CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

FC2Q: Exploiting Fuzzy Control in Server Consolidation for Cloud Applications with SLA Constraints

Cosimo Anglano[1]*, Massimo Canonico[1] and Marco Guazzone[1]

[1]*Department of Science and Technological Innovation, University of Piemonte Orientale, Italy*

SUMMARY

Modern cloud data centers rely on server consolidation (the allocation of several Virtual Machines (VMs) on the same physical host) to minimize their costs. Choosing the right consolidation level (how many and which VMs are assigned to a physical server) is a challenging problem, since contemporary multi-tier cloud applications must meet Service Level Agreements (SLAs) in face of highly dynamic, non-stationary, and bursty workloads.
In this paper, we deal with the problem of achieving the best consolidation level that can be attained without violating application SLAs. We tackle this problem by devising FC2Q, a resource management framework exploiting feedback fuzzy-logic control, that is able to dynamically adapt the physical CPU capacity allocated to the tiers of an application in order to precisely match the needs induced by the intensity of its current workload.
We implement FC2Q on a real testbed, and use this implementation to demonstrate its ability of meeting the above goals by means of a thorough experimental evaluation, carried out with real-world cloud applications and workloads. Furthermore, we compare the performance achieved by FC2Q against those attained by existing state-of-the-art alternative solutions, and we show that FC2Q works better than them in all the considered experimental scenarios. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

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

1. INTRODUCTION

In recent years, hosting applications in a cloud has increasingly become an attractive solution that, from the perspective of the enterprise, results in better efficiency and scalability. As consequence, the market has seen the rise of *Cloud Infrastructure Providers* (CIPs), such as Amazon [1] and Rackspace [2], that host these 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.

These applications are typically characterized by a set of *Service Level Agreements* (SLAs), that specify the minimum level of service that must be guaranteed to customers. Failing to meet a SLA usually results in monetary penalties for the CIP. Thus, the CIP has to ensure that, at any given point in time, each application is allocated enough resource capacity to meet its SLAs.

In order to increase its profit, however, each CIP typically resorts to *server consolidation* [3], 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 its customers. In this way, it may reduce the number of

*Correspondence to: Viale T. Michel, 11 – 15121 Alessandria (Italy). Phone: +39 0131 360188. Fax: +39 0131 360198. E-mail: cosimo.anglano@di.unipmn.it

Copyright © 0000 John Wiley & Sons, Ltd.

physical servers that must be switched on to run the VMs of its customers, and at the same time maximize their utilization, so as to reduce both the cost of the energy required to feed them, and their amortized cost [4, 5]. 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 CIPs.

However, the consolidation level on a given server cannot be increased freely, as a trade-off exists between the number of VMs allocated on that 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 SLA. 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.

In this paper, we argue that, in order to maximize the consolidation level, it is necessary to allocate to each VM the smallest amount of capacity it needs to meet its SLAs, and nothing more. This, in turn, requires the ability of precisely estimating the above capacity, and to enforce its allocation when multiple VMs compete for the same resource. 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 SLA of the corresponding application.

Static provisioning techniques [6, 7, 8], where the capacity allocations are not changed at run-time, fail to properly estimate the required CPU capacity when dealing with highly dynamic workloads that fluctuate over multiple time scales [9] and that exhibit a significant burstiness [10]. Hence, they typically result in either SLA violations (in case of underestimation) or poor utilization (in case of overestimation).

Conversely, dynamic *VM vertical scaling* techniques, whereby the CPU capacity allocated to each VM is adjusted at run time to meet workload demand, are considered more appropriate to tackle these issues.

