a Service Level Agreement (SLA) that stipulates a Service Level Objective (SLO) and optionally minimum and maximum resource requirements. The minimum resource constraints are often used to allow each service to maintain some functionality at all times. The maximum limits are usually set to shield the user from unexpectedly high costs due to service malfunctioning or an attack. The SLO is a target value for a KPI – for example, a specific value for *average response time* or *throughput* of the system. The goal is to continuously adjust the allocated resource levels, without human intervention, to drive KPIs toward their targets. Specifically, the resource allocation strategy should be capable of allocating just the right amount of resources for each service at the right time in order to meet its respective performance target, avoiding both resource under- and over-provisioning.

Fig. 1 shows the architecture of the proposed resource allocation strategy. It loosely follows a Monitor Analyze Plan and Execute with Knowledge (MAPE-K) loop based on self-adaptive software terminology [19]. The monitor part is implemented with sensors gathering information about the observed KPIs and the resource consumption. The monitor collects measurements periodically, the period between observations is also used as the period between

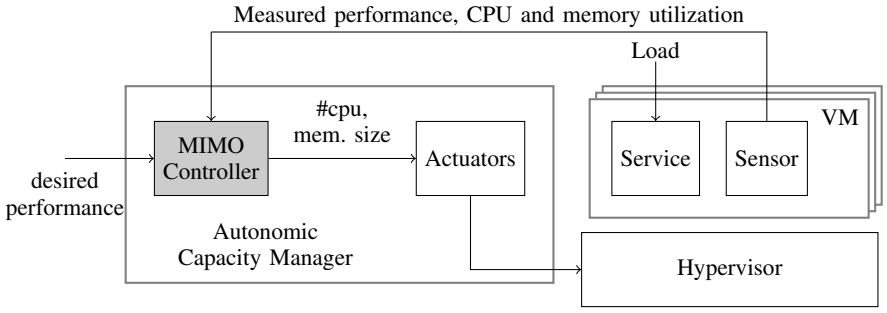


Figure 1. The architecture of the system. The contribution of this work is shown in gray.

activations of the controller. The analysis and planning phase is left to the MIMO controller, that computes the CPU and memory levels for the next time interval. Execution consists in configuring the Hypervisor to enforce the computed resources. Previous monitoring data is used to fit the model parameters, which represent the knowledge component in the MAPE-K loop. To easily integrate the controller under different cloud frameworks such as Kubernetes, one instance of the controller can be deployed for each Kubernetes node.

A. Controller design

This subsection describes in details the Multiple Input Multiple Output (MIMO) controller. The design of the controller is performed in two distinct phases. First, we collect data about the application running conditions and we use system identification to build a MIMO model of the applications' response to changes in resources. This phase is carried out offline, while the application is working but there is no control, to capture the typical operating conditions. We then use that model to synthesize a controller that is executed periodically and informs the hypervisor about the amount of CPU cores and memory to allocate to the running VM.

For the model construction, we use the MOESP algorithm [37] as the identification method. The method provides us with a linear MIMO model based on the data we collected offline. We select the lowest possible order that fits the data, which we denote with n, to limit the model size[1]. Selecting a higher order increases the accuracy of the model, together with the risk of overfitting. In the obtained model, the system has n states and a state vector $x = [x_1, x_2, \ldots, x_n]$. The states are not linked to any meaningful quantity in the system but describe the dynamic relationship between the inputs (the amount of resources given) and the output (the KPI measured).

Formally, the MOESP algorithm returns a model $\mathscr{S}$ in the difference equation form

$$\mathscr{S} : \begin{cases} x(k+1) = A \cdot x(k) + B \cdot res(k) \\ meas(k) = C \cdot x(k) \end{cases}, \qquad (1)$$

[1]The subspace identification procedure relies on the Matlab function `n4sid`, using as parameters the given data and the keyword `best` for the model order.

where $res(k)$ is a vector denoting the amount of resources to be given to the application and $meas(k)$ is a vector that includes the current KPI values and the percentages of CPU and memory utilization to be informed on and able to prevent over-provisioning. The model uses Δt as the sampling interval, which is the distance in time between two subsequent measurements in the data trace that we collected for identification purposes. The time is represented with the letter k, that denotes the sampling instants and assumes values in the set of integers where a number k is the instant $t = \Delta t \cdot k$ in time.

Based on the model $\mathscr{S}$, we can synthesize a Model Predictive Control (MPC) in the augmented velocity form, with standard techniques [11]. MPC is a control technique that formulates an optimization problem to use the available actuators (the resources) to achieve a set of goals (the KPIs). At every control instant k, the problem becomes the minimization of a cost function F_k, subject to given constraints. Having a controller in the augmented velocity form informally means that the controller is going to compute the variations that should be applied to the control signals (the resources allocated to the application), denoted with $\Delta res(k)$ rather than their absolute values $res(k)$. This controller is complemented with a Kalman Filter [23] to update the system model as the controller runs.

