FCMS: a Fuzzy Controller for CPU and Memory Consolidation under SLA Constraints

Please, cite this paper as:
Cosimo Anglano, Massimo Canonico, Marco Guazzone,
"FCMS: a Fuzzy Controller for CPU and Memory Consolidation under SLA Constraints,"
Concurrency Computat.: Pract. Exper., 29(5):e3968, 2017.
DOI:10.1002/cpe.3968
Publisher: https://doi.org/10.1002/cpe.3968

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:2–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.3968

FCMS: a Fuzzy Controller for CPU and Memory Consolidation under SLA Constraints

Cosimo Anglano, Massimo Canonico and Marco Guazzone*

Department of Science and Technological Innovation, University of Piemonte Orientale, Italy

SUMMARY

Cloud Providers (CPs) rely on server consolidation (the allocation of several Virtual Machines (VMs) on the same physical server) to minimize their costs. Maximizing the consolidation level is thus become one of the major goals of CPs. This is a challenging task since it requires the ability of estimating, in a resource contention scenario, multidimensional resource demands for multi-tier cloud applications that must meet Service Level Agreements (SLAs) in face of non-stationary workloads. In this paper, we cope with the problem of jointly allocating CPU and memory capacity to (a) precisely estimate their capacity required by each VM to meet its SLAs, and (b) coordinate their allocation to limit the negative effects due to the interactions of dynamic allocation mechanisms, which, if ignored, can lead to SLA violations. We tackle this problem by devising FCMS, a feedback fuzzy controller that is able to dynamically adjust the CPU and memory capacity allocated to each VM in a coordinated way, to precisely match the needs induced by the incoming workload. By means of an extensive experimental evaluation, we show that FCMS is able to achieve the above goals and works better than existing state-of-the-art alternative solution in all the considered experimental scenarios. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Cloud computing; resource management; feedback control; fuzzy control; server consolidation; virtualized cloud applications

1. INTRODUCTION

One of the key factor for the success of the *cloud computing* paradigm is its ability to provide the appearance of infinite computing resources available on demand, thereby eliminating the need for its users to make long-term provisioning plans for their applications [1, 2, 3].

To provide such illusion, *Cloud Providers* (CPs), like Amazon [4] and Rackspace [5], host customer applications on their data centers by encapsulating each one of them into a set of *Virtual Machines* (VMs), that are run on their physical infrastructures. To increase their profit, CPs typically resort to *server consolidation* [6], which consists in allocating several VMs on each physical server, in the attempt to use as little servers as possible to run the VMs of their customers. Server consolidation allows a CP to reduce the number of physical resources that must be turned-on to run the VMs of its customers and, at the same time, to raise their utilization, so as to reduce both their electricity and amortized costs [7, 8, 9, 10]. The maximization of the *consolidation level*, that is achieved by allocating on each physical server as many VMs as possible, has thus become one of the major goals of CPs.

However, the consolidation level cannot be increased freely. As a matter of fact, typically CPs negotiate with customers suitable *Service Level Agreements* (SLAs) – specifying the minimum

*Correspondence to: Viale T. Michel, 11 – 15121 Alessandria (Italy). Phone: +39 0131 360484. E-mail: marco.guazzone@di.unipmn.it

level of service that must be guaranteed – for their applications [11]. An SLA consist of several components, including a description of the services to be delivered, the monetary penalties that a CP must pay in case of violations, and the *Service Level Objectives* (SLOs), expressing the measurable service level characteristics of the SLA (e.g., stated in terms of availability and performance), that must be fulfilled by the CP in order to avoid paying penalties to its customers. For example, the SLA for Amazon EC2 [12] specifies that the *Monthly Uptime Percentage* (MUP) during any monthly billing cycle must be of at least 99.95%, and that, in the event Amazon EC2 does not meet this goal, the customer is eligible to receive a service credit. In this case, the SLA is the whole statement and the associated SLO defines the constraint that $MUP \leq 99.95\%$ in a monthly billing cycle.

Consequently, when consolidating these VMs on their physical infrastructures, CPs have to consider the trade-off existing between the number of VMs allocated on a given server and the amount of physical resource capacity that is assigned to each one of them. Indeed, the higher the consolidation level, the lower the amount of physical server capacity that can be allocated to each VM that, if too low, may yield to violations of its SLAs. The best consolidation level that can be attained on a given physical server is therefore the one that maximizes the number of VMs allocated on it, without inducing SLA violations in the corresponding applications.

Finding the best consolidation level is a challenging task, as it requires the ability of both (a) precisely estimating the smallest amount of physical capacity each VM needs to meet its SLAs in face of highly dynamic, non-stationary and bursty workloads [13, 14], and (b) enforcing its allocation when multiple VMs compete for the same physical resources.

One of these challenges is posed by the multidimensional resource demand of applications (e.g., CPU and memory capacity, as well as network and I/O bandwidth), that calls for strategies able to coordinate the allocation of multiple resource types to the various applications.

In this paper, we focus on the problem of jointly allocating CPU and memory capacity, that are usually considered the most representative ones for determining both application performance and the effective usage of physical servers [15, 16]. To suitably solve this problem, it is necessary to address two distinct issues, namely to (a) precisely estimate and allocate the CPU and memory capacity required by each VM to meet the SLAs of the application it runs under non-stationary workloads, and (b) coordinate the allocations of CPU and memory to limit the negative effects due to the interactions of the mechanisms used to carry out them, which, if ignored, can affects application performance and thus lead to the violation of SLAs [17, 18] (as discussed in Section 2).

This problem has been already addressed in the literature [19, 20], but – to the best of our knowledge – none of the existing solutions is able to address both the issues mentioned above. To fill this gap, in this paper we propose the *Fuzzy Controller for CPU and Memory Consolidation under SLA Constraints* (*FCMS*), a dynamic resource allocation framework that is able to dynamically adjust the CPU and memory capacity allocated to the set of VMs hosting a given multi-tier application in such a way to meet its SLAs in face of (a) bursty and non-stationary workloads, whose intensity varies over time, and (b) the presence of other VMs that compete for the same set of physical resources, as well as to avoid any negative interaction between the corresponding allocation mechanisms.

The heart of our proposal is a *Multiple-Input Multiple-Output* controller based on the fuzzy logic that, in face of non-stationary workloads and resource contention, continuously adjusts the CPU and memory capacity allocation of each VM in a coordinated fashion to meet the corresponding application-level SLAs and to avoid the negative effects resulting from the interactions of dynamic resource allocation mechanisms. As a result, it is able to provide percentile-based performance guarantees for both throughput and response time.

As discussed in literature (e.g., [19, 20, 21, 22]), approaches based on fuzzy control have shown to be better to cope with non-stationary workloads than those based on linear feedback control [23, 24]. The inherent nonlinearities of computing systems [25] make indeed the design of such controllers very challenging. As a matter of fact, to the best of our knowledge existing model-based linear controllers are unsuitable to properly tackle the issues arising in the scenarios considered in this paper, because of the linearization operations they have to perform, whose side-effect is to lead the controller to make inaccurate or even wrong allocation decisions. Conversely, fuzzy controllers

DOI: 10.1002/cpe.3968

have been shown [26, 27] to be able to suitably approximate any nonlinear function, and as such to be able to properly deal with such nonlinearities.

To demonstrate the ability of FCMS of meeting its design goals, we implement it on a real testbed, and use this implementation to carry out a comprehensive experimental evaluation involving a set of real-world cloud applications and workloads. Furthermore, we compare the performance achieved by FCMS against those attained by two existing state-of-the-art alternative solutions (that we also implemented).

Our results show that FCMS, unlike its counterparts, is able to meet the SLAs of all applications and to provide the highest consolidation level by allocating the minimum CPU and memory capacity. Specifically, we show that – while existing alternatives either fail to meet application-level SLAs, or over-allocate CPU and memory capacity, or both – FCMS is always able to allocate to each application as little CPU and memory capacities as needed to meet its SLA, thus achieving a better consolidation level without violating any SLA.

Our Contributions

In this paper, we extend our previous work [21], focused on the dynamic provisioning of CPU capacity, in such a way to properly handle both CPU and memory capacity. To the best of our knowledge, this is the first work in which the allocation of both resource types is performed in a coordinated way, so that the negative effects of their interactions are avoided.

In this work, we present FCMS, a resource management framework, based on feedback fuzzy control, which significantly extends the above paper as follows:

1. we design a novel fuzzy controller able to precisely allocate memory capacity to VMs, while simultaneously avoiding to perform memory allocations when such an action would negatively impact on application performance, and we couple it with our existing CPU fuzzy controller [21];
2. we enhance the design of the fuzzy controller by means of suitable algorithms for the estimation of application performance indices that improve existing ones [21];
3. we implement FCMS, as well as two existing state-of-the-art feedback controllers, and compare FCMS against them for a larger set of real-world cloud applications;
4. we show that FCMS outperforms these existing alternatives in all the experimental scenarios we consider.

The rest of the paper is organized as follows. In Section 2, we define the model of the system that we consider as well as the context for the problem that we tackle. In Section 3, we describe the design of the FCMS framework. In Section 4, we illustrate the implementation of FCMS on a real testbed, that we use to carry out an experimental evaluation to assess its efficacy, and in Section 5, we present the results obtained from it. In Section 6, we discuss related works. Finally, in Section 7, we conclude the paper and discuss possible future works.

2. SYSTEM MODEL AND PROBLEM DEFINITION

2.1. System Model

We consider a computing infrastructure, sketched in Figure 1, consisting in a set of physical servers that are managed by a virtualization platform, through a suitable *Virtualization Management System*. The infrastructure hosts a set of multi-tier cloud applications, each one providing services to a population of clients. Each tier A_i ($i = 1, \ldots, n_A$) of an application A is deployed in a VM, which is hosted on one of the physical servers of the infrastructure. Each VM is equipped with a suitable number of *virtual CPUs* (vCPUs), and suitable amounts of RAM and disk space. Each vCPU of a VM is allocated (typically non-exclusively) on a physical CPU core, and to each VM is allocated a portion of the physical RAM. In the following, we use the terms "application tier" and "VM" interchangeably.

DOI: 10.1002/cpe.3968

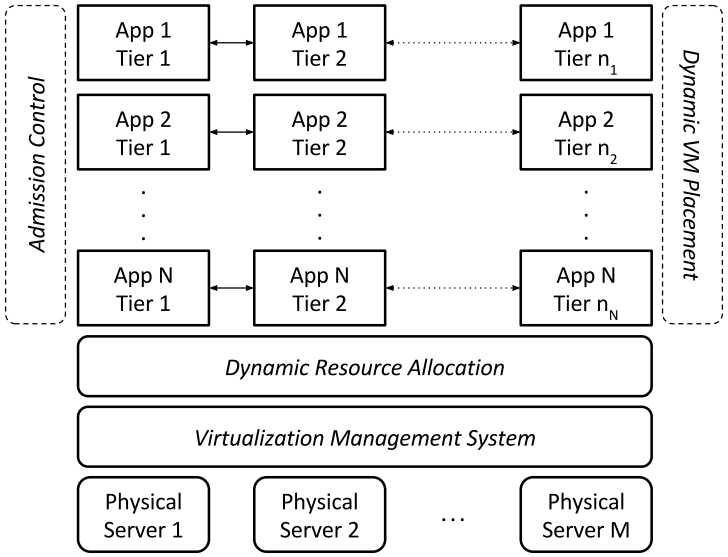


Figure 1. Architecture of the computing infrastructure. Dashed boxes represent components outside the scope of this paper.

We assume that the initial placement of VMs onto the hosts of the computing infrastructure, and the allocation of physical resources to them, is carried out by means of suitable capacity planning [28] or resource over-commitment techniques [29].

The workload of each application A consists of a stream of requests for service coming from the population of its clients according to a given distribution $\mathcal{W}_A(t)$ of arrivals at time t. We assume that the workload is *non-stationary*, i.e. that $\mathcal{W}_A(t)$ changes over time.