Various vertical scaling approaches have been proposed in the literature (e.g., [11, 12, 13, 14, 15]). Among them, those based on feedback control theory are considered to be very promising, as they are particularly suited to work at a very short time scale. However, the inherent non-linearities of computing systems [16] make the design of such controllers very challenging. Indeed, to the best of our knowledge, existing model-based linear controllers are unsuitable to properly tackle the problem addressed in this paper, due to the linearization operations they have to perform that lead the controller to make inaccurate or even wrong allocation decisions (we show this in our experimental evaluation, in Section 5, where we compare the performance of our approach with the one attained by a state-of-the-art approach that uses linear control-theoretic techniques). To deal with such non-linearities, fuzzy control comes to help as it has been shown that fuzzy systems have very strong functional capabilities (e.g., see [17, 18]). However, even though works that use fuzzy control in computing systems can already be found in literature (e.g., [15, 19]), none of them, to the best of our knowledge, are designed for the problem we tackle in this paper.

In this paper, we propose the *Fuzzy Controller for Consolidation and QoS* (*FC2Q*), a dynamic vertical scaling framework by means of which the CPU capacity required by each VM running on a physical server is continuously estimated as the corresponding application delivers service to its clients, and the allocated capacity is dynamically adjusted to cope with workload variations so that the corresponding SLAs are met.

The core of our proposal is a fuzzy Multiple-Input Single-Output (MISO) controller that, by using as input the deviations of performance and of used CPU capacity from the respective targets, adjusts the CPU capacity allocation of each VM to the smallest amount of capacity that suffices to meet the SLAs of the corresponding application in the sense discussed before, i.e. in the medium to long time scale. Over the short time scale (i.e., when FC2Q takes decision), however, the capacity can still possibly be over-allocated or under-allocated due to the magnitude and duration of the incoming workload. FC2Q is capable of dealing with multi-tier applications in a shared virtualized infrastructure, and with the inherent non-linearities resulting from the interaction of complex applications with time-varying and non-stationary workloads. As a result, it is able to provide percentile-based performance guarantees for both throughput and response time.

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

To demonstrate the ability of FC2Q of meeting its design goals, we implement it on a real testbed, and use this implementation to carry out a thorough experimental evaluation involving a set of real-world cloud applications and workloads. Furthermore, we compare the performance achieved by FC2Q against those attained by two existing state-of-the-art alternative solutions (that we also implement).

Our results show that FC2Q is able to meet the SLAs of applications by using significantly less CPU capacity than its counterparts in all the experimental scenarios we consider. In particular, we show that – while existing alternatives either fail to meet application-level SLAs, or over-allocate CPU capacity, or both – FC2Q is always able to allocate to each application as little capacity as is required to meet its SLA, thus achieving a better consolidation level without violating any SLA.

Our contributions

In this paper, we deal with the problem of achieving the best consolidation level that may be attained to meet the SLA of applications hosted by a CIP under highly dynamic, non-stationary and bursty workloads. While works about the allocation of physical CPU capacity to VMs so to achieve SLAs can already be found in literature, the problem we are tackling in this paper is different. To the best of our knowledge, our paper is the first one focusing on this problem, since existing works focus on the different problem of fulfilling application SLAs without constraining the amount of resources allocated to applications. Indeed, as discussed later, existing solutions fail to achieve either the goal of SLA satisfaction or the maximization of the consolidation level, or both of them.

A preliminary version of this paper has been presented in [20]. The work presented here significantly extends the above paper as follows:

1. we present and thoroughly discuss the design of FC2Q, a resource management framework, based on feedback fuzzy control, that is able to achieve the best consolidation level that can be attained without violating the SLAs of the application running on a physical server;
2. we enhance the design of the fuzzy controller by means of improved membership functions and rule base, as well as of better algorithms for the estimation of the application performance indices;
3. we implement this improved version of FC2Q, as well as two existing feedback controllers, and compare FC2Q against them (and against a static provisioning technique) for a much larger set of operational scenarios including several real-world cloud applications and realistic workloads;
4. we show that FC2Q 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 context for the problem that we tackle. In Section 3, we describe the design of the FC2Q framework. In Section 4, we illustrate the implementation of FC2Q on a real testbed, that we use to carry out an experimental evaluation 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