The first equation in Eq. (1) describes how $x(k+1)$ is function of $x(k)$, while in the augmented velocity form $\Delta x(k+1) := x(k+1) - x(k)$ is as follows:

$$\overbrace{\begin{bmatrix} \Delta x(k+1) \\ meas(k) \end{bmatrix}}^{\tilde{x}(k+1)} = \overbrace{\begin{bmatrix} A & 0_{n\times 3} \\ C & I_{3\times 3} \end{bmatrix}}^{\tilde{A}} \overbrace{\begin{bmatrix} \Delta x(k) \\ meas(k-1) \end{bmatrix}}^{\tilde{x}(k)} + \overbrace{\begin{bmatrix} B \\ 0_{3\times 2} \end{bmatrix}}^{\tilde{B}} \Delta res(k)$$

$$meas(k) = \overbrace{\begin{bmatrix} C & I_{3\times 3} \end{bmatrix}}^{\tilde{C}} \overbrace{\begin{bmatrix} \Delta x(k) \\ meas(k-1) \end{bmatrix}}^{\tilde{x}(k)} \qquad (2)$$

where $0_{r\times c}$ is an all-zeroes matrix with r rows and c columns, while $I_{p\times p}$ is the identity matrix of size p. Here, $\Delta x(k) = x(k) - x(k-1)$ is the state variation and $\Delta res(k) := res(k) - res(k-1)$ is the control increment. The augmented velocity form is typically used for the formulation of MPC

controllers, since it guarantees that the controlled system reaches all the goals when possible. The system output $meas(k)$ is unchanged but now expressed with respect to the state variations $\Delta x(k)$ and not with respect to the state values $x(k)$. This new model is used by the MPC controller to predict the future state of the system with a time horizon of L steps, a parameter that is configurable in our approach.

Denoting with $kpi_{m,k}$ the measured value of the KPIs and with kpi_k the desired value for them[2], the MPC minimizes the following cost function

$$F_k = \sum_{i=1}^{L} \left[kpi_{k+i} - kpi_{m,k+i}\right]^\top Q_i \left[kpi_{k+i} - kpi_{m,k+i}\right] + \left[\Delta res_{k+i-1}\right]^\top R_i \left[\Delta res_{k+i-1}\right], \tag{3}$$

where $Q_i \in \mathbb{R}^{3,3}$ and $R_i \in \mathbb{R}^{2,2}$ are a symmetric positive semi-definite weighting matrices. Q_i represents the weights on the considered KPIs and R_i represents the inertia to changing the actuators. The superscript $^\top$ used in (3) indicates the transpose operator for a vector.

Q_i and R_i are chosen to prioritize KPIs and resource usage. The values in Q_i are typically chosen to specify a weight for the considered KPIs. For example, one may configure the controller to prefer reaching a specific average response time, even if this may result in allocating more CPU cores. The matrix R_i indicates preferences on actuators, i.e., how reactive the controller should be in changing each resource specific allocation. For example, this allows one to configure the controller to be very quick in assigning new cores, but be more conservative with changing the memory allocation.

The resulting optimization problem is

$$\begin{aligned} \text{minimize}_{\Delta res_{k+i-1}} \quad & F_k \\ \text{subject to} \quad & res_{\min} \leq res_{k+i-1} \leq res_{\max} \\ & \Delta res_{\min} \leq \Delta res_{k+i-1} \leq \Delta res_{\max} \\ & \tilde{x}_{k+i} = \tilde{A} \cdot \tilde{x}_{k+i-1} + \tilde{B} \cdot \Delta res_{k+i-1} \\ & kpi_{m,k+i-1} = \tilde{C} \cdot \tilde{x}_{k+i-1} \\ & i = 1, \ldots, L, \end{aligned} \tag{4}$$

where the constraints limits the amount of changes in the actuators and their absolute values, and impose that the model dynamics are followed for the solution finding. For example, a VM cannot have more cores than the physical machine it is deployed onto and the user may want the number of cores not to increase more than a certain value per iteration.

This formulation is equivalent to a convex Quadratic Programming (QP) problem [20]. The solution of the QP problem is an optimal plan for the future $\Delta res^\star_{k+i-1}$, $i = 1, \ldots, L$. A *receding horizon* [20] approach is adopted, and only the first action of the plan, $\Delta res^\star_k$, is applied. This is to cope with disturbances that may occur at runtime and have not being part of the model formulation.

[2]The vector of KPIs and desired values for them includes instances like the response time or the throughput and also the percentages of CPU and memory utilizations with respect to the allocated resources, for which the desired values are 100%.

The MPC strategy assumes that the process state x is measurable, but this is not possible, since the system state has a non-trivial interpretation. Instead of measuring x, we then estimate it based on the values of res_m and of the measured kpi_m. We designed a Kalman filter that computes an estimate $\hat{x}(k+1)$ of the state $x(k+1)$, as a function of $res(k)$, $kpi_m(k)$, and the estimation error $kpi_m(k) - \widehat{kpi_m}(k)$:

$$\begin{aligned} \widehat{kpi_m}(k) &= C \cdot \hat{x}(k) \\ \hat{x}(k+1) &= A \cdot \hat{x}(k) + B \cdot res(k) + K(k) \cdot \left(kpi_m(k) - \widehat{kpi_m}(k)\right) \end{aligned} \tag{5}$$

where $K(k)$ is called Kalman gain and changes over time according to the estimation error [23]. The estimate $\hat{x}(k)$ can be used, in place of $\tilde{x}(k)$, to solve the optimization problem in Eq. (4).

Overall, the input to the MPC controller are: (a) the model that is computed offline with the subspace identification method, (b) the prediction horizon L, (c) the values of Q_i and R_i, that determines the weights of the different entities involved in the optimization, and (d) the constraints $res_{\min}$, $res_{\max}$, $\Delta res_{\min}$ and $\Delta res_{\max}$ used to limit the solutions to the effectively applicable ones.