Each application is characterized by *percentile-based* SLOs [30], which are expressed as bounds on a suitable percentile of the distribution of the performance measure of interest. Percentile-based SLOs are indeed considered preferable for many applications than others based on simpler metrics like average [31], since they are more robust to variations over multiple time scales that are typical of cloud workloads [32, 33], but they are very challenging to meet [34].

Specifically, the SLO of an application A is expressed as a pair $(p, r)_A$, where r (the *SLO value*) is the upper bound on the p^{th} percentile of the distribution of the measured performance indicator, and states that $p\%$ of the observed values must be lower than or equal to r during a prescribed time interval. For instance, a SLO $(95, 0.1 \text{ sec})_A$ on response time states that the 95% of the observed response times of the requests to application A must be lower than or equal to 0.1 sec, in a prescribed interval.

Given an application A, the *Dynamic Resource Allocation* module (see Figure 1) is in charge of allocating to each one of its tiers A_i a suitable fraction of the capacity of the physical resources on which the corresponding VM is running in order to meet the SLO $(p, r)_A$. This module is the focus of this paper.

In particular, we focus on the management of the amounts of physical CPU capacity $C_{A_i}(t)$ and physical memory capacity $M_{A_i}(t)$ allocated to each tier A_i at time t in order to meet $(p, r)_A$. CPU and memory are indeed considered the most relevant ones, as they determine both application performance and the efficient usage of physical resources [15, 16], as demonstrated also by the commercial offerings of most CPs (e.g., Amazon Web Services [35], Google Cloud Platform [36], and Rackspace [37]), that charge their customers based on their CPU and RAM consumption only.

DOI: 10.1002/cpe.3968

We postulate that the *Virtualization Management System* provides suitable mechanisms to perform dynamic CPU (in particular, *non-work-conserving* CPU scheduling [38]) and memory management (e.g., ballooning drivers or memory hotplug [18]). †

We finally assume that the system we consider complements dynamic resource allocation (the focus of our work) with two additional mechanisms, namely *Admission Control* (to make sure that the workload intensity experienced by application A never exceeds the maximum level W_A^* agreed upon by the CP and the customer), and *Dynamic VM Placement* (that is in charge of reallocating VMs in order to ensure that each server has enough capacity to meet the SLOs of all the applications using it). To minimize the interference of VMs co-located on the same physical server [42], we assume that dynamic VM placement is carried out so as to place on each server only VMs exhibiting as little interference as possible. These mechanisms, however, are outside the scope of this paper, and we assume that existing techniques are used (e.g., see [43, 44] for admission control, and [45, 46, 47, 48] for dynamic and contention-aware VM placement).

2.2. *Problem Definition*

As discussed in the previous subsection, we focus on the problem of the joint dynamic allocation of CPU and memory capacity to the various application tiers, that can be stated as follows: given the number $N_A(t)$ of requests for application A that must be processed at time t, allocate to each tier A_i the smallest amounts of CPU and memory capacity (i.e., suitable values of $C_{A_i}(t)$ and $M_{A_i}(t)$, respectively) so that $(p, r)_A$ is met. By smallest amount of capacity, we mean that the aggregate capacity allocated to each VM over a medium to long time scale is, on average, the minimum required to meet the SLO of the corresponding application.

The need of allocating the smallest amount of resource capacity stems from the desire of simultaneously meeting application SLAs and maximizing the consolidation level. On the one hand, under-allocating CPU capacity to a VM limits its ability to process incoming requests in a timely fashion, while under-allocating memory capacity results in increases in swapping activity and may even result in the forced terminations because of the lack of memory. On the other hand, over-allocating CPU and memory capacities to a VM limits consolidation opportunities.

To solve this problem, two issues must be properly faced. First, we must be able to precisely estimate the resource needs of each application tier, in face of the non-stationarity of the workload, and allocate that capacity as time goes by. This problem has been already addressed in the literature, where various approaches exist to dynamically allocate the capacity of a single resource type (see, for instance, [21] for the CPU, and [49] for the memory).

Second, we must be able to coordinate the allocation of CPU and memory, in order to avoid the arising of the effects caused by the interactions of the mechanisms used to carry out them. In particular, as reported in the literature [17, 18], when the amount of memory allocated to a given VM is changed, the utilization of the corresponding vCPU suddenly increases for a magnitude and a duration that vary according to the magnitude of memory variation and the type of operation (i.e., memory increment or decrement), and shows a bursty behavior. ‡

If, when reallocating memory to a VM, it is using almost all the CPU capacity it has been allocated, then the memory adjustment operation will subtract a (usually) large amount of CPU capacity to the applications running in the VM until CPU capacity reallocation is not performed again. During that time, application performance will suffer, and the SLO will be missed. When CPU capacity is reallocated, the new value will be possibly set to a value that takes into account also the effects induced by memory reallocation. However, when this latter operation has been completed, the extra CPU capacity will be wasted, thus lowering the consolidation level that can be attained.

†These mechanisms are provided by the most common virtualization platforms, e.g. VMware [39], Xen [40], and KVM [41].

‡For instance, in some experiments we performed with Xen 4.4, we observed that an increment (decrement) of 9.6 GiB results in an increase of 80% (70%) of the vCPU utilization for 6 (11) sec, while for variations of 1.6 GiB this utilization amounts to about 42% for 3 sec.

DOI: 10.1002/cpe.3968

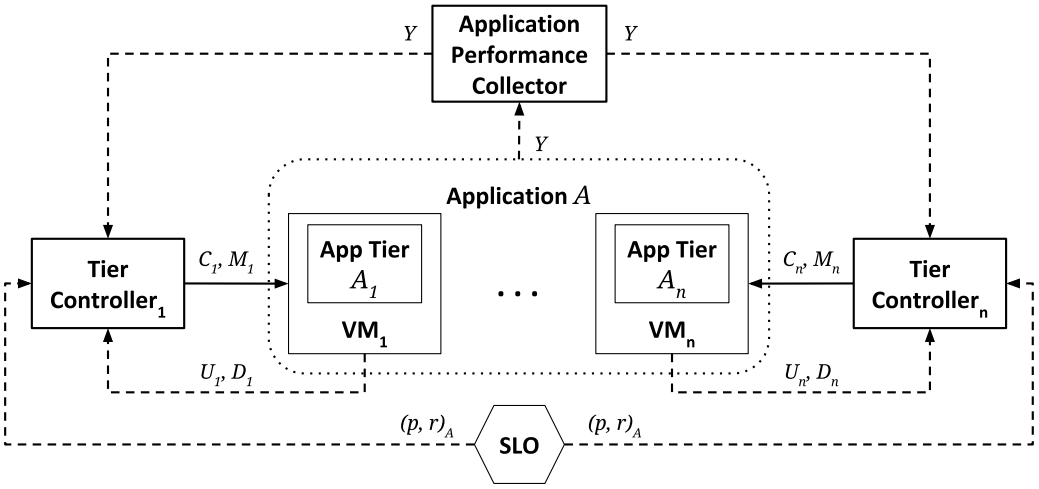


Figure 2. Architecture of FCMS.

For these reasons, the dynamic management of CPU and memory cannot be done independently from each other. In this paper, we propose FCMS, a dynamic management approach that – by coordinating CPU and memory allocations – is able to take into account their interactions and avoid the negative effects illustrated above.

3. THE FCMS FRAMEWORK

The architecture of FCMS is shown in Figure 2 where – to enhance readability – we show only the components corresponding to a specific application (i.e., the architecture shown in the figure should be replicated for each application running on the infrastructure), and we drop the identifier of the application from all the subscripts whenever it is clear from the context (i.e., we denote as i instead of A_i the entities and the quantities corresponding to tier i of application A).

As shown in Figure 2, each application tier VM_i is associated with a *Tier Controller* T_i, [§] which periodically computes the amount of vCPU capacity C_i and of memory capacity M_i to be assigned to VM_i in order to meet the application SLO $(p, r)_A$. Both C_i and M_i are real numbers taking values in the interval $[0, 1]$, that represent the overall fraction of CPU time and memory size, respectively, normalized over the corresponding maximum values.

To carry out its task, T_i is activated regularly at equispaced points in time (the interval elapsing between 2 consecutive activations is called *control interval*). During each control interval, T_i takes as input (a) the currently observed application performance Y (i.e., the one used for defining the SLO) provided by the *Application Performance Collector*, (b) the SLO specification $(p, r)_A$, and (c) the vCPU utilization U_i and the memory demand D_i of VM_i, and computes the values of C_i and M_i that will be allocated during the next control interval. To properly handle multi-tier applications, Tier Controllers that are associated to the same application are kept synchronized, i.e., they are activated at the same time, so that they make decisions with respect to the same workload conditions. The effectiveness of this solution is clearly demonstrated experimentally in Sec. 5, where we show that FCMS is able to achieve its design goals, namely to achieve a consolidation level better than other state-of-the-art controllers without violating any application SLA, for several real-world multi-tier cloud applications exposed to challenging time-varying, non-stationary, and bursty workloads.

To compute C_i and M_i, T_i uses two distinct *Multiple-Input-Single-Output* static *fuzzy controllers* [50] (i.e., feedback controllers based on fuzzy logic [51, 52]), namely the *Fuzzy CPU*

[§]FCMS can be easily extended to deal also with situations where the same tier is replicated into multiple distinct VMs for load balancing, by simply associating each VM of the same tier with a different Tier Controller.

DOI: 10.1002/cpe.3968

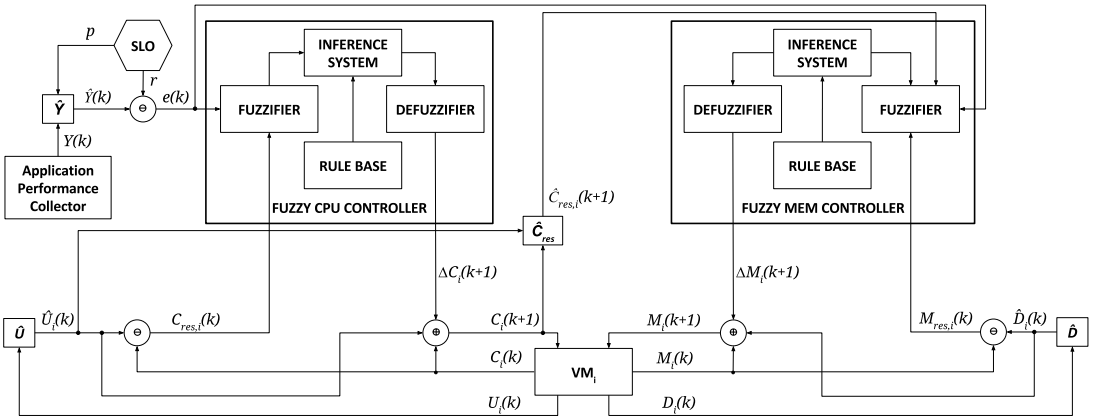


Figure 3. Architecture of the *Tier Controller*.

Controller and the *Fuzzy Mem Controller*, whose structure and interconnections are shown in Figure 3. As indicated by their respective names, these controllers are in charge of allocating CPU and memory capacity, respectively, to the corresponding tier.

In particular, the Fuzzy CPU Controller (simply *CPU Controller* in the following) computes the CPU capacity allocated to the corresponding tier during control interval $k+1$ as in Eq. (1):

$$C_i(k+1) = \max\left\{\widehat{U}_i(k), \min\left\{C_i(k)\cdot\big(1+\Delta C_i(k+1)\big), 1\right\}\right\} \tag{1}$$

where $\Delta C_i(k+1)$ represents the percentage adjustment, and the final value is bound between an estimate of the currently used capacity $\widehat{U}_i(k)$ and the maximum achievable value 1.

Similarly, the Fuzzy Mem Controller (simply *Mem Controller* in the following) computes the memory size allocated during control interval $k+1$ as in Eq. (2):

$$M_i(k+1) = \max\left\{\widehat{D}_i(k), \min\left\{M_i(k)\cdot\big(1+\Delta M_i(k+1)\big), 1\right\}\right\} \tag{2}$$

where $\Delta M_i(k+1)$ represents the percentage change and $\widehat{D}_i(k)$ is an estimate of the current memory demand. In Figure 3, these two equations are denoted by $\oplus$.