We consider a computing infrastructure, owned by a CIP, schematically depicted in Figure 1, consisting in a set of physical servers that are managed by a virtualization platform. This infrastructure hosts a set of multi-tier cloud applications, each one providing services to a population of clients. Each tier A_i ($i = 1, \ldots, n$) of an application A is deployed in a VM, which is hosted on one of the 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. In the following, we use the terms "application tier" and "VM" interchangeably.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

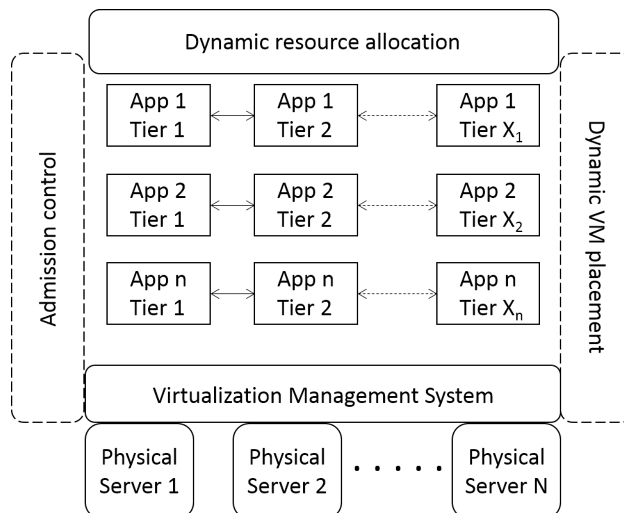


Figure 1. Architecture of the computing infrastructure. Dashed boxes represent components outside the scope of this paper.

The workload of each application A consists in a stream of requests for service coming from the population of its clients, and is characterized by means of its *intensity* $W_A(t)$, that quantifies the amount of work that must be carried out at time t by the computing infrastructure to serve requests. $W_A(t)$ corresponds to the sum of the computing demands of the individual requests arrived at the application and being served or waiting to be served at that time. We assume that the workload is non-stationary, meaning that the distributions of the number of arrived requests, and of their computing demands, change over time.

Each application is characterized by its *Service Level Objectives* (SLOs), expressing the measurable characteristics of SLAs (e.g., response time, throughput, etc.), that must be fulfilled by the CIP in order to avoid paying penalties to its customers. We assume that SLOs are expressed as bounds on a suitable percentile of the distribution of the performance measure of interest (in the literature, these SLOs are termed *percentile-based* [21]). Percentile-based performance guarantees are indeed considered preferable for many applications than simpler metrics like average [22], since they are more robust to fluctuations over multiple time scales that are typical of cloud workloads [23, 24], but they are very challenging to meet [25].

In particular, the SLO of 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 seconds, in the prescribed interval. We assume that, when negotiating the SLAs of application A, the CIP and the customer agree on the maximum value W_A^* of the workload intensity under which the corresponding SLO has be to guaranteed.

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 $(p, r)_A$.

Clearly, for a cloud application there exist multidimensional resource demands, such as CPU, memory, network bandwidth, disk I/O bandwidth, etc. Among them, usually CPU and memory are considered the most representative ones [19, 26], since memory is typically the determining factor on how many VMs a server can run simultaneously, while CPU is the determining factor on application performance. This, moreover, matches the commercial offers of most cloud providers (e.g., Amazon [27] and Rackspace [28]), that charge their users based on their CPU and RAM consumption only. Furthermore, the relationship between application performance and the way

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

other resources are used is still unclear [29], and suitable partitioning mechanisms for such resources are still unavailable [30].