B. Controller Implementation

This subsection describes the practical implementation of the controller and the values of the parameters used in the experimental campaign for the different applications. The value of the prediction horizon L determines how far ahead in time the controller is supposed to look at. This parameter can be chosen keeping in mind that a larger value of L will result in a better control plan, but at the same time in higher computational overhead. In our experiments, we chose the value of 15, meaning that the controller determines a plan for the next 15 time steps and then applies only the first instance of that plan, to always be able to cope with runtime variations. The controller is executed every 10 seconds, therefore having a prediction horizon of 15 steps means predicting the behavior of the system in the next 2.5 minutes.

The values of $res_{\min}$, $res_{\max}$, $\Delta res_{\min}$ and $\Delta res_{\max}$ determine the physical limitations of the controller both in absolute terms and in relative terms (the controller is not able to allocate more than a certain number of CPUs per application, and it is also not able to vary the number of CPUs more than a given amount per execution). For our experiments, the controller can allocate a minimum of 0.5 CPUs and a maximum of 25 and is able to add and remove 24 cores in a single instant. For the memory, the limitations are a mimum of 0.7 GB and a maximum of 8 GB, with a minimum and maximum variation of 7 GB at a time.

In our experimental evaluation, we use different applications, see Section IV for more details. The values in Q_i and R_i are chosen differently for each application, by

looking at the identification data. For RUBiS and Olio, the values in Q_i are $[1000, 0.1, 0.1]$, the higher value giving more importance to the goal. The goals in this case are respectively "keeping the response time to a target", "using all the given CPU", and "using all the given memory". The weight 1000 multiplies a number of the order of 1 second, while the two weights at 0.1 multiplies a number in the range $[0 \dots 100]$. This therefore determines that response time is two orders of magnitude more important than keeping the resource utilization to its target. In the case of RUBBoS, the values in Q_i are $[15, 0.1, 0.1]$ as we have seen that the response time varies less than in the other interactive applications, and we can therefore focus more on resource-related goals. For the video encoder, the values in Q_i are $[100, 0.5, 0.5]$, because of similar considerations.

Finally, the weights R_i are determined so that the controller is very quick in assigning new cores while it is more conservative with memory allocation, as recovering from memory under-provisioning (swapping) is more costly. For all the interactive applications, the values of R_i are $[0.1, 10]$. For the video encoder, due to the high variability of the load, we want to be slightly more conservative in assigning CPUs as well, so we chose a weight vector $R_i = [0.5, 10]$.

In general, the Q_i and R_i values were different for each application since they capture the intrinsic and complex relationships between the applications and resources. In an ideal scenario, the goal of the controller is to ensure that the application meets its target performance while both CPU and memory utilizations are at 100%. However, as predicted by queuing theory, low response times cannot be obtained under high CPU utilization, whereas a high memory utilization leads to the risk of swapping and poor performance. Therefore, to mitigate the conflict between utilization and performance, more priority is given to the application performance by providing more weight than the two utilizations. As a result the control strategy ensures that the application meets its target performance while keeping both CPU and memory utilizations as high as possible. Further information on how tune the different control parameters can be found in [8].

IV. Experimental Setup

Experiments were conducted on a single Physical Machine (PM) equipped with a total of 32 cores[3] and 56 GB of memory. To emulate a typical cloud environment and easily enable vertical elasticity, we used the Xen hypervisor [10]. Each tested application was deployed with all of its components such as web servers and database servers inside its own VM as is commonly done in practice [34], e.g. by using a LAMP stack [1].

To demonstrate the applicability of the proposed solution for different types of KPIs, we used two types of applications: interactive applications – requiring a target response-time – and a video application – requiring a target throughput.

[3]Two AMD Opteron™ 6272 processors, 2100 MHz, 16 cores each, no hyper-threading.

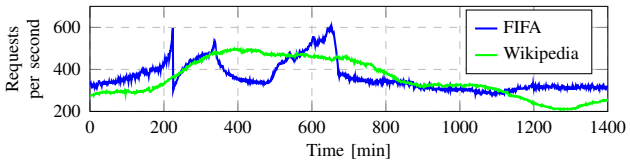


Figure 2. Workloads from real world traces:Predictable, Wikipedia-based and Unpredictable, FIFA-based traces.

A. Interactive Applications

This type of applications only perform computations as a result of a user request. We performed a wide range of experiments using three different interactive applications: RUBiS [4], RUBBoS [5] and Olio [2]. These applications are widely-used cloud benchmarks – see e.g., [17], [32], [36], [35] – and respectively represent an eBay-like e-commerce application, a Slashdot-like bulletin board, and an Amazon-like book store.

We performed experiments with different workloads to characterize the performance models' responses to workload changes. Specifically, we used real workload from traces and synthetic workload based on the open and closed-system models [31]. The real workloads were extracted from the Wikipedia [3] and FIFA [6] traces. These two traces were selected due to their complementary nature. While the Wikipedia workload shows a periodic and predictable trend, the FIFA trace shows a bursty and an unpredictable trend. For each application, we selected a representative day as shown in Figure 2, to focus on different trends and peaks, and to show that not all the applications may have the same workload patterns. Note that, although the workload traces are part of the evaluation, they are considered known only a posteriori, i.e., the controller cannot access them neither at design time nor during execution. In addition, synthetic workloads were generated based on the open- and closed-system models that gave us the freedom of evaluating situations that were not present in the real world workloads, such as increasing the number of requests five- or ten-fold. Combined with periods of steady load, the synthetic workloads allow us to better study the behavior of the controller in both steady- and transient-state.