Given that $\Delta C_i(k+1)$ and $\Delta M_i(k+1)$ can take both positive and negative values, the CPU and memory capacity allocated in interval $k+1$ may be either higher or lower than those in interval k (and, of course, they may remain unchanged as well).

To avoid the negative effects originated by the lack of coordination between CPU and memory allocation discussed in Section 2, the above controllers interact according to the cascade control strategy [25], whereby the "outer" CPU Controller controls, through its output vCPU capacity, the "inner" Mem Controller, which takes that vCPU capacity as input (see the related blocks in Figure 3). In particular, the former controller provides the value of $C_i(k+1)$ as input to the latter one, that in turn can adjust $\Delta M_i(k+1)$ to be proportional to the available vCPU capacity in the same control interval. By doing so, we avoid that a too large memory variation reduces too much the amount of CPU capacity available to applications.

In the rest of this section, we describe the structure and the behavior of the fuzzy controllers composing the Tier Controller. More precisely, for conciseness, we will focus on the Mem Controller, and we will omit the discussion of the CPU Controller. As a matter of fact, in this paper we use the same CPU Controller that we developed in the past [21], whose behavior does not significantly differ from that of the Mem Controller; furthermore, the expressions for the $C_{res,i}(k)$, $\widehat{U}_i(k)$, $\Delta C_i(k+1)$ and $C_i(k+1)$ (see Figure 3) can be simply obtained by replacing the corresponding symbols in the formulas for their counterparts $M_{res,i}(k)$, $\widehat{D}_i(k)$, $\Delta M_i(k+1)$, and $M_i(k+1)$, respectively, in the Mem Controller.

DOI: 10.1002/cpe.3968

3.1. The Fuzzy Mem Controller

The purpose of the Mem Controller is to determine the amount of memory capacity to allocate to tier i in the next control interval $k+1$. As any fuzzy controller, it contains the following 4 building blocks (see Figure 3):

- the *rule base*, that stores a set of *fuzzy rules* through which control decisions are made;
- the *fuzzyfier*, that converts the real input values $e(k)$, $M_{res,i}(k)$, and $\widehat{C}_{res,i}(k)$ into equivalent fuzzy values;
- the *inference system*, that decides which rules can be applied to the current system state on the basis of the values computed by the fuzzyfier, and determines a fuzzy output;
- the *defuzzifiers*, that combines the fuzzy output into a single real value $\Delta M_i(k+1)$.

3.1.1. Controller inputs and outputs As shown in Figure 3, the Mem Controller uses three inputs, namely the *relative error* $e(k)$, the *relative residual memory capacity* $M_{res,i}(k)$, and the *relative estimated residual vCPU capacity* $\widehat{C}_{res,i}(k)$, to compute its single output $\Delta M_i(k+1)$.

The relative error $e(k)$ represents the normalized difference between the desired value r and the incremental estimate of the achieved one $\widehat{Y}(k)$, and is computed as in Eq. (3) (in Figure 3 this operation is denoted by $\ominus$), where we distinguish response time-based from throughput-based SLOs:

$$e(k) = \begin{cases} \frac{r-\widehat{Y}(k)}{r}, & \text{for response time SLOs,} \\ \frac{\widehat{Y}(k)-r}{r}, & \text{for throughput SLOs.} \end{cases} \tag{3}$$

where $\widehat{Y}(k)$ is computed by means of the *t-digest algorithm* [53, 54]. The error $e(k)$ is used to determine whether the SLO is actually respected ($e(k) \geq 0$) or not ($e(k) < 0$).

The relative residual memory capacity $M_{res,i}(k)$ represents the normalized difference between $M_i(k)$ and an estimate of the actual memory demand $\widehat{D}_i(k)$ of the VM, and is computed as shown in Eq. (4):

$$M_{res,i}(k) = \frac{M_i(k) - \widehat{D}_i(k)}{M_i(k)} \tag{4}$$

where $\widehat{D_i}(k)$ is a smoothed value that suitably combines past observations of memory demand D_i by means of the *Exponentially Weighted Moving Average* (EWMA), that is:

$$\widehat{D}_i(k) = \beta \cdot D_i(k) + (1-\beta) \cdot \widehat{D}_i(k-1) \tag{5}$$

where β is the *smoothing factor* (a real number taking values in the $[0,1]$ interval).

Finally, the relative estimated vCPU capacity $\widehat{C}_{res,i}(k)$ represents the normalized difference between the new vCPU allocation $C_i(k+1)$, provided by the CPU Controller, and the estimate of the vCPU utilization $\widehat{U}_i(k)$, and is computed as in Eq. (6):

$$\widehat{C}_{res,i}(k) = \frac{C_i(k+1) - \widehat{U}_i(k)}{C_i(k+1)} \tag{6}$$

3.1.2. The rule base The actual fuzzy logic is implemented as a set of *if-then* rules, stored in the *rule base*, that translate human expert's control knowledge into a form that can be used by the inference system.

Fuzzy rules are defined by means of *linguistic variables* that take *linguistic values* (or *terms*), and represent the control inputs and outputs. In particular, each rule defines the conditions under which it can be applied (the "if" part, or *antecedent*), and the output deriving from its application (the "then" part, or *consequent*).

The design of the rule base takes into account (a) the objective of the Mem Controller, namely to meet the SLO (i.e., to keep $e(k) \geq 0$) and, at the same time, to minimize the allocated memory capacity (i.e., to keep $M_{res,i}(k) \cong 0$), and (b) the effects of dynamic memory adjustments on the vCPU utilization (as discussed in Section 2).

DOI: 10.1002/cpe.3968

(a) Case: "$\widehat{C}_{res}$" is *LOW*.

		"e"		
		NEG	*FINE*	*POS*
"M_{res}"	*LOW*	STY	STY	STY
	FINE	STY	STY	STY
	HIGH	STY	STY	STY

(b) Case: "$\widehat{C}_{res}$" is *FINE*.

		"e"		
		NEG	*FINE*	*POS*
"M_{res}"	*LOW*	UP	STY	UP
	FINE	UP	STY	DWN
	HIGH	STY	STY	DWN

(c) Case: "$\widehat{C}_{res}$" is *HIGH*.

		"e"		
		NEG	*FINE*	*POS*
"M_{res}"	*LOW*	BUP	UP	UP
	FINE	UP	STY	DWN
	HIGH	STY	DWN	BDW

Table I. The rule base.

To define the rule base, we must first define the linguistic variables and values corresponding to the numerical inputs and output of the controller, as follows:

- "e" (the linguistic counterpart of the $e(k)$ control input), that can take *NEG*, *FINE*, or *POS* as linguistic values to denote negative, zero, and positive values of $e(k)$, respectively;
- "M_{res}" (the linguistic counterpart of the $M_{res,i}(k)$ control input), that can take *LOW*, *FINE*, or *HIGH* as linguistic values to denote values close to 0, equal to 0, and larger than 0, respectively;
- "$\widehat{C}_{res}$" (the linguistic counterpart of the $\widehat{C}_{res,i}(k)$ control input), that takes the same linguistic terms as the "M_{res}" linguistic variable;
- "ΔM" (the linguistic counterpart of the $\Delta M_i(k+1)$ control output), that can take *BUP*, *UP*, *STY*, *DWN*, or *BDW* as linguistic terms, whose meaning is defined as follows: large increment (*BUP*), small increment (*UP*), stationary (*STY*), small decrease (*DWN*), and large decrease (*BDW*).

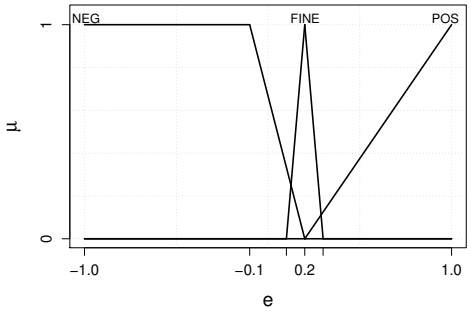
Intuitively, the various rules in the rule base determine how the controller decides if memory allocation should be changed, and the amount of such variation, when the residual CPU capacity (a) is being used completely (i.e., $\widehat{C}_{res,i}(k) = 0$), or (b) is almost depleted (i.e., $\widehat{C}_{res,i}(k) \cong 0$), or (c) is available in a large amount (i.e., $\widehat{C}_{res,i}(k) > 0$).

In each one of the above scenarios, the magnitude of memory variation is determined according to the values of $e(k)$ and $M_{res,i}(k)$. Intuitively, if the SLO is not being met (i.e., $e(k) < 0$), the amount of allocated memory is increased, while in the opposite case the controller tends to leave memory allocation unchanged if the SLO value is being tracked very closely (i.e., $e(k) = 0$), and to decrease it if the SLO is abundantly exceeded (i.e., $e(k) > 0$).

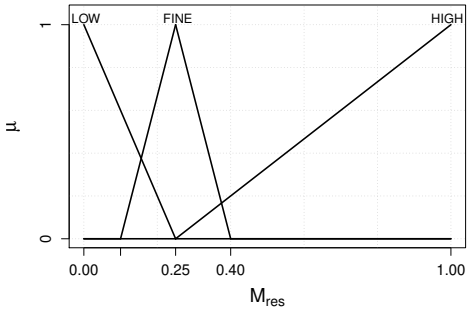
The resulting rule base is expressed in a compact form as a set of tables (see Table I), one for each one of the scenarios mentioned above. In each table, the cell (B, C) contains the term taken by "ΔM" when "$\widehat{C}_{res}$" is A, "M_{res}" is B, and "e" is C (we simply denote this as $\langle A, (B, C)\rangle$).

To exemplify, let us consider the case $\langle HIGH, (LOW, NEG)\rangle$ in Table Ic, that corresponds to the rule *if "$\widehat{C}_{res}$" is HIGH and "M_{res}" is LOW and "e" is NEG then "ΔM" is BUP*. This rule encodes the control knowledge stating that if the CPU capacity value used by the VM is smaller than that allocated by the CPU Controller (encoded by the *"$\widehat{C}_{res}$" is HIGH* condition), and the memory capacity value demanded by VM is close to that allocated by the Mem Controller (encoded by the *"M_{res}" is LOW* condition), and the interested percentile of the observed SLO performance metric is close to or worse than the reference one (encoded by the *"e" is NEG* condition), then the

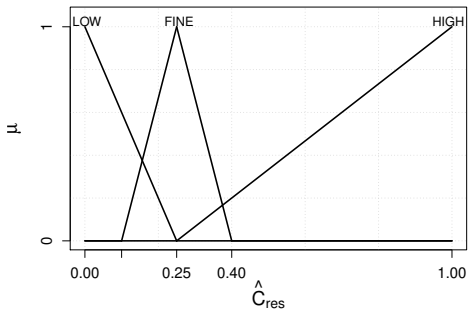
DOI: 10.1002/cpe.3968



(a) MFs for the "e" input variable.



(b) MFs for the "M_{res}" input variable.



(c) MFs for the "$\widehat{C}_{res}$" input variable.

Figure 4. MFs for the input variables "e", "M_{res}", and "$\widehat{C}_{res}$".

Mem Controller has to significantly increase the allocated memory capacity value (encoded by the *"ΔM" is BUP* proposition).

It is worth to point out that the rule base does not take into account the workload characteristics. This agnosticism is a purposely-designed feature of our approach. Indeed, any change in the workload behavior is reflected in the values taken by the performance measures of interest (e.g., a steep increase of the arrival rate induces smaller values of $M_{res,i}(k)$ and negative values of $e(k)$). Therefore, given that the rule base (and the membership functions, see below) has been conceived in such a way to deal only with changes on the performance measures of interest as induced by the workload, our controller is able to closely follow any change in the workload, provided that it does not saturate the maximum capacity of the physical resource.