For these reasons in this paper, without loss of generality, we assume that each server has enough memory to accommodate the needs of all the VMs assigned to it, and we focus on controlling the amount of physical CPU capacity $C_{A_i}(t)$ allocated to each tier A_i at time t to match $W_A(t)$. [†] $C_{A_i}(t)$ is a real number, taking values in the interval $[0, 1]$, that expresses the overall fraction of CPU time that must be allocated to the vCPUs of A_i over any time interval, normalized with respect to the number m of physical CPU cores assigned to A_i. For instance, if $C_{A_i}(t) = 0.75$ and $m = 2$, then the two vCPUs of A_i are globally entitled to receive 75% of the total physical capacity of the two CPU cores on which they are allocated (that amounts to $2 \cdot 100\% = 200\%$). The Virtualization Management System may fulfill this obligation in various ways, e.g., by either allocating 75% of each physical core to each vCPU, or by allocating 100% and 50% of a distinct physical core to the first and the second vCPU, respectively or, still, by choosing any other assignment that – when denormalized – yields a value of $200\% \cdot 0.75 = 150\%$.

In order to ensure performance isolation among the VMs running on the same physical machine, we assume that *non-work-conserving* scheduling (whereby the unused portion of $C_{A_i}(t)$ is not assigned to any other VM) is in use on the virtualization platform.

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 CIP 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 [31], 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., [32, 33, 34, 35, 36, 37, 38, 39] for dynamic and contention-aware VM placement, and [40, 41] for admission control).

3. THE FC2Q FRAMEWORK

The architecture of FC2Q is depicted in Figure 2, where we show the components corresponding to a specific application (i.e., the architecture shown in the figure is replicated for each application running on the infrastructure). [‡]

FC2Q associates with each application an *Application Performance Collector*, that is in charge of sampling the performance measure Y used to define the SLO (e.g., response time), and with each one of its tier VM_i a *Fuzzy Controller* (a feedback controller based on fuzzy logic [16, 42]), that periodically determines the CPU capacity C_i to be assigned to VM_i in order to meet the corresponding SLO. Each fuzzy controller makes its decision on the basis of (a) the performance Y attained by the application, (b) the SLO value r, and (c) the utilization U_i of the vCPUs of VM_i.

To properly handle multi-tier applications, fuzzy controllers that are associated to different tiers of the same application are kept synchronized, i.e. they are activated nearly at the same time instant so that they take decisions with respect to the same workload conditions.

The fuzzy controller, whose structure in shown in Figure 3, implements a static fuzzy control logic [42, 43, 44], and determines the CPU capacity allocated in control interval $k + 1$ as $C_i(k + 1) = C_i(k) + \Delta C_i(k)$ (where $C_i(k)$ and $\Delta C_i(k)$ represent the capacity allocated in the current interval, and the adjustment for the next one, respectively). Given that $\Delta C_i(k)$ can take both positive and negative values, the capacity allocated in the interval $k + 1$ may be either higher or lower than the one in interval k (and, of course, it may remain unchanged as well).

[†] We note, however, that our framework can be extended to incorporate also other types of resources. This extension is however left as future work.

[‡] To enhance readability, we drop the identifier of the application from all the subscripts whenever it is clear from the context (i.e., we will denote as i instead of A_i the entities and the quantities corresponding to tier i of application A).

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

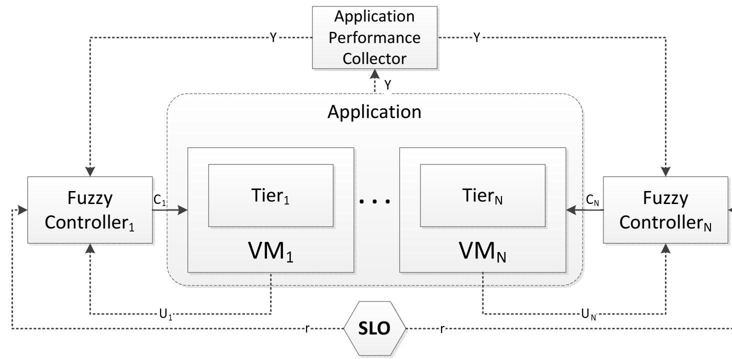


Figure 2. Architecture of the FC2Q system.