To emulate the users accessing the applications, under the synthetic workload, we used our custom `httpmon` workload generator[4], which supports both open- and closed-system model client behavior. For open clients, we changed the arrival rate during the course of the experiments as required to stress-test the system. For the closed case, the think-time of each client was fixed at 1 second and the number of users was varied. The change in arrival rates or number of users was made instantly. This made it possible

[4]https://github.com/cloud-control/httpmon

Table I
MEAN (M) AND VARIANCE (V) FOR RESPONSE TIME (RT) AND RESOURCE UTILIZATION FOR MIMO AND FUZZY CONTROLLERS WITH THE CLOSED AND OPEN WORKLOADS.

Controller	Workload	RT		CPU Util		Mem Util	
		M	V	M	V	M	V
MIMO	Closed	0.61	0.06	95.82	34.4	56.3	174.8
Fuzzy	Closed	0.66	0.45	81.8	247.4	67.4	29.1
MIMO	Open	0.51	0.03	92.0	169.0	64.8	371.0
Fuzzy	Open	0.57	0.35	82.2	138.6	52.6	22.4

to meaningfully compare the system's behavior under the two client models.

The *response time* of a request – our KPI – is defined as the time that passes between sending the first byte of the request and receiving the last byte of the reply. In our evaluation, we focus on average response times per minute.

B. Non-Interactive Applications

For this type of application we used a video encoding application, which requires a certain target frame-per-second. The video itself changes in complexity, which induces varying CPU and memory requirements. The *throughput* of a video – our KPI – is defined as the number of frames computed per second (fps).

V. EXPERIENTAL EVALUATION

In this section we highlight the dynamic behavior of the controller and show how it responds to varying application requirements. The plots in this section are structured as follows. Each figure shows the results of a single experiment. The bottom x-axis represents the time elapsed since the start of the experiment. The upper graph in each figure plots the mean response times or frames-per-second while the middle and lower graphs plot the amount CPU cores and memory (in GB) allocated to the application as computed by the controller, respectively. Besides, for graphs that shows resource utilization information, the graph right below the upper graph plots the average CPU and memory utilization.

A. Comparison with state-of-the-art techniques

In this section, we present the experimental results of our proposed MIMO controller in comparison with the fuzzy controller presented in [16]. The fuzzy controller tries to meet target response times by coordinating two separate resource controllers, a separate memory and CPU controller using fuzzy logic.

The evaluation goal is to compare how the *fuzzy controller* and our new *MIMO controller* meet the desired *response time* objective and manage to use a minimum amount of resources. If the response time goal is met, a controller is better than the other if it uses less resources.

Fig. 3 shows the results of the two controllers with 0.5s target under open and closed system models for RUBiS application. The response time remains stable and close to the target value in the MIMO controller case, while more oscillations are observed using the fuzzy controller. The oscillations for the fuzzy controller are also noticeable in the CPU allocation and utilization. Moreover, in the MIMO controller case, the response time converges to the target value immediately when the workload changes. In the fuzzy controller case, the response time spikes are noticeable under sudden workload surge, and the convergence time is higher.

Table I shows the aggregate means and variances of response time and utilization during the life time of the experiments. The MIMO controller allocates less CPU cores on average, as can inferred from the high utilization value, while achieving a better performance with low variability than the fuzzy controller. The fuzzy controller appears more efficient in terms of memory for closed workload. This is due to the MIMO controller being more conservative with memory, performing a higher initial allocation and reducing memory allocation rather slowly. However, the MIMO controller does slowly reduce memory allocated to the application, until it reaches an optimal steady state. Figure 3 shows indeed that less or comparable memory is allocated after 340 seconds have passed. The longer the MIMO controller executes, the more efficient it becomes.

In general, during the offline system identification phase, the intrinsic relationship between application performance, in this case response time, and resources is fully captured by the MIMO controller, while this relationship requires expert knowledge – which is often subjective – to train the fuzzy controller. In addition, the fuzzy controller is designed for a single KPI (i.e., response time) while the MIMO controller is KPI-agnostic. The following sections will present a thorough analysis of our new MIMO controller for both response-time- and throughput-oriented applications.

B. Interactive Services under Real Workloads

Fig. 4 show the allocated resources and observed response times when RUBBoS is subject to both the periodic and predictable Wikipedia workload, as well as the more unpredictable and bursty FIFA one, for target response times of 1.5s and 0.5s. The results show that the controller was able to allocate the resources that satisfies the performance bounds under both workloads. The utilization for both resources is very high indicating efficient use of resources. Besides, a close look at the results in the figures show that the CPU cores allocated follow similar trend as the corresponding workload patterns depicted in Fig. 2 indicating that the application is more sensitive to CPU than memory allocation. This phenomenon is also reflected in the fact that a decrease in the response time target from 1.5s to 0.5s shows a significant increase in the number of CPU cores required for comparable workload pattern while memory is invariable. Another important point to note is that the controller is very conservative when allocating memory since the application acquires and releases memory slowly. This situation is reflected by how the memory utilization mimics the workload pattern.