3.1.3. The fuzzifier The *fuzzifier* converts each real input value (either $e(k)$, $M_{res,i}(k)$ or $\widehat{C}_{res,i}(k)$) into the equivalent linguistic variable (either "e", "M_{res}" or "$\widehat{C}_{res}$", respectively), and assigns it one or more linguistic terms. In fuzzy logic, indeed, a single real value may correspond to different linguistic terms, each one characterized by a (possibly different) degree of *certainty*. The fuzzifier, therefore, given a numeric input value x and the corresponding linguistic variable "x", computes for each possible linguistic term T of "x" the membership $\mu \in [0, 1]$, representing the grade of certainty that x belongs to T. This is accomplished by using a *Membership Function* (MF) for each (numeric) input variable x that quantifies, for each possible linguistic term T that the corresponding linguistic variable "x" may take, the grade of certainty μ of x in T, and we denote it as μ_T("x"). In our controller, we use the *triangular-shaped* and *ramp-shaped* MFs, which are the most commonly used MFs in practice, that are shown in Figure 4.

As shown in this figure, for each input variable, there is one MF for each linguistic term. Specifically, for the "e" linguistic variable, there are two ramp-shaped MFs corresponding to the linguistic terms *NEG* and *POS*, and one triangular-shaped MF for the linguistic term *FINE*. For the "M_{res}" linguistic variable, there are two ramp-shaped MFs for the linguistic terms *LOW* and *HIGH*, and one triangular-shaped MF for the linguistic term *FINE*. Finally, for the "$\widehat{C}_{res}$"

Prepared using cpeauth.cls
DOI: 10.1002/cpe.3968

linguistic variable, there are two ramp-shaped MFs for the linguistic terms *LOW* and *HIGH*, and one triangular-shaped MF for the linguistic term *FINE*.

To exemplify, if $e(k) = 0.2$, the MF in Figure 4a associates to this input the value *FINE* with certainty 1.0, since in the MF the value 0.2 projects up to the peak of the MF corresponding to the linguistic term *FINE*. This corresponds to state that the linguistic variable "e" takes the value *FINE* with certainty 1.0, i.e., that $\mu_{FINE}(\text{“}e\text{”}) = 1.0$.

Note also that, if $e(k) = 0.15$, then "e" takes *two* linguistic terms (namely, *NEG* and *FINE*), since its projection up intersects the MFs of both these terms, each one characterized by its own certainty value greater than 0.

3.1.4. The inference system The *inference system* determines the *active rules* (i.e., the rules that will be applied to compute the fuzzy output) by checking for each rule whether at least one of the linguistic expressions in its *antecedent* (i.e., the "if" part) has a membership value greater than 0, and, to do so, it applies the *min* or the *max* operator depending on whether the terms in the antecedent are joined by a conjunction (i.e., "and") or by a disjunction (i.e., "or"), respectively.

To exemplify, assume that $\mu_{HIGH}(\text{“}\widehat{C}_{res}\text{”}) = 1.0$, $\mu_{HIGH}(\text{“}M_{res}\text{”}) = 1.0$ and $\mu_{NEG}(\text{“}e\text{”}) = 0.6$ (and that the certainties of all the other linguistic values are 0). By looking at Table Ic, we see that only the rule corresponding to $\langle HIGH, (HIGH, NEG)\rangle$ applies, thus resulting in a consequent with a certainty of 0 for all the linguistic values but *STY*, whose certainty is given by:

$$\mu_{STY}(\text{“}\Delta M\text{”}) = \min\big(\mu_{HIGH}(\text{“}\widehat{C}_{res}\text{”}), \mu_{HIGH}(\text{“}M_{res}\text{”}), \mu_{NEG}(\text{“}e\text{”})\big) = \min(1.0, 1.0, 0.6) = 0.6$$

Conversely, if $\mu_{HIGH}(\text{“}\widehat{C}_{res}\text{”}) = 1.0$, $\mu_{LOW}(\text{“}M_{res}\text{”}) = 0.24$, $\mu_{FINE}(\text{“}M_{res}\text{”}) = 0.6$, and $\mu_{FINE}(\text{“}e\text{”}) = 1.0$ (while the certainties of all the other linguistic values are 0), then *two* rules apply, namely $\langle HIGH, (LOW, FINE)\rangle$ and $\langle HIGH, (FINE, FINE)\rangle$, each one with a different certainty degree. According to the rule base (see Table Ic), the consequent of $\langle HIGH, (LOW, FINE)\rangle$ is *UP*, while the one of $\langle HIGH, (FINE, FINE)\rangle$ is *STY*. The certainty of these two consequents, namely $\mu_{UP}(\text{“}\Delta M\text{”})$ and $\mu_{STY}(\text{“}\Delta M\text{”})$, is computed as:

$$\mu_{UP}(\text{“}\Delta M\text{”}) = \min\big(\mu_{HIGH}(\text{“}\widehat{C}_{res}\text{”}), \mu_{LOW}(\text{“}M_{res}\text{”}), \mu_{FINE}(\text{“}e\text{”})\big) = \min(1.0, 0.24, 1.0) = 0.24,$$
$$\mu_{STY}(\text{“}\Delta M\text{”}) = \min\big(\mu_{HIGH}(\text{“}\widehat{C}_{res}\text{”}), \mu_{FINE}(\text{“}M_{res}\text{”}), \mu_{FINE}(\text{“}e\text{”})\big) = \min(1.0, 0.6, 1.0) = 0.6.$$

3.1.5. The defuzzifier Finally, the *defuzzifier* combines the *conclusions* (i.e., the "then" part) of the active rules, identified by the inference system, by means of the *centroid method* [52] to obtain a single numeric value, representing the value for the $\Delta M_i(k+1)$ output variable.

In the centroid method, the numeric value of the control output is computed as a weighted average of the certainty of the conclusions of the active rules, where the weight of each conclusion is the center point of the MF associated to the output linguistic term (indeed, this is equivalent to compute the center of the area under the curve resulting by joining the MFs involved in the active rules). To do so, we quantify the linguistic values associated to the output variable "ΔM" by means of the triangular-shaped MFs shown in Figure 5.

For instance, if $\mu_{UP}(\text{“}\Delta M\text{”}) = 0.24$ and $\mu_{STY}(\text{“}\Delta M\text{”}) = 0.6$, the area under these MFs is the gray one shown in Figure 6. As expected, the gray area for the *STY* MF is greater than the gray area for the *UP* MF. This is due to the fact that the degree of certainty of *STY* is higher than the one of *UP*. Figure 6 also shows the centroid of the gray area (i.e., the black-filled circle at $(0.029, 0.177)$). The x-coordinate of the centroid is the output of the defuzzification process which in our case corresponds to the value for $\Delta M_i(k+1)$. In particular, in our example, $\Delta M_i(k+1) = 0.029$ which means that the Mem Controller has to increase the allocated memory capacity value by 2.9% of the current capacity.

DOI: 10.1002/cpe.3968

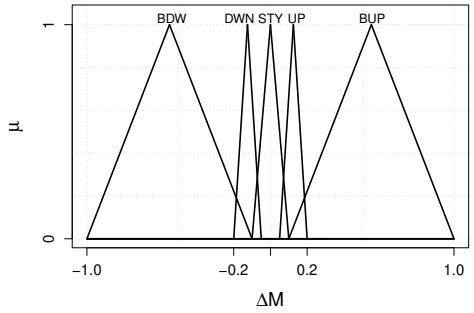


Figure 5. MFs for the output variable "ΔM".

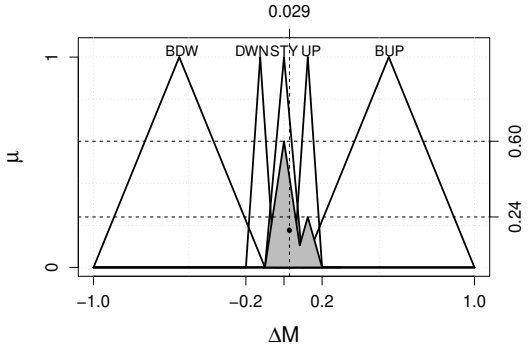


Figure 6. Defuzzification of the aggregate output with the centroid method. The black-filled circle at $(0.029, 0.177)$ is the centroid of the gray area.

4. SYSTEM IMPLEMENTATION

To assess the capability of FCMS to achieve its design goals, we implemented its various modules in C++, and integrated them into a testbed that provides various software components that are needed to run them. To foster further research, and to provide reproducibility, all these modules are publicly available on [55].

The architecture of the resulting system is shown in Figure 7. As shown in the figure, the testbed includes a virtualized computing infrastructure, running a set of cloud applications $APP_1, \ldots, APP_k$, exposed to a suitably chosen workload (generated by the *Workload Driver*). The *Application Manager* implements suitable mechanisms that can be leveraged by a controller to allocate resource capacity to the various applications on the basis of the inputs provided by the *VM Utilization Collector* and the *Application Performance Collector*. FCMS has been implemented as a module sitting on top of the Application Manager.

Virtualization support is provided by means of *Xen* [40] (version 4.4), whose *credit scheduler* and *balloon driver* are used to provide non-work-conserving scheduling and dynamic memory management, respectively, as follows:

- non-work-conserving scheduling: under the credit scheduler, each VM i is associated a cap Γ_i, an integer number taking values in the interval $[0, 100 \cdot m_i]$ (where m_i is the number of physical CPU cores allocated to VM i), representing the maximum percentage of CPU cycles that VM i is entitled to consume (even if the host has idle CPU cycles). Non-work-conserving scheduling is implemented by setting, at each control interval k, $\Gamma_i = \lceil C_i(k) \cdot 100 \cdot m_i \rceil$ (recall that $C_i(k) \in [0, 1]$);
- dynamic memory management: under the memory ballooning technique, each VM i is characterized by the maximum memory $\Psi_{\text{max},i}$ it can use, and by the current memory Ψ_i is using. Thus, the balloon driver hands back to Xen the memory given by $(\Psi_{\text{max},i} - \Psi_i)$. To implement dynamic memory allocation with the memory ballooning technique, at each

DOI: 10.1002/cpe.3968

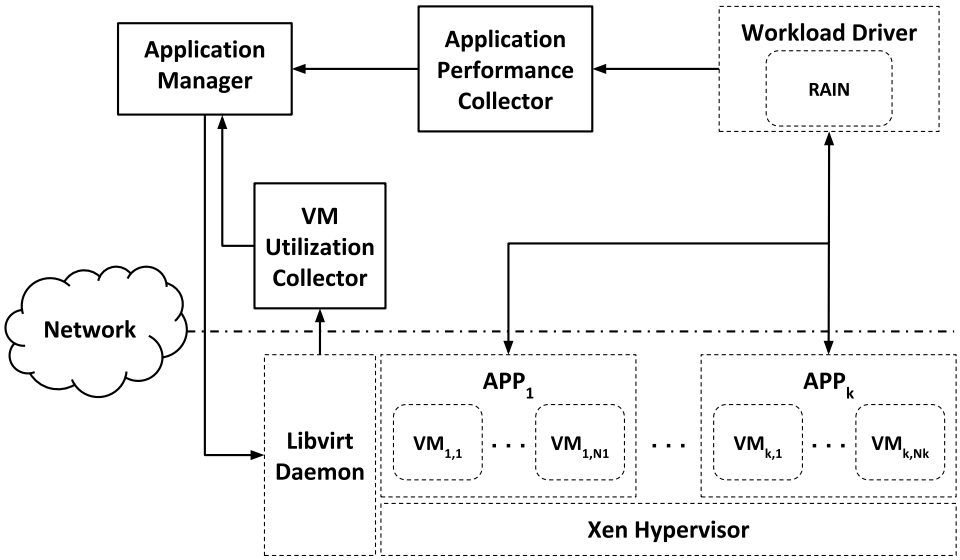


Figure 7. Architectural diagram of our testbed.

control interval k, we adjust the current memory in use by VM i to the value $\Psi_i = \lceil M_i(k) \cdot \Psi_{\max,i} \rceil$ (recall that $M_i(k) \in [0, 1]$).

Xen is coupled with the *libvirt* daemon [56] to enable the Application Manager to interact with Xen to set the CPU and memory capacity of the various VMs as described above. This daemon is also used by the VM Utilization Collector to harvest the CPU utilization values for the various VMs, while memory utilization is collected via the */proc/meminfo* special file of the Linux operating system (running inside each VM).