The fuzzy controller contains the following four building blocks:

- the *Rule Base*, that stores a set of *fuzzy rules* through which control decisions are made;
- the *Fuzzifier*, that converts real input values 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 fuzzifier, and determines a fuzzy output;
- the *Defuzzifier*, that combines the fuzzy output into a single real value $\Delta C_i(k)$.

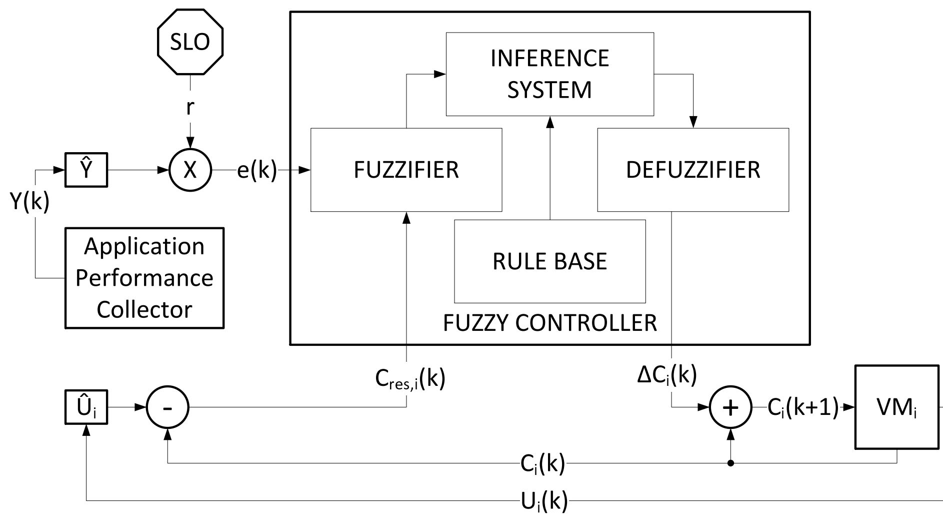


Figure 3. The fuzzy controller component.

3.1. Controller inputs and output

As shown in Figure 3, the fuzzy controller uses two inputs, namely the *relative error* $e(k)$ (the normalized difference between the desired value r and the incremental estimate of the achieved one $\widehat{Y}(k)$) and the *residual capacity* $C_{\text{res},i}(k)$ (the difference between $C_i(k)$ and the actual CPU utilization of the VM), to compute its single output $\Delta C_i(k)$.

The relative error $e(k)$ is computed as in Eq. (1) (in Figure 3 this operation is denoted as X), where we differentiate response time-based from throughput-based SLOs:

$$e(k) = \begin{cases} \frac{r - \widehat{Y}(k)}{r}, & \text{for response time SLOs,} \\ \frac{\widehat{Y}(k) - r}{r}, & \text{for throughput SLOs.} \end{cases} \tag{1}$$

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

where $\widehat{Y}(k)$ is computed by means of the *Stochastic Approximation* algorithm [45]. The error $e(k)$ is used to determine whether the SLO is actually respected ($e(k) \geq 0$) or not ($e(k) < 0$).

The value of $C_{\text{res},i}(k)$ is instead computed as in Eq. (2):

$$C_{\text{res},i}(k) = C_i(k-1) - \widehat{U}_i(k) \tag{2}$$

where $\widehat{U}_i(k)$ is a smoothed value that suitably combines past values of U_i by means of the *Exponentially Weighted Moving Average* (EWMA), that is:

$$\widehat{U}_i(k) = \beta \cdot U_i(k) + (1-\beta) \cdot \widehat{U}_i(k-1) \tag{3}$$

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

$C_{\text{res},i}(k)$ is used to enable the controller to rapidly determine whether (a) the SLO is met (i.e., $e(k) \geq 0$), since the right amount of CPU capacity has been allocated to tier i ($C_{\text{res},i}(k) \cong 0$), so that no corrective actions are needed, or (b) too much capacity has been allocated to tier i ($C_{\text{res},i}(k) \gg 0$), so that $C_i(k+1)$ can be suitably decreased without violating the SLO. Furthermore, it can be used to tell whether the system is approaching saturation ($C_{\text{res},i}(k) = 0$) before $e(k)$ drops below 0, so that $C_i(k+1)$ can be suitably increased before it is too late.