A closer observation to the figures shows that the results in Fig. 4b exhibit more oscillation in the observed response time around the target values than the corresponding results

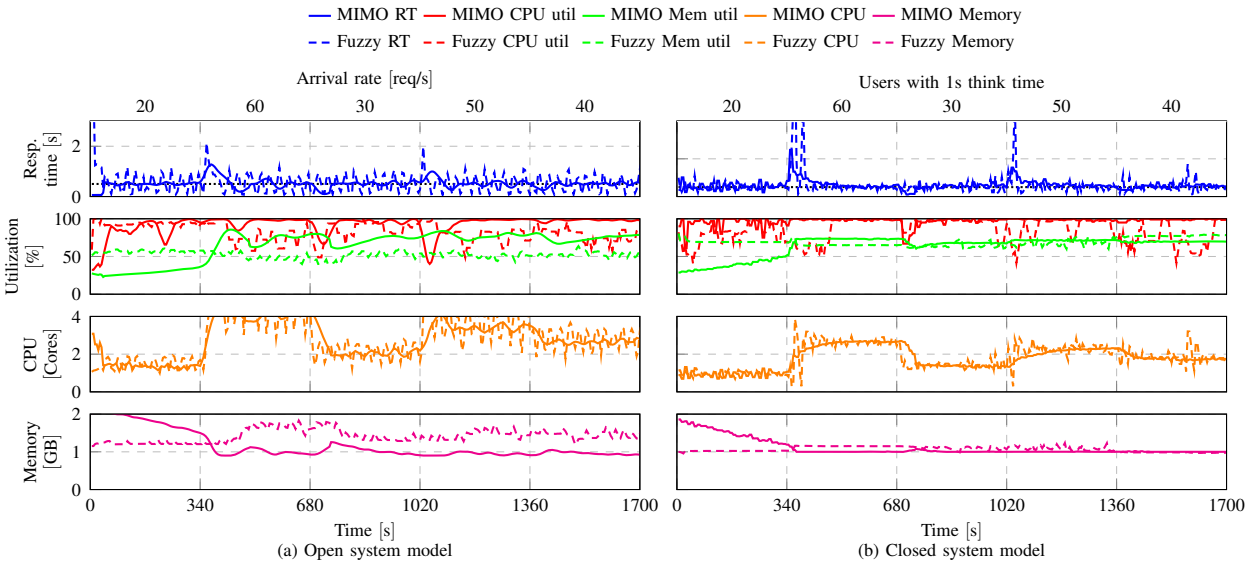


Figure 3. RUBiS–under open and closed system models. The target response time is set to 0.5s for both controllers.

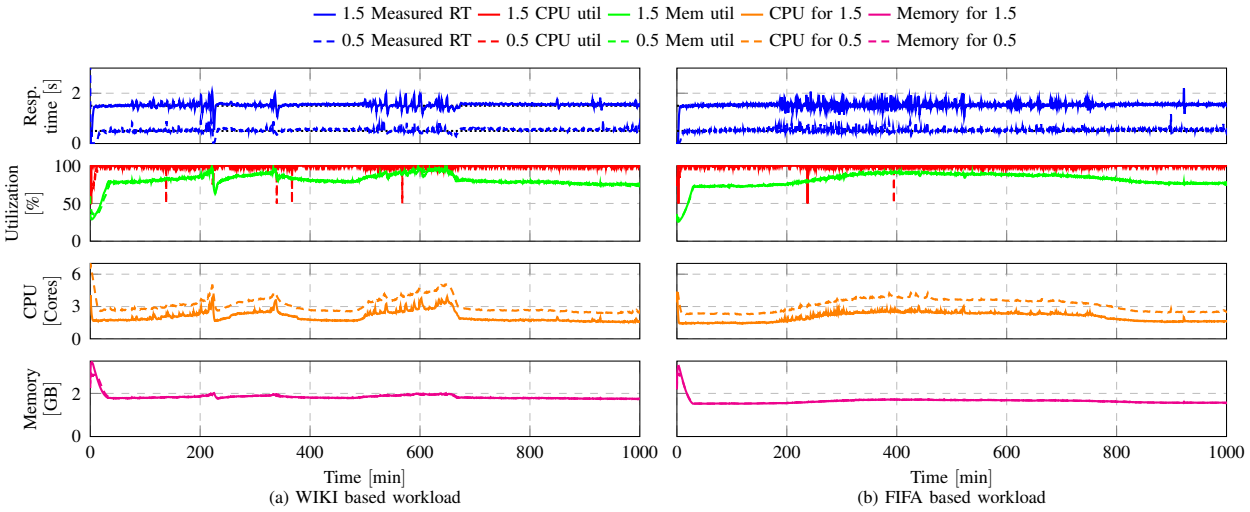


Figure 4. RUBBoS–under WIKI and FIFA workloads with 1.5s and 0.5s target response times.

in Fig. 4a. This is due to the unpredictability of the FIFA workload. However, the controller behaves as intended even under unpredictable workloads.

C. Interactive Services under Synthetic Workloads

Fig. 5 shows the results when the controller was configured with 1.0s and 1.5s target response times under open- and closed-system models for the RUBBoS application. The results show that the response times converge to their respective target values quite quickly after detecting a sudden increase or decrease in workloads (manifested as a rapid increase or decrease in the response time values) as depicted in the figures at the beginning of each interval. This indicates that the controller correctly detects changes in demands and allocates the optimal amount of the resources to meet the performance targets for both open- and closed-system models.

Close observation of the results show that the open-system model required more CPU cores than the closed-system model when using comparable arrival rates and numbers of users, respectively. Moreoever, the controller was able to determine the right amount of memory for both models which happens to be is relatively the same for both models for the evaluated applications.