5. EXPERIMENTAL RESULTS

To assess the ability of FCMS to allocate as little CPU and memory capacity as needed to meet applications SLOs, we carry out a thorough experimental evaluation by means of the testbed described in Section 4, on which we run several cloud applications, each one processing a suitably chosen workload, whose CPU and memory capacity is allocated by our controller.

For the experiments, we use 2 Fujitsu Server PRIMERGY RX300 S7, connected via a Gigabit Ethernet switch, each one equipped with two 2.4 GHz Intel Xeon E5-2665 processors with 8 cores each, and with 96 GiB of RAM, both running the Linux kernel version 4.1.13 and Xen 4.4. One of these machines is used to run all the VMs corresponding to the cloud applications used in our evaluation, while the other one is used to run all the software modules composing the testbed.

To compare FCMS against state-of-the-art alternative solutions, we implement two additional state-of-the-art controllers, namely *APPLEware* [19] (which uses a model predictive controller and a neuro-fuzzy inference system) and *FMPC* [20] (which uses a predictive controller, based on genetic algorithms, and a neuro-fuzzy inference system), that have been chosen thanks to their ability to allocate both CPU and memory capacity, and carry out the same experiments also with them.

The results we obtained in the experiments demonstrate the ability of FCMS to achieve its design goals, and to outperform its alternative counterparts in all the scenarios considered in our evaluation.

In this section, after describing the performance metrics (Section 5.1), and the experimental settings (Section 5.2 and Section 5.3), we discuss the results of our experiments (Section 5.4).

DOI: 10.1002/cpe.3968

Cloud Apllication	Web or Single Tier VM			DB Tier VM		
	vCPUs	RAM (GiB)	Disk (GiB)	vCPUs	RAM (GiB)	Disk (GiB)
Cassandra	1	4	30	–	–	–
Olio	1	3	30	1	3	30
Redis	1	16	30	–	–	–
RUBBoS	1	3	30	1	3	30
RUBiS	1	3	30	1	3	30

Table II. Resource capacities allocated to each VM of the cloud applications. Cassandra and Redis are single-tier applications.

5.1. Performance Metrics

To quantify the performance attained by FCMS, as well as by the other controllers we consider for comparison purposes, we use the following three metrics:

- the *Mean CPU Capacity* (MCC) assigned to each application tier, defined as the average of the CPU capacity allocated in each control interval;
- the *Mean Memory Capacity* (MMC) allocated to each application tier, defined as the average of the memory capacity allocated in each control interval;
- the *Percent Error* $\widehat{E}$, defined as the relative percentage change between the achieved value $\widehat{Y}$ and the SLO value r, that is:

$$\widehat{E} = 100 \cdot \frac{\widehat{Y} - r}{r} \tag{7}$$

These metrics are used to rank controllers as follows: if $\widehat{E} > 0$, then the controller is violating the SLO, so it is placed at the bottom of the ranking. If, instead, $\widehat{E} \leq 0$, then the lower MCC and MMC values, the better the controller (i.e., among two controllers that both meet the SLO, the best one is that which uses less CPU and memory capacity).

To take into account variability, we run each experiment 3 times, and we compute each metric as an average of the results collected in each individual run.

5.2. Cloud Applications and Workloads

For our evaluation, we consider 5 applications that are considered to be representative of those that run on today's virtualized infrastructures [57], namely:

- *Cassandra* [58], a Java-based NoSQL data serving application (originally developed by Facebook) designed to handle large amounts of data across many commodity servers.
- *Olio* [59, 60], a two-tier PHP-based Web 2.0 Internet application for social events modeled after the Yahoo! Upcoming service.
- *Redis* [61], an in-memory data structure store, used as NoSQL key-value database, cache and message broker.
- *RUBBoS* [62], a two-tier PHP-based Web 1.0 Internet application that implements a bulletin board system modeled after an online news forum like Slashdot.org.
- *RUBiS* [63, 64], a two-tier PHP-based Web 1.0 Internet application that implements an auction site prototype modeled after eBay.com.

We deploy and run each application tier inside a separate VM, whose settings are reported on Table II (note that both Cassandra and Redis are single-tier applications).

To drive the workload of the above applications, we rely on the *RAIN* toolkit [65], a widely used workload generator for cloud applications that we extended whenever needed to make it suitable for our purposes (these extensions are now part of its official repository).

Each workload features a number of users, each one alternating its activity between *thinking* and *waiting for a response* to the request issued to the cloud application after stop thinking. For all applications, we use a negative exponentially distributed think-time whose mean is set to 7 sec.

DOI: 10.1002/cpe.3968

Cloud Application	Mix of Operations	Load (Number of Users) Phase 1	Phase 2	Phase 3
Cassandra	Zipfian distribution of key values, 40% reads, 40% writes, 20% deletes	400	800	200
Olio	*default* RAIN *Olio mix*	100	150	50
Redis	Zipfian distribution of key values, 40% reads, 40% writes, 20% deletes	500	1000	250
RUBBoS	*default* RAIN *RUBBoS mix*	150	200	100
RUBiS	*default* RAIN *RUBiS mix*	30	45	15

Table III. Three-phase workload parameters.

Cloud Application	r (sec)
Cassandra	0.068
Olio	0.366
Redis	0.034
RUBBoS	0.167
RUBiS	0.915

Table IV. Application SLOs. The SLO performance metric is the response time.

Different types of operations may be requested to each cloud application, as indicated in Table III, where we specify the proportions of each request mix for the various applications.

To reproduce realistic operational conditions, characterized by time-varying and bursty workloads, the number of users is not kept constant for the entire duration of the experiments, but it is instead varied in a step-wise fashion. In particular, for each cloud application we create a step-like workload pattern with 3 phases (1 phase for each burst), each one characterized by a specific intensity, as shown in Table III, and whose duration is set for all applications to 750 sec for the first and third phases and to 300 sec for the second phase.

As shown in Table III, each workload starts with a medium intensity phase (*Phase 1*), then continues with a higher intensity phase (*Phase 2*), and then it ends with a lower intensity phase (*Phase 3*). In every experiment, each step-like workload pattern is submitted twice sequentially, thus making each experiment 1 hour long.

The SLO (p, r) for each cloud application, shown in Table IV, has been set in such a way to make it, at the same time, achievable with the physical resources available in our testbed and a nontrivial target to meet. In particular, we computed the empirical distribution of the response times of each application when running it in isolation – under the maximal intensity workload (i.e., under *Phase 2*) – after allocating each one of its vCPUs on a distinct physical CPU core. We then computed the value r corresponding to the 95^{th} percentile, and we set the SLO to $(95, r)$. In this way, the chosen SLO value is not so small that could not be achieved by any controller even by using the whole resource capacity, and not so large to be trivial to achieve even when resource capacity is low.

5.3. Controllers Parameters and Setup

All the controllers considered in our evaluation require to set the value of the *sampling time* t_s (representing the distance in time between 2 consecutive measurements of the SLO performance metric of interest), and of the *control time* t_c (representing the distance in time between 2 consecutive controller activations). In our experiments, we set $t_s = 2$ sec and $t_c = 10$ sec for all the controllers. Also, to ensure that (in each experiment) each controller starts with the same initial VM resource allocations, we set the CPU and memory shares of each VM to 100%.

The values of the remaining parameters are reported in Table V, where there is a distinct table for each controller. For the alternative solutions to FCMS, we use, whenever possible, the parameter values found in their respective papers; in cases where the respective paper did not define them, we set them to convenient values or to the best values we obtain by means of a trial-and-error approach.

DOI: 10.1002/cpe.3968

(a) Parameters for FCMS (see Section 3 for details).

Parameter	Value
Smoothing Factor β	0.9

(b) Parameters for APPLEware (see [19] for details).

Parameter	Value
Control Horizon H_c	5
Control Penalty R	1
Output Order ϱ	1
Prediction Horizon H_p	20
RLS Forgetting Factor γ	0.9
Subtractive Clustering Radius for Cassandra	0.35
Subtractive Clustering Radius for Olio	0.30
Subtractive Clustering Radius for Redis	0.35
Subtractive Clustering Radius for RUBBoS	0.35
Subtractive Clustering Radius for RUBiS	0.30
Tracking Error Weight P	1

(c) Parameters for FMPC (see [20] for details).

Parameter	Value
Control Horizon M	1
Control Penalty R	1
Genetic Algorithm Parameters	MATLAB's default [66]
Input Order m	1
Prediction Horizon P	1
Output Order n	1
RLS Forgetting Factor	0.9
Subtractive Clustering Radius for Cassandra	0.40
Subtractive Clustering Radius for Olio	0.30
Subtractive Clustering Radius for Redis	0.35
Subtractive Clustering Radius for RUBBoS	0.50
Subtractive Clustering Radius for RUBiS	0.30
Tracking Error Weight Q	1

Table V. Controllers parameters.

In addition to the above parameters, both APPLEware and FMPC need, for each cloud application, an offline step to build their initial neuro-fuzzy model from previously collected data. In their respective papers, there are not enough details to reproduce this step, so that we use a trial-and-error approach where, for each cloud application and for each controller, we use standard system identification techniques [67, 68] to select the best neuro-fuzzy model.

5.4. Results

Let us now discuss the results collected in our experiments. We run experiments in which a pair of distinct applications is executed on the same cores of the physical CPU, so that resource contention arises. More specifically, we consider all the 10 distinct pairs that can be obtained by combining the 5 cloud applications. To reduce space, in this paper we report the results for 3 of these pairs namely, $\langle RUBiS, Olio\rangle$, $\langle RUBBoS, Olio\rangle$, and $\langle Cassandra, Redis\rangle$ (however, the results for the remaining pairs do not differ significantly from these ones).

To ensure that competing VMs are executed on the same physical CPU cores, we use vCPU pinning as follows. For the $\langle RUBiS, Olio\rangle$ and $\langle RUBBoS, Olio\rangle$ pairs, we pin the VMs running the same application tier (either *Web* or *DB*) to the same physical CPU core (e.g., the 2 VMs running the Web tiers were assigned to the physical CPU core 0). Conversely, for the $\langle Cassandra, Redis\rangle$ pair, we pin the single VM of each application to the same physical CPU core. The remaining physical CPU cores are allocated to Xen's Domain-0.

The results corresponding to the $\langle RUBiS, Olio\rangle$, to the $\langle RUBBoS, Olio\rangle$, and to the $\langle Cassandra, Redis\rangle$ pairs are shown in Table VI, Table VII, and Table VIII, respectively. The rows of each table correspond to controllers, and columns report the values of $\widehat{E}$ (together with a textual

DOI: 10.1002/cpe.3968

(a) RUBiS Application.

	SLO		*MCC*		*MMC*	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)	Web (%)	DB (%)
FCMS	**Yes**	$\mathbf{-16.35}$	**37.29**	**72.81**	**35.35**	**34.72**
APPLEware	No	1513.43	6.24	16.88	23.51	18.39
FMPC	No	1506.53	13.86	37.02	56.28	54.46

(b) Olio Application.

	SLO		*MCC*		*MMC*	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)	Web (%)	DB (%)
FCMS	**Yes**	$\mathbf{-12.15}$	**42.18**	**22.81**	**76.66**	**89.71**
APPLEware	No	920.90	17.38	5.75	34.63	65.27
FMPC	No	547.64	22.56	13.63	60.89	72.12

Table VI. Results for $\langle RUBiS, Olio \rangle$.

(a) RUBBoS Application.

	SLO		*MCC*		*MMC*	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)	Web (%)	DB (%)
FCMS	**Yes**	$\mathbf{-3.90}$	**45.84**	**54.52**	**86.05**	**37.68**
APPLEware	No	2183.36	7.91	7.24	67.79	19.60
FMPC	No	3079.03	18.99	25.25	63.58	45.35

(b) Olio Application.

	SLO		*MCC*		*MMC*	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)	Web (%)	DB (%)
FCMS	**Yes**	$\mathbf{-11.95}$	**33.10**	**16.01**	**73.32**	**89.08**
APPLEware	No	1053.62	22.85	7.04	54.61	73.15
FMPC	No	526.77	22.23	14.28	53.32	68.49

Table VII. Results for $\langle RUBBoS, Olio \rangle$.

annotation indicating whether the SLO has been satisfied or not) and those of both MCC and MMC metrics attained by every controller for each application.