To show that the joint use of $e(k)$ and $C_{\text{res},i}(k)$ enables a controller to properly distinguish among the above situations, while controllers using $e(k)$ and its first-order difference $\Delta e(k)$ (e.g., *DynaQoS* [15]) are not provided with this ability, let us discuss the results we collected in an experiment in which we use DynaQoS to control the CPU capacity allocated to *Olio* [46], a Web application featuring two tiers (named *Web* and *DB*, respectively), under a time-varying workload featuring 100 users for the first 750 seconds of the experiment, 150 users for the next 300 seconds and, finally, 50 users for the last 750 seconds.

In Figure 4 we plot the values of e, Δe, $C_{\text{res,Web}}$, $C_{\text{res,DB}}$, as well as the CPU capacity allocated to the Web tier (C_{Web}) and to the DB tier (C_{DB}), as a function of time.

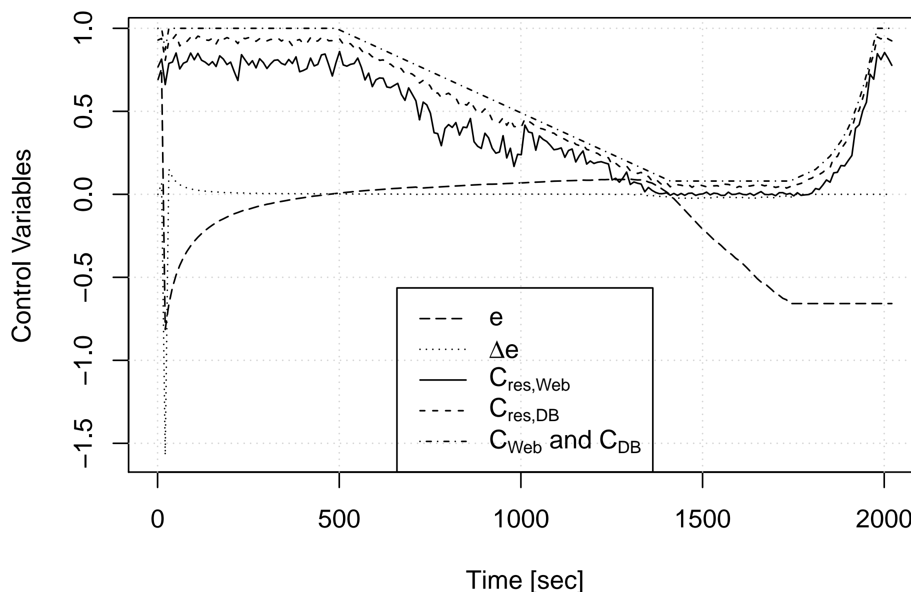


Figure 4. Comparison between Δe and C_{res} control variables: Δe does not provide enough information about the system state.

As can be seen from this graph, after time $t = 1500$, the value of $e(k)$ becomes more and more negative, indicating that the SLO is being violated, while $C_{\text{Web}}(k)$ and $C_{\text{DB}}(k)$ *are not promptly increased* to react to these violations.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

The explanation of this behavior of the DynaQoS controller lies in the fact that $\Delta e(k)$ remains practically constant even after $e(k)$ has dropped below 0. This is due to the fact that $e(k)$ remains constant for a long time after the workload intensity has increased ($t = 1050$), since it takes a possibly large number of measurements to affect the current estimate of the percentile used to define the SLO, and these constant values make also hard for $\Delta e(k)$ to quickly vary.