We also performed experiments with Olio and RUBiS. Due to the similarity of the results to RUBBoS application, we only present time series plots generated using a target response time of 1.0*s* for these applications. As shown in Figs. 6 and 7, the results show that the controller behave as

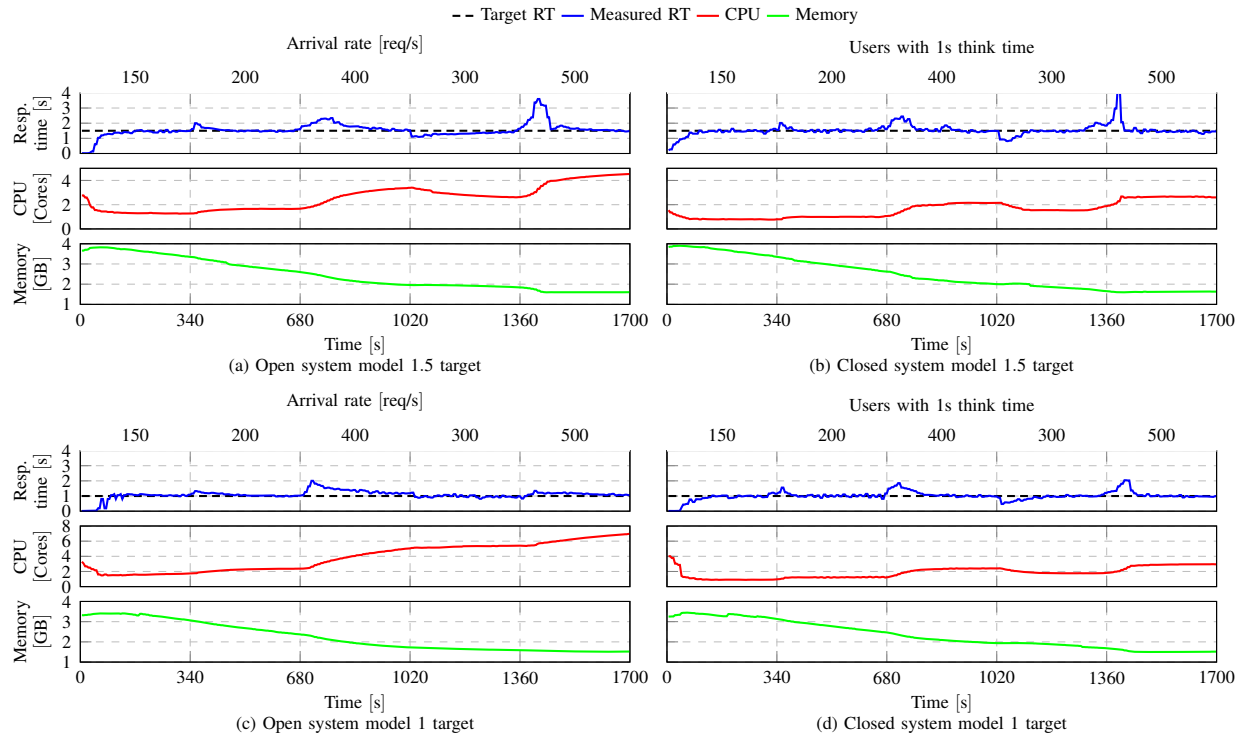


Figure 5. RUBBoS–under open and closed system models with 1.0s and 1.5s target response times.

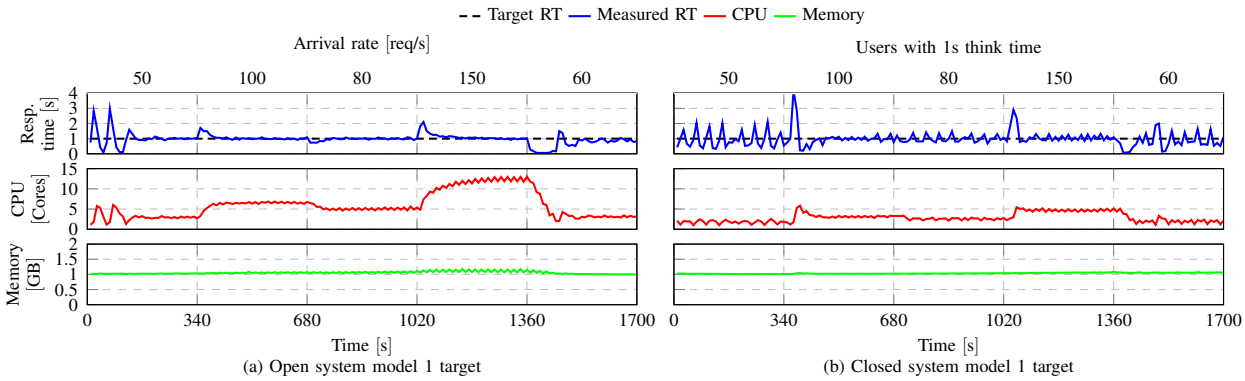


Figure 6. RUBiS–under open and closed system models with 1.0s target response time.

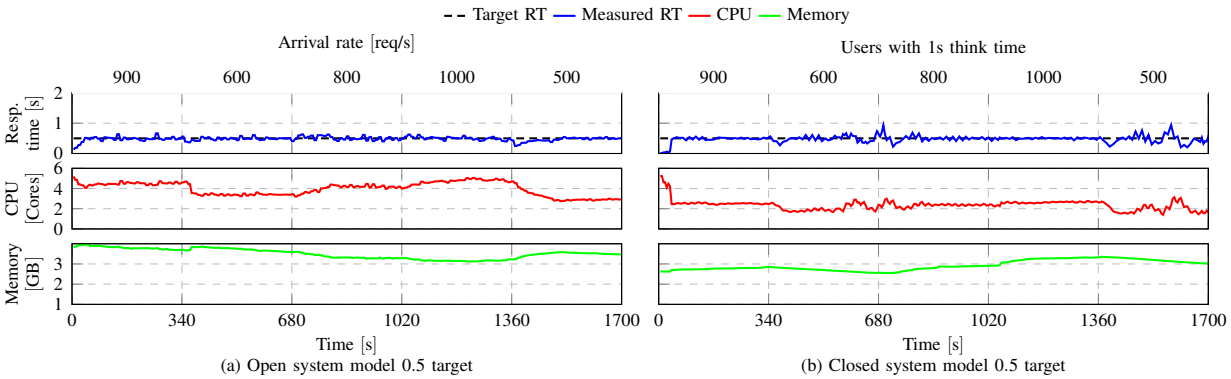
intended.