From these results, we conclude that FCMS is the best approach, since it is the only one able to simultaneously meet the SLO of every application (i.e., $\widehat{E}$ is negative) and provide the best consolidation level. Conversely, none of the alternative solutions is able to do the same, since each one of them is unable to meet the SLO for both applications.

The poor performance of APPLEware and FMPC is mainly due to their inability to quickly adapt to time-varying workloads. This primarily depends by the low accuracy and generalization ability of the neuro-fuzzy model built offline (see Section 5.3), that both approaches use to model the controlled applications and thus to take their control decisions. Indeed, a too accurate model leads to poor prediction accuracy when the input space is not completely covered by the fuzzy rules, and a too generic model leads to poor prediction accuracy as well, since it is too coarse. This, in turn, affects the online adaptation algorithms of the two approaches. Specifically, in the former case, the model cannot be effectively trained online with inputs that it is not able to cover, whereas in the latter case, the speed of adaptation is low. In both cases, the model adaptation is not sufficiently

DOI: 10.1002/cpe.3968

(a) Cassandra Application.

	SLO Satisfied?	$\widehat{E}$	MCC Single Tier (%)	MMC Single Tier (%)
FCMS	**Yes**	$\mathbf{-11.82}$	$\mathbf{29.28}$	$\mathbf{63.09}$
APPLEware	No	9126.67	6.69	26.60
FMPC	No	4783.15	12.61	42.17

(b) Redis Application.

	SLO Satisfied?	$\widehat{E}$	MCC Single Tier (%)	MMC Single Tier (%)
FCMS	**Yes**	$\mathbf{-14.58}$	$\mathbf{33.61}$	$\mathbf{57.80}$
APPLEware	No	3785.25	6.08	26.28
FMPC	No	1755.66	6.60	90.53

Table VIII. Results for $\langle Cassandra, Redis \rangle$.

fast to keep the pace of the changes of the incoming workload. In addition to this issue, we also note that another determining factor for the poor performance of APPLEware is the linearization step used for building the linear state-space model for the model predictive controller, which may significantly deteriorate the performance of the controller in case of strong nonlinearities in the controlled application. This issue does not affect FMPC (which indeed exhibits better performance than APPLEware) since the linearization step is not needed because the nonlinear optimization control problem is solved by means of genetic algorithms.

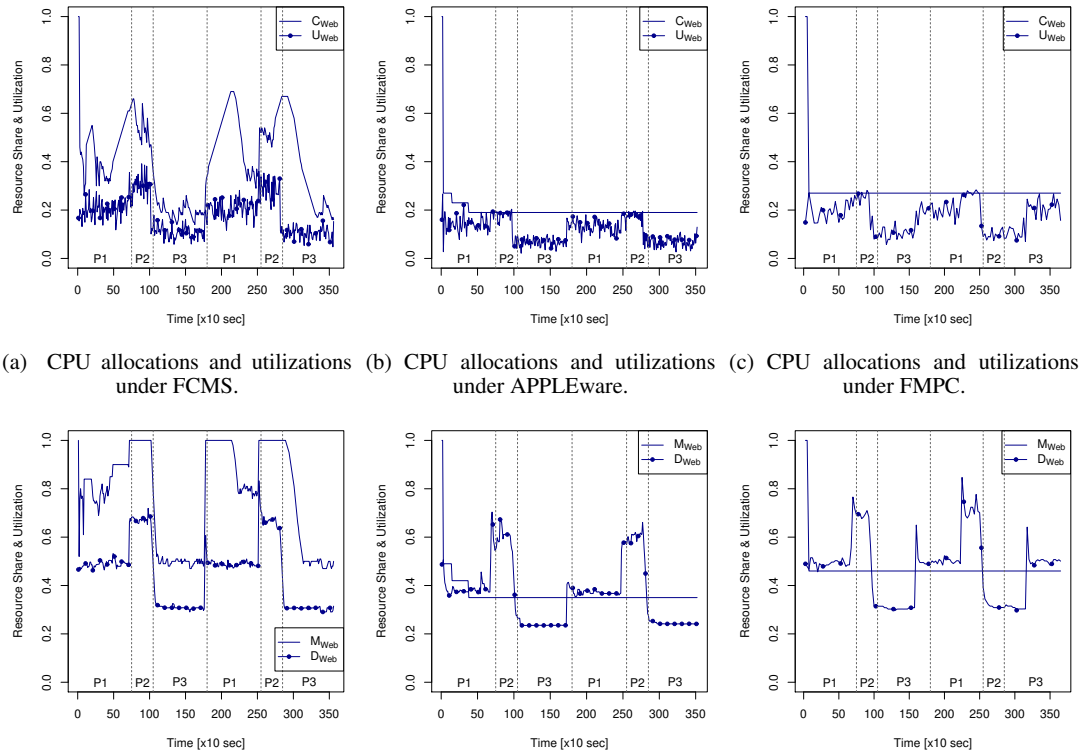
Unlike the other controllers, FCMS is the only one that is able to meet the SLO of every application (i.e., $\widehat{E}$ is negative) and to provide the highest consolidation level by allocating the minimum resource capacity (i.e., it obtains the smallest values for MCC and MMC with respect to its competitors). It is able to do so thanks to its ability to (a) take control decisions by considering, at each control interval, the available capacity of each managed resource, in addition to the performance error, and (b) take memory allocation decisions by taking into account the currently available CPU capacity.

To explain the above results, in Figures 8, 9 and 10 we plot the CPU and memory shares allocated to every tier by each of the 3 controllers, as well as the actual CPU and memory utilizations of that tier, and the percent error $\widehat{E}$ as a function of time, respectively, for the Olio application in the $\langle RUBiS, Olio \rangle$ pair (the results for the other pairs, do not significantly differ from those reported here). In the figure, to better explain the evolution of the CPU and memory utilizations in each tier and of the percent error, we highlight the regions corresponding to the different workload phases, where a vertical dashed line marks the boundary between each phase and the next one (i.e., when the workload intensity changes), and a label denotes the phase to which a region is associated, with *P1*, *P2*, and *P3* representing the *Phase 1*, *Phase 2*, and *Phase 3* of the workload, respectively.

As shown in Figures 8b, 8e, 9b, and 9e for APPLEware and in Figures 8c, 8f, 9c, and 9f for FMPC, both controllers are unable to track workload changes since the assigned resource shares (the solid lines) remains essentially constant over time, instead of tracking the related resource utilizations (the dotted lines), and often they are lower than the actual resource demands. We note that this behavior is mainly determined by the inability of the neuro-fuzzy model, used by each controller, to quickly adapt to the time-varying workload.

In particular, the initial decision taken by both APPLEware and FMPC for assigning to each tier of Olio a too low share of memory capacity (in the figures, the solid line for M_{Web} and M_{DB} in the first *P1* region) with respect to the actual demand (in the figures, the dotted line for D_{Web} and D_{DB} in the first *P1* region), leads to the situation where both tiers are not capable to serve incoming

DOI: 10.1002/cpe.3968



(a) CPU allocations and utilizations under FCMS.
(b) CPU allocations and utilizations under APPLEware.
(c) CPU allocations and utilizations under FMPC.
(d) Memory allocations and utilizations under FCMS.
(e) Memory allocations and utilizations under APPLEware.
(f) Memory allocations and utilizations under FMPC.

Figure 8. Resource allocations and utilizations for the Olio Web tier in the $\langle RUBiS, Olio \rangle$ pair under different controllers. Visual markers (•) represent only a subset of actual observations; they are used for improving the readability of the plots.

requests due to lack of memory (e.g., they are unable to spawn new worker threads or processes) and start refusing them.

This, in turn, impacts on the behavior of the RAIN workload generator as follows. When a request is refused, the thread used by RAIN to emulate a user dies, with the consequence that it does not generate any additional request. The resulting effect is that the request rate drops, and the CPU utilization of both tiers drops as well (see the dotted line for U_{Web} and U_{DB}).

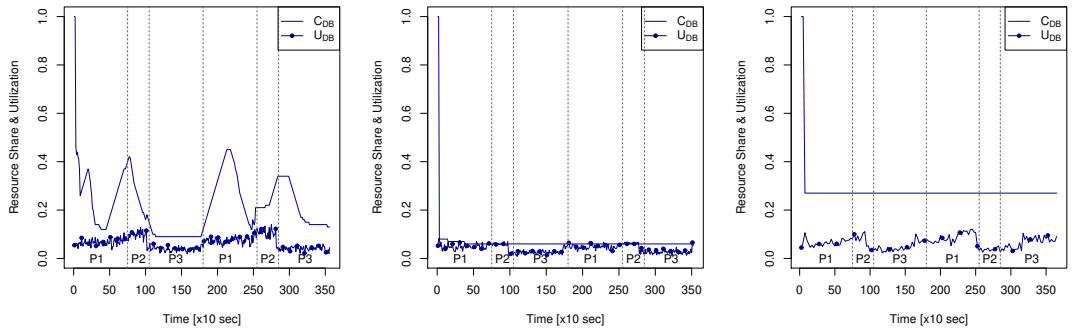
The inability of APPLEware and FMPC to track workload changes and thus to take right control decisions, causes the application SLO to be always violated. This can be seen in Figure 10b for APPLEware and in Figure 10c for FMPC, where the value of the percent error $\widehat{E}$ (which has been limited in the range between -100% and 100%, for the sake of readability) is always positive.

Unlike APPLEware and FMPC, the CPU and memory shares allocated by FCMS to each tier (the solid lines in Figures 8a, 8d, 9a, and 9d) as time goes by exhibits practically the same profile of the corresponding actual utilizations (the dotted lines in the same figures). This means that FCMS is able to determine the actual CPU and memory capacity needs of each tier, and to allocate it suitable CPU and memory shares.

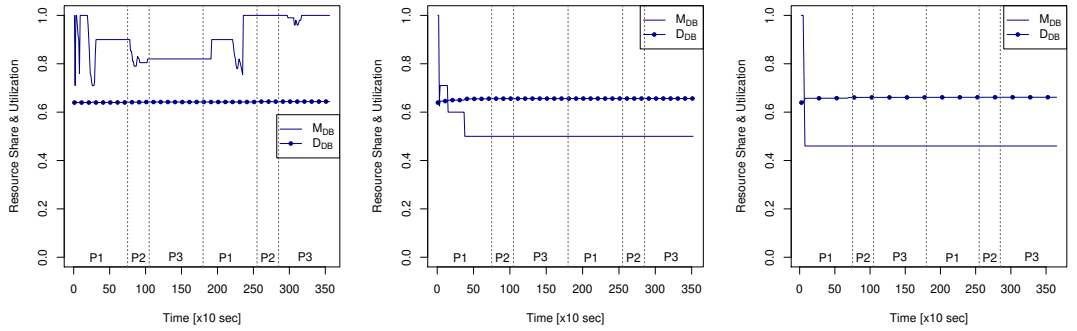
As can be noted from the figures, FCMS always allocate to each tier more CPU and memory capacity than strictly needed. In particular, the difference between allocated and used CPU is reasonably small most of the time and, for all tiers and applications, it ranges between 20% and 25%. This effect is not unexpected, but it is instead a design choice that has been encoded into the membership function for "C_{res}".