To suitably react to workload increases, DynaQoS would need a way to tell whether each tier is running short of CPU capacity *before* too much time has elapsed from the increase, *even if* $e(k) \cong 0$. Unfortunately, as discussed above, $e(k)$ (and, consequently, $\Delta e(k)$) vary too slowly to carry such an information.

Conversely, as shown in Figure 4, the values of $C_{\text{res,Web}}$ and $C_{\text{res,DB}}$ start to decrease immediately after the workload intensity increases, so that they can be used as indicators that the system is quickly approaching saturation even when $e(k)$ is still close to 0 and, consequently, the controller could be able to quickly raise C_{Web} and C_{DB} as soon as $e(k)$ becomes negative.

3.2. 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* 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*).

We design the Rule Base by taking into consideration the objective of our controller, namely to meet the SLO (i.e., to keep $e(k) \geq 0$) and, at the same time, to minimize the allocated CPU capacity (i.e., to keep $C_{\text{res},i}(k) \cong 0$). Furthermore, while doing so, we want to avoid that our controller be too aggressive so to limit oscillations that would make the controlled system unstable.

Intuitively, the various rules in the Rule Base encode the following behaviors, as defined by the following situations referring to three distinct scenarios:

- **scenario 1**: if the residual CPU capacity is lacking (i.e., $C_{\text{res},i}(k) = 0$), then the allocated CPU capacity must be either:
  - a) increased by a suitably small amount if the SLO is currently met (i.e., $e(k) \geq 0$), so to avoid that a worsening of the operating conditions (e.g., an increment of the workload intensity) in the next control interval leads to SLO violations, or
  - b) increased by a suitably large amount if the SLO is currently being violated (i.e., $e(k) < 0$), so as to escape from the situation of SLO violation.
- **scenario 2**: if the allocated CPU capacity is not completely saturated (i.e., $C_{\text{res},i}(k) \cong 0$), then it must be either
  - a) increased by a suitably small amount if the SLO is being violated (i.e., $e(k) < 0$), so to make the achieved value $\widehat{Y}(k)$ approaching to the target value r, or
  - b) decreased by a suitably small amount if the SLO is met but the achieved value $\widehat{Y}(k)$ is too far from the target value r (i.e., $e(k) \gg 0$), or
  - c) unchanged if the SLO is met and the achieved value $\widehat{Y}(k)$ is near to the SLO value r (i.e., $e(k) \geq 0$).
- **scenario 3**: if the residual CPU capacity is abundant (i.e., $C_{\text{res},i}(k) > 0$), then the allocated CPU capacity must be either
  - a) decreased by a suitably small amount if the SLO is currently met (i.e., $e(k) \geq 0$), just to be careful that the allocated CPU capacity is reduced without leading to any SLO violation, or
  - b) decreased by a suitably large amount if the SLO is met and the achieved value $\widehat{Y}(k)$ is far from the target value r (i.e., $e(k) \gg 0$), as this is a clear situation of over-allocation, or

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

  - c) unchanged if the SLO is violated (i.e., $e(k) < 0$), as neither an increment nor a decrease would lead to any benefit.

To express the above behaviors, in our Rule Base we use the following linguistic variables and values:

- "e", which is the linguistic counterpart of the $e(k)$ control input, that can take *NEG*, *OK*, or *POS* as linguistic values. The *NEG* value (which stands for "negative") represents the case $e(k) < 0$ (i.e., the SLO is violated), the *OK* value describes the condition $e(k) \geq 0$ (i.e., the achieved value $\widehat{Y}(k)$ is near to the SLO value r, and the SLO is essentially met), and *POS* value (which stands for "positive") is used for the case $e(k) \gg 0$ (i.e., the SLO is surely met).
- "C_{res}", which is the linguistic counterpart of the $C_{\text{res},i}(k)$ control input, that can take *LOW*, *FINE*, or *HIGH* as linguistic values. The *LOW* value is for the situation where $C_{\text{res},i}(k) \cong 0$ (i.e., the allocated CPU capacity is being saturated), *FINE* (i.e., the allocated CPU capacity is not saturated), and *HIGH* (i.e., the allocated CPU capacity is underutilized).
- "ΔC", which is the linguistic counterpart of the $\Delta C_i(k)$ control output, that can take *BUP*, *UP*, *STY*, *DWN*, or *BDW* as linguistic values, whose meaning is defined as follows (where the notation *x.(y)* denotes situation *y* of scenario *x* above):
  - *BUP* (which stands for "big up") represents the output $\Delta C(k) \gg 0$ (i.e., the allocated CPU capacity must be increased by a large amount) and is used in situation *1.(b)*;
  - *UP* (which stands for "up") describes the output $\Delta C(k) > 0$ (i.e., the allocated CPU capacity must be increased by a small amount) and is used in situations *1.(a)* and *2.(a)*;
  - *STY* (which stands for "stay") is for the output $\Delta C(k) \cong 0$ (i.e., the allocated CPU capacity must be unchanged) and is used in situations *2.(c)* and *3.(c)*;
  - *DWN* (which stands for "down") represents the output $\Delta C(k) < 0$ (i.e., the allocated CPU capacity must be decreased by a small amount), and is used in situations *2.(b)* and *3.(a)*;
  - *BDW* (which stands for "big down") is for the output $\Delta C(k) \ll 0$ (i.e., the allocated CPU capacity must be decreased by a large amount), and is used in situation *3.(b)*.

The resulting Rule Base is expressed in a compact form as a table (see Table I) where rows and columns report the values of "C_{res}" and "e", respectively, and where each cell (A, B) contains the value taken by "ΔC" when "C_{res}" is A and "e" is B. In the following, we denote each rule as a pair (A, B), each one representing respectively the linguistic value of "C_{res}" and "e".

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

Table I. The Rule Base.

For instance, *(LOW, NEG)* corresponds to the rule: *if "C_{res}" is LOW and "e" is NEG then "ΔC" is BUP*. This rule encodes the control knowledge stating that if the CPU capacity value used by the VM is close to that allocated by the controller (encoded by the *"C_{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 fuzzy controller has to significantly increase the allocated CPU capacity value (encoded by the *"ΔC" is BUP* proposition).

The agnosticism of the Rule Base with respect to the workload characteristics 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 $C_{\text{res},i}(k)$ 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

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

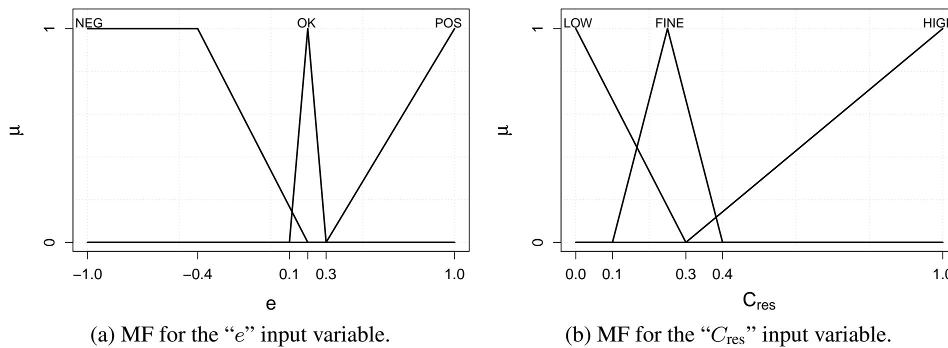
the performance measures of interests as induced by the workload, our controller is able to closely follow any change in the workload, provided that it does not saturate the maximum capacity of the physical resource.

3.3. The Fuzzifier

The *Fuzzifier* converts each real input value (either $e(k)$ or $C_{\text{res},i}(k)$) into the equivalent linguistic variable (either "e" or "C_{res}", respectively), and assigns it one or more linguistic values.

In fuzzy logic, indeed, a single real value may correspond