D. Non-interactive applications

Fig. 8 shows the results obtained when throughput is used as KPI, i.e., number of frames per second (fps) for the video application. We ran the experiment using two different throughput target values in order to see how the controller behaves. In the first experiment we set the target to 800fps, whereas in the second experiment the target was set half, 400fps. We ran each experiment until all videos in a predefined list were converted. The total duration can be inferred from the length of time each experiment took, and, as expected, the 400fps target took twice as long as the 800fps target.

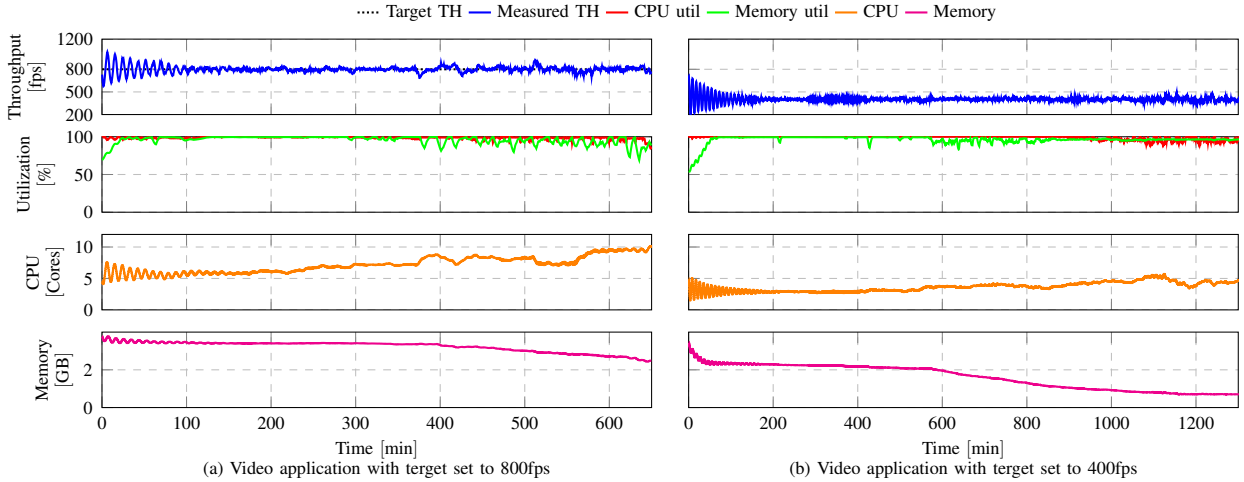
The target throughputs were met for both target values indicating that the controller provides accurate capacity estimation irrespective of the target value set. A closer observation of the figure show that there is much oscillation at the beginning, due to the controller needing some time to capture the behavior of the application.

Looking at the resources required to meet each targets, as expected, higher target values need more resources than lower target values. Specifically, meeting the 800fps target requires twice the number of CPU cores and 40% more memory compared to meeting the 400fps target. The con-



(a) Open system model 0.5 target (b) Closed system model 0.5 target

Figure 7. Olio–under open and closed system models with 0.5s target response time.



(a) Video application with terget set to 800fps (b) Video application with terget set to 400fps

Figure 8. Video encoding application with different targets: 800fps and 400fps.

troller did manage to allocate the right amount of both CPU and memory, as can be observed through the utilization plot. The results show that the controller is able to maintain the performance targets of the application no matter what affects it i.e., whether the disturbance is from within the cloud (e.g., noisy neighbours) or outside (e.g., intensity of the video scenes).

E. Aggregate Analysis

To see the aggregate behavior of the controller over the course of the experiment, we use mean and variance as shown Table II. On average the observed performance of applications is very close to the target. Response time oriented applications observed very low variance. In contrast, the variance for throughput-oriented application was high due to the high oscillation at the initial stage of the experiment (see Fig. 8) until the controller learns the behavior of the application.

VI. Conclusion

This paper presented a KPI-agnostic Model Predictive Multiple Input Multiple Output (MIMO) Controller to automatically adjust the amount of CPU cores and memory that an application receives to meet its performance requirement. We carried out an extensive set of experiments using response time and throughput as performance indicators. For interactive applications, we varied the workload mix using both real and synthetic workloads. For non-interactive

Table II
Observed aggregate performance of applications under MIMO controller.