For the memory, the situation is somewhat different since FCMS is less aggressive and adjusts the memory in use by a VM according to its available CPU capacity (to avoid the issues discussed in

DOI: 10.1002/cpe.3968

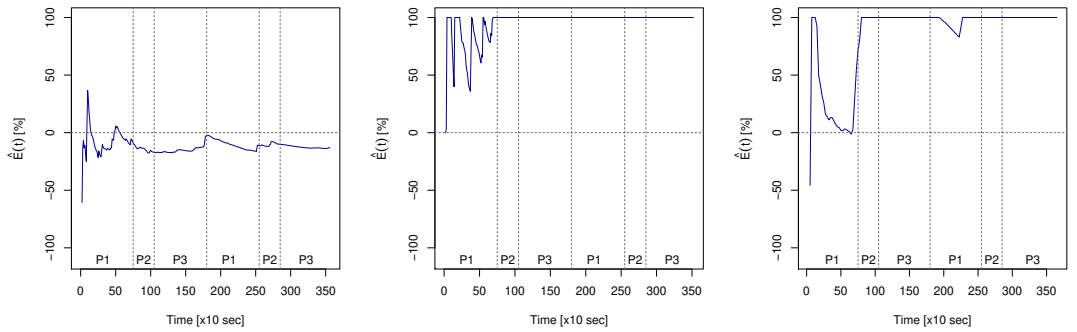


(a) CPU allocations and utilizations under FCMS. (b) CPU allocations and utilizations under APPLEware. (c) CPU allocations and utilizations under FMPC.



(d) Memory allocations and utilizations under FCMS. (e) Memory allocations and utilizations under APPLEware. (f) Memory allocations and utilizations under FMPC.

Figure 9. Resource allocations and utilizations for the Olio DB tier in the $\langle RUBiS, Olio \rangle$ pair under different controllers. Visual markers (•) represent only a subset of actual observations; they are used for improving the readability of the plots.



(a) Evolution of $\widehat{E}$ under FCMS. (b) Evolution of $\widehat{E}$ under APPLEware. (c) Evolution of $\widehat{E}$ under FMPC.

Figure 10. Evolution of $\widehat{E}$ for Olio in the $\langle RUBiS, Olio \rangle$ pair under different controllers.

Section 2). Again, this behavior is by design, and has been encoded into the membership functions for "M_{res}".

DOI: 10.1002/cpe.3968

6. RELATED WORKS

The dynamic management and provisioning of physical resources for virtualized cloud applications with SLO constraints is a topic that has been widely studied in the past years. However, to the best of our knowledge, our work is the only one that focuses on the maximization of the consolidation level under application performance constraints by taking into account the effects of the dynamic allocation of the memory in use by VMs on the CPU utilization.

Modern platform virtualization offer sophisticated automatic memory allocation mechanisms to adjust the memory in use by the VMs they host, based on the current memory pressure regime (e.g., [39, 69, 70]). Furthermore, other approaches (e.g., [18, 71, 72, 73, 74], just to name a few) aim at improving existing memory management techniques by overcoming their limitations and by reducing their overhead (and hence their impact) on the running VMs. However, all of these mechanisms are unaware of application performance and thus may lead to SLO violations, unless they are complemented by approaches like ours. Also, some of these approaches (e.g., [70, 72]) are of limited applicability as they require either a para-virtualized system or a modified version of the hypervisor.

Thus application performance-aware approaches are needed to make sure that application SLOs are met. Many of these approaches (like ours) are based on the dynamic VM vertical scaling technique, which focuses on dynamically provisioning physical resources to one or more VMs in order to achieve the SLOs of the applications they run. Various approaches have been proposed in the literature.

In [75], the authors propose a resource management system consisting of two parallel adaptive linear feedback controllers for CPU and memory, aimed at assuring average-based application SLO metrics. In [76], the authors present a hierarchical system based on adaptive and linear quadratic optimal feedback control theory for the dynamic vertical scaling of VMs in a resource pool hierarchy. In [77], the authors propose a resource management system where a fuzzy controller coordinates two linear feedback controllers to dynamically adjust the amount of CPU and memory required to assure average-based application response times. In this case, coordination only consists in determining the degree of contribution of the CPU and memory controllers according to the actual usage of the respective controlled resource, and it does not take into account the interferences due to the dynamic allocation mechanisms, like we do. Both approaches suffer from the following two drawbacks: (a) they do not coordinate CPU and memory allocation, thus making memory allocation adversely impact on vCPU utilization that, in turn, causes SLA violations (see Section 2), and (b) linear feedback controllers are usually less effective to cope with non-stationary workloads than those based on fuzzy control [19, 21, 22].

In [19], the authors propose APPLEware, an autonomic middleware for the joint performance and power control of co-located web applications in virtualized infrastructures, based on adaptive neuro-fuzzy modeling and on model predictive control. In [20], the authors present FMPC, a resource management system for dynamic VM vertical scaling that is based on adaptive neuro-fuzzy modeling and genetic algorithms. Both approaches use a neuro-fuzzy model to describe the relationships between the application performance (the control output) and the resource allocations (the control inputs), and to design a predictive controller (either based on model predictive control or on genetic algorithms) that aims at reducing the control error by minimizing the allocated physical resources. Besides the uncoordinated allocation of CPU and memory capacity, these approaches suffer from the following drawbacks.

First, the input-output model they use does not take into account the resource usage of the various application tiers, so it cannot detect the tiers that represent a bottleneck and, consequently, suitably differentiate the amount of capacity allocated to each tier.

Second, the control objective does not consider the sign of the performance error, thus making them unable to distinguish between deviations from the SLO value due to SLO violations and ones due to SLO achievement but with over-provisioned resources.

Third, the APPLEware approach, proposed in [19], needs to use linearization to design the model predictive controller, with consequent errors deriving from this operation.

DOI: 10.1002/cpe.3968

In contrast, as already discussed in Section 5, FCMS is able to take into consideration resource bottlenecks at different application tiers (thus assigning them a different resource capacity), to make different decisions according to the sign of the control error (thus differentiating between conditions where SLO is violated and others where it is surely met), and to take into account the interference of dynamic adjustment of the memory in use by each tier on its CPU utilization.

Some recent works, like [78, 79], propose an approach based on application-level ballooning, which extends the concept of memory ballooning to those applications that manage their own memory. These works use a white-box approach, by leveraging the full knowledge of the managed application. On the one hand, these works may be very effective in reaching high consolidation levels and application SLOs since they use a white-box approach, by leveraging the full knowledge of the managed applications. On the other hand, the white-box approach limits their flexibility since they are specifically tailored to selected applications. Instead, we use a black-box approach which is able to work with any application without any prior knowledge.

Our work differs from these related works in the following aspects: (a) it uses a black-box control theoretic approach based on fuzzy logic, (b) it takes control decisions by considering, at each control interval, the resource bottlenecks at different application tiers, the interference of dynamic memory adjustment on vCPU utilization (and, thus, on application performance), and the sign of the performance error, (c) it is able to provide service level assurance for percentile-based SLOs, and (d) it does not require any modification neither to the kernel of the VMs (i.e., it can be applied to fully virtualized systems) nor to the hypervisor, but it simply relies on the mechanisms already offered by the virtualization software.

7. CONCLUSIONS AND FUTURE WORKS

In this paper, we presented FCMS, a dynamic vertical scaling resource management framework, based on feedback fuzzy control, that is able to achieve the best consolidation level (in term of CPU and memory capacity) that can be attained on a physical infrastructure without violating the SLAs of the applications running on it. FCMS works by constantly monitoring application performance and resources usage in order to determine whether and how to adjust the amount of physical CPU and memory capacity allocated to each application tier, so that it is able to properly cope with the highly dynamic, non-stationary and bursty workloads that characterize modern cloud applications, and to avoid the negative effects on application performance caused by the interactions of the mechanisms used for the dynamic CPU and memory allocation.

To assess the efficacy and performance of FCMS, we implemented it on a real testbed, that we used in an thorough experimental evaluation involving 5 real-world cloud applications (namely, Cassandra, Olio, Redis, RUBBoS, and RUBiS), that can be considered representative of the applications that run today on cloud infrastructures, exposed to time-varying and bursty workloads, in presence of resource contention. Furthermore, we compare FCMS against two existing, state-of-the-art dynamic vertical scaling controllers, namely APPLEware and FMPC, aiming at similar goals of FCMS.

Our results show that, in all the experimental scenarios we considered, FCMS outperforms the other approaches, as it is able to achieve a better consolidation level without violating any application SLAs, unlike competing solutions that fail to achieve either one of these goals, or both of them.

As future works, we plan to extend FCMS to incorporate other types of physical resources (e.g., network and disk bandwidth) in addition to CPU and memory. Also, we plan to investigate the use of adaptive control techniques to dynamically adapt the parameters of the membership functions to the incoming workload, and to evaluate its possible benefits in obtaining a better consolidation level, while still achieving application SLAs and keeping the controller design and tuning as simple as possible.

REFERENCES

DOI: 10.1002/cpe.3968

1. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, *et al.*. A view of cloud computing. *Commun. ACM* Apr 2010; **53**(4):50–58, doi:10.1145/1721654.1721672.
2. Guazzone M, Anglano C, Canonico M. Exploiting VM migration for the automated power and performance management of green cloud computing systems. *Proc. of the* 1^{st} *International Workshop on Energy-Efficient Data Centres (E2DC)*, Springer-Verlag, 2012; 81–92, doi:10.1007/978-3-642-33645-4_8.
3. Guazzone M, Anglano C, Sereno M. A game-theoretic approach to coalition formation in green cloud federations. *Proc. of the* 14^{th} *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, IEEE Computer Society, 2014; 618–625, doi:10.1109/CCGrid.2014.37.
4. Amazon Web Services, Inc. Amazon Web Services. Online: `https://aws.amazon.com` 2016.
5. Rackspace. Rackspace: The Open Cloud Company. Online: `http://www.rackspace.com` 2016.
6. Vogels W. Beyond server consolidation. *Queue* Jan/Feb 2008; **6**(1):20–26, doi:10.1145/1348583.1348590.
7. Greenberg A, Hamilton J, Maltz DA, Patel P. The cost of a cloud: Research problems in data center networks. *ACM SIGCOMM Computer Communication Review* Jan 2009; **39**(1), doi:10.1145/1496091.1496103.
8. Maltz DA. Challenges in cloud scale data centers. *SIGMETRICS Performance Evaluation Review* Jun 2013; **41**(1), doi:10.1145/2494232.2465767.
9. Anglano C, Canonico M, Guazzone M, Botta M, Rabellino S, Arena S, Girardi G. Peer-to-peer desktop grids in the real world: The ShareGrid project. *Proc. of the 2008* 8^{th} *IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, IEEE Computer Society, 2008; 609–614, doi:10.1109/CCGRID.2008.23.
10. Anglano C, Canonico M, Guazzone M. The ShareGrid peer-to-peer desktop grid: Infrastructure, applications, and performance evaluation. *Journal of Grid Computing* Dec 2010; **8**(4):543–570, doi:10.1007/s10723-010-9162-z.
11. Wu L, Buyya R. Service level agreement (SLA) in utility computing systems. *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*, Cardellini V, Casalicchio E, Branco KRLJC, Estrella JC, Monaco FJ (eds.). IGI Global, 2011.
12. Amazon Web Services, Inc. Amazon EC2 Service Level Agreement. Online: `https://aws.amazon.com/ec2/sla/` 2016.
13. Mi N, Casale G, Cherkasova L, Smirni E. Injecting realistic burstiness to a traditional client-server benchmark. *Proc. of the* 6^{th} *International Conference on Autonomic Computing (ICAC)*, 2009, doi:10.1109/ICAS.2006.13.
14. Singh R, Sharma U, Cecchet E, Shenoy P. Autonomic mix-aware provisioning for non-stationary data center workloads. *Proc. of the* 7^{th} *International Conference on Autonomic Computing (ICAC)*, 2010, doi:10.1145/1809049.1809053.
15. Lama P, Guo Y, Zhou X. Autonomic performance and power control for co-located web applications on virtualized servers. *Proc. of the 2013 IEEE/ACM* 21^{st} *International Symposium on Quality of Service (IWQoS)*, 2013; 1–10, doi:10.1109/IWQoS.2013.6550266.
16. Xiao Z, Chen Q, Luo H. Automatic scaling of internet applications for cloud computing services. *IEEE Transactions on Computers* May 2014; **63**(5):1111–1123, doi:10.1109/TC.2012.284.
17. Lloyd W, Pallickara S, David O, Lyon J, Arabi M, Rojas K. Performance implications of multi-tier application deployments on infrastructure-as-a-service clouds: Towards performance modeling. *Future Generation Computer Systems* Jul 2013; **29**(5):1254–1264, doi:10.1016/j.future.2012.12.007.
18. Liu H, Jin H, Liao X, Deng W, He B, z Xu C. Hotplug or ballooning: A comparative study on dynamic memory management techniques for virtual machines. *IEEE Transactions on Parallel and Distributed Systems* May 2015; **26**(5):1350–1363, doi:10.1109/TPDS.2014.2320915.
19. Lama P, Guo Y, Jiang C, Zhou X. Autonomic performance and power control for co-located web applications in virtualized datacenters. *IEEE Transactions on Parallel and Distributed Systems* May 2016; **27**(5):1289–1302, doi:10.1109/TPDS.2015.2453971.
20. Wang L, Xu J, Duran-Limon HA, Zhao M. QoS-driven cloud resource management through fuzzy model predictive control. *Proc. 2015 IEEE International Conference on the Autonomic Computing (ICAC)*, 2015; 81–90, doi:10.1109/ICAC.2015.41.
21. Anglano C, Canonico M, Guazzone M. FC2Q: Exploiting fuzzy control in server consolidation for cloud applications with SLA constraints. *Concurrency and Computation: Practice and Experience* 2015; **27**(17):4491–4514, doi:10.1002/cpe.3410.
22. Rao J, Wei Y, Gong J, Xu CZ. QoS guarantees and service differentiation for dynamic cloud applications. *IEEE Transactions on Network and Service Management* Mar 2013; **10**(1), doi:10.1109/TNSM.2012.091012.120238.
23. Guazzone M, Anglano C, Canonico M. Energy-efficient resource management for cloud computing infrastructures. *Proc. of the* 3^{rd} *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE Computer Society, 2011; 424–431, doi:10.1109/CloudCom.2011.63.
24. Albano L, Anglano C, Canonico M, Guazzone M. Fuzzy-Q&E: Achieving QoS guarantees and energy savings for cloud applications with fuzzy control. *Proc. of the* 3^{rd} *International Cloud and Green Computing Conference (CGC)*, IEEE Computer Society, 2013; 159–166, doi:10.1109/CGC.2013.31.
25. Hellerstein J, Diao Y, Parekh S, Tilbury D. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.
26. Wang LX, Mendel J. Fuzzy basis functions, universal approximation, and orthogonal least-squares learning. *IEEE Transactions on Neural Networks* Sep 1992; **3**(5):807–814, doi:10.1109/72.159070.
27. Kosko B. Fuzzy systems as universal approximators. *IEEE Transactions on Computers* Nov 1994; **43**(11):1329–1333, doi:10.1109/12.324566.
28. Ghosh R, Longo F, Xia R, Naik VK, Trivedi KS. Stochastic model driven capacity planning for an infrastructure-as-a-service cloud. *IEEE Transactions on Services Computing* 2014; **7**(4):667–680, doi:10.1109/TSC.2013.44.
29. Ghosh R, Naik VK. Biting off safely more than you can chew: Predictive analytics for resource over-commit in iaas cloud. *Proc. of the 2012 IEEE* 5^{th} *International Conference on Cloud Computing (CLOUD)*, 2012; 25–32, doi:10.1109/CLOUD.2012.131.
30. Urgaonkar B, Shenoy P, Chandra A, Goyal P, Wood T. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems* 2008; **3**(1):1–39, doi:10.1145/1342171.