Application	Workload	Target	Mean	Variance
RUBBoS	Wikipedia	1.500	1.511	0.019
		0.500	0.469	0.009
	FIFA	1.500	1.328	0.041
		0.500	0.326	0.014
	Open	1.500	1.504	0.087
		1.000	1.027	0.080
	Closed	1.500	1.457	0.229
		1.000	1.030	0.168
RUBiS	Open	1.000	0.976	0.144
	Closed	1.000	0.985	0.264
Olio	Open	0.500	0.461	0.015
	Closed	0.500	0.478	0.027
Video	–	800.000	799.170	1832.910
	–	400.000	402.850	1235.530

applications, we varied the target throughput value. The results show that our MIMO controller is able to allocate the right amount of resources to meet the performance targets irrespective of the selected KPI, while keeping the utilization of the resources – CPU and memory – high. We have compared our solution with a state-of-the-art controller implementation and showed that the proposed MIMO solution is more efficient in terms of resource utilization. Future work includes extending the resource dimension to network bandwidth slicing and guaranteeing tail values of KPIs for more accurate control.

Acknowledgement: This work was partially supported by the Swedish Research Council (VR) for the projects *Cloud Control* and *Power and temperature control for large-scale computing infrastructures*, through the *LCCC Linnaeus* and *ELLIIT* Excellence Centers, by the Swedish Government's strategic effort *eSSENCE*, by the Swedish Foundation for Strategic Research under the project "Future factories in the cloud (FiC)" with grant number GMT14-0032, and by the Wallenberg Autonomous Systems Program.

REFERENCES

[1] Tutorial: Installing a LAMP web server, 2013. available online: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html.

[2] Olio, 2014. Available online: http://incubator.apache.org/projects/olio.html.

[3] Page view statistics for wikimedia projects, 2014. Available online: http://dumps.wikimedia.org/other/pagecounts-raw/, Visited 2014-10-20.

[4] Rice university bidding system, 2014. Available online: http://rubis.ow2.org.

[5] Rubbos, 2014. Available : http://jmob.ow2.org/rubbos.html.

[6] Worldcup98, 2014. Available online: http://ita.ee.lbl.gov/html/contrib/WorldCup.html, Visited 2014-10-20.

[7] O. Agmon Ben-Yehuda. The resource-as-a-service (RaaS) cloud. In *HotCloud*, 2012.

[8] K. Angelopoulos, A. V. Papadopoulos, V. E. Silva Souza, and J. Mylopoulos. Model predictive control for software systems with cobra. In *SEAMS*.

[9] L. Baresi et al. A discrete-time feedback controller for containerized cloud applications. FSE, pages 217–228, 2016.

[10] P. Barham et al. Xen and the art of virtualization. In *SOSP*. ACM, 2003.

[11] E. Camacho and C. Bordons. *Model Predictive Control*. Springer London, 2004.

[12] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support PaaS. In *IC2E*, 2014.

[13] B. Falsafi, T. Harris, D. Narayanan, and D. A. Patterson. Rack-scale Computing. *Dagstuhl Reports*, 5(10):35–49, 2016.

[14] S. Farokhi et al. Performance-based vertical memory elasticity. In *ICAC*, pages 151–152, 2015.

[15] S. Farokhi et al. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *FGCS*, 65, 2016.

[16] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth. Coordinating cpu and memory elasticity controllers to meet service response time constraints. In *ICCAC*, 2015.

[17] Z. Gong et al. PRESS: Predictive elastic resource scaling for cloud systems. In *CNSM*. IEEE, 2010.

[18] E. Kalyvianaki, T. Charalambous, and S. Hand. Adaptive resource provisioning for virtualized servers using Kalman filters. *ACM TAAS*, 9(2):10:1–10:35, 2014.

[19] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[20] B. Kouvaritakis and M. Cannon. *Model Predictive Control – Classical, Robust and Stochastic*. Springer International Publishing, 2016.

[21] E. B. Lakew et al. Towards faster response time models for vertical elasticity. In *UCC*, pages 560–565, 2014.

[22] A. Le-Quoc et al. The top 5 AWS EC2 performance problems. Technical report, Datadog Inc, 2013.

[23] L. Ljung. *System Identification: Theory for the User*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[24] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.*, 12(4):559–592, 2014.

[25] L. Lu et al. Application-driven dynamic vertical scaling of virtual machines in resource pools. In *NOMS*, 2014.

[26] J. Mars et al. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. MICRO, 2011.

[27] G. Moltó et al. Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. *Procedia Computer Science*, 18:159–168, 2013.

[28] H. Nguyen et al. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

[29] P. Padala et al. Adaptive control of virtualized resources in utility computing environments. In *SIGOPS Operating Systems*, volume 41, pages 289–302, 2007.

[30] I. Pietri and R. Sakellariou. Mapping virtual machines onto physical machines in cloud computing: A survey. *ACM Comput. Surv.*, 49(3):49:1–49:30, Oct. 2016.

[31] B. Schroeder et al. Open versus closed: A cautionary tale. In *NSDI*, 2006.

[32] Z. Shen et al. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*. ACM, 2011.

[33] S. Spinner et al. Runtime vertical scaling of virtualized applications via online model estimation. In *SASO*, 2014.

[34] K. Sripanidkulchai et al. Are clouds ready for large distributed applications? *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[35] C. Stewart et al. Exploiting nonstationarity for performance prediction. In *EuroSys*. ACM, 2007.

[36] N. Vasić et al. DejaVu: accelerating resource allocation in virtualized environments. In *ASPLOS*. ACM, 2012.

[37] M. Verhaegen. Identification of the deterministic part of MIMO state space models given in innovations form from input-output data. *Automatica*, 30(1):61–74, 1994.

[38] W. Vogels. Beyond server consolidation. *Queue*, 6(1):20–26, Jan. 2008.

[39] L. Yazdanov and C. Fetzer. Vertical scaling for prioritized VMs provisioning. In *CGC*, pages 118–125, 2012.