1342172.
31. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. *Proc. of* 21^{st} *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007; 205–220, doi:10.1145/1294261.1294281.
32. Casale G, Mi N, Cherkasova L, Smirni E. Dealing with burstiness in multi-tier applications: Models and their parameterization. *IEEE Transactions on Software Engineering* Sept 2012; **38**(5):1040–1053, doi:10.1109/TSE.2011.87.
33. Chen Y, Alspaugh S, Katz R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proceedings of the VLDB Endowment* Aug 2012; **5**(12):1802–1813, doi:10.14778/2367502.2367519.
34. Lama P, Zhou X. Coordinated power and performance guarantee with fuzzy mimo control in virtualized server clusters. *IEEE Transactions on Computers* Jan 2015; **64**(1):97–111, doi:10.1109/TC.2013.184.
35. Amazon Web Services, Inc. Amazon EC2 Pricing. Online: `https://aws.amazon.com/ec2/pricing/` 2016.
36. Google. Google Compute Engine Pricing. Online: `https://cloud.google.com/compute/pricing` 2016.
37. Rackspace. Rackspace Cloud Servers Pricing. Online: `http://www.rackspace.com/cloud/servers/pricing/` 2016.
38. Fedorova A, Seltzer M, Smith MD. A non-work-conserving operating system scheduler for SMT processors. *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2006.
39. Waldspurger CA. Memory resource management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.* Dec 2002; **36**(SI):181–194, doi:10.1145/844128.844146.
40. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proc. of the* 19^{th} *ACM Symposyum on Operating Systems Principles (SOSP)*, ACM, 2003; 164–177, doi:10.1145/945445.945462.
41. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. kvm: the linux virtual machine monitor. *Proc. of the Linux Symposium*, vol. 1, 2007; 225–230.
42. Kambadur M, Moseley T, Hank R, Kim MA. Measuring interference between live datacenter applications. *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE Computer Society Press, 2012; 51:1–51:12, doi:10.1109/SC.2012.78.
43. Almeida J, Almeida V, Ardagna D, Cunha Í, Francalanci C, Trubian M. Joint admission control and resource allocation in virtualized servers. *Journal of Parallel and Distributed Computing* 2010; **70**(4):344–362, doi:10.1016/j.jpdc.2009.08.009.
44. Urgaonkar R, Kozat UC, Igarashi K, Neely MJ. Dynamic resource allocation and power management in virtualized data centers. *Proc. of the 2010 IEEE Network Operations and Management Symposium (NOMS)*, 2010; 479–486, doi:10.1109/NOMS.2010.5488484.
45. Borgetto D, Casanova H, Costa GD, Pierson JM. Energy-aware service allocation. *Future Generation Computer Systems* 2012; **28**(5):769–779, doi:10.1016/j.future.2011.04.018.
46. Verma A, Ahuja P, Neogi A. pMapper: Power and migration cost aware application placement in virtualized systems. *Proc. of the* 9^{th} *ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2008; 243–264, doi:10.1007/978-3-540-89856-6_13.
47. Beloglazov A, Abawajy J, Buyya R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems* 2012; **28**(5):755–768, doi:10.1016/j.future.2011.04.017.
48. Liu H, He B. Vmbuddies: Coordinating live migration of multi-tier applications in cloud environments. *IEEE Transactions on Parallel and Distributed Systems* Apr 2015; **26**(4):1192–1205, doi:10.1109/TPDS.2014.2316152.
49. Zao W, Want Z. Dynamic Memory Balancing for Virtual Machines. *Proc. of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 09)*, ACM, USA, 2009, doi:10.1145/1508293.1508297.
50. Mamdani E. Application of fuzzy logic to approximate reasoning using linguistic synthesis. *IEEE Transactions on Computers* 1977; **26**(12):1182–1191, doi:10.1109/TC.1977.1674779.
51. Zadeh LA. Fuzzy sets. *Information and Control* 1965; **8**(3):338–353, doi:10.1016/S0019-9958(65)90241-X.
52. Chen G, Pham T. *Introduction to Fuzzy Sets, Fuzzy Logic, and Fuzzy Control Systems*. CRC Press, 2001.
53. Dunning T, Friedman E. *Practical Machine Learning: A New Look at Anomaly Detection*. O'Reilly Media, 2014.
54. Dunning T. t-digest: an algorithm for computing extremely accurate quantiles. Online: `https://github.com/tdunning/t-digest` 2015.
55. Anglano C, Canonico M, Guazzone M. Repository for the code used in our experimental evaluation. Online: `https://github.com/sguazt/fcms2016` 2016.
56. Red Hat. Libvirt virtualization API. Online: `http://libvirt.org` 2016.
57. Ferdman M, Adileh A, Kocberber O, Volos S, Alisafaee M, Jevdjic D, Kaynak C, Popescu AD, Ailamaki A, Falsafi B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Proc. of the* 17^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012; 37–48, doi:10.1145/2150976.2150982.
58. Apache Software Foundation. The Cassandra project. Online: `http://cassandra.apache.org` 2016.
59. Sobel W, Subramanyam S, Sucharitakul A, Nguyen J, Wong H, Klepchukov A, Patil S, Fox A, Patterson D. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. *Proc. of the Cloud Computing and Its Applications (CCA)*, vol. 8, 2008.
60. Apache Software Foundation. Olio web 2.0 toolkit. Online: `https://incubator.apache.org/projects/olio.html` 2011.
61. Sanfilippo S. Redis. Online: `http://redis.io` 2008.

62. OW2 Consortium. RUBBoS: Rice university bulletin board system. Online: `http://jmob.ow2.org/rubbos.html` 2004.
63. Amza C, Chanda A, Cox A, Elnikety S, Gil R, Rajamani K, Zwaenepoel W, Cecchet E, Marguerite J. Specification and implementation of dynamic web site benchmarks. *Proc. of the IEEE International Workshop on Workload Characterization (WWC-5)*, 2002; 3–13, doi:10.1109/WWC.2002.1226489.
64. OW2 Consortium. RUBiS: Rice university bidding system. Online: `http://rubis.ow2.org/index.html` 2008.
65. Beitch A, Liu B, Yung T, Griffith R, Fox A, Patterson DA. RAIN: A workload generation toolkit for cloud computing applications. *Technical Report UCB/EECS-2010-14*, University of California at Berkeley Feb 2010.
66. The Mathworks, Inc. MATLAB® 2015a. Online: `http://www.mathworks.com`.
67. Jang JSR, Sun CT, Mizutani E. *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*. Prentice Hall, 1997.
68. Ljung L. *System Identification: Theory for the User*. 2nd edn., Prentice Hall, 1999.
69. Magenheimer D. Memory overcommit... without the commitment. *Xen Summit*, 2008; 1–3. Extended abstract.
70. Magenheimer D, Mason C, McCracken D, Hackel K. Transcendent memory and linux. *Proc. of the Linux Symposium*, 2009; 191–200.
71. Schopp JH, Fraser K, Silbermann MJ. Resizing memory with balloons and hotplug. *Proc. of the Linux Symposium*, vol. 2, 2006; 313–319.
72. Kim J, Fedorov V, Gratz PV, Reddy ALN. Dynamic memory pressure aware ballooning. *Proc. of the 2015 International Symposium on Memory Systems (MEMSYS)*, ACM, 2015; 103–112, doi:10.1145/2818950.2818967.
73. Zhang WZ, Xie HC, Hsu CH. Automatic memory control of multiple virtual machines on a consolidated server. *IEEE Transactions on Cloud Computing* 2015; **PP**(99):1–1, doi:10.1109/TCC.2014.2378794.
74. Moltó G, Caballer M, de Alfonso C. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems* 2016; **56**:1–10, doi:10.1016/j.future.2015.10.002.
75. Heo J, Zhu X, Padala P, Wang Z. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. *Proc. of the IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2009; 630–637, doi:10.1109/INM.2009.5188871.
76. Lu L, Zhu X, Griffith R, Padala P, Parikh A, Shah P, Smirni E. Application-driven dynamic vertical scaling of virtual machines in resource pools. *Proc. of the 2014 IEEE Network Operations and Management Symposium (NOMS)*, 2014; 1–9, doi:10.1109/NOMS.2014.6838238.
77. Farokhi S, Lakew EB, Klein C, Brandic I, Elmroth E. Coordinating cpu and memory elasticity controllers to meet service response time constraints. *Proc. of the 2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, 2015; 69–80, doi:10.1109/ICCAC.2015.20.
78. Salomie TI, Alonso G, Roscoe T, Elphinstone K. Application level ballooning for efficient server consolidation. *Proc. of the 8th ACM European Conference on Computer Systems (EuroSys)*, ACM, 2013; 337–350, doi:10.1145/2465351.2465384.
79. Spinner S, Herbst N, Kounev S, Zhu X, Lu L, Uysal M, Griffith R. Proactive memory scaling of virtualized applications. *Proc. of the 2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, 2015; 277–284, doi:10.1109/CLOUD.2015.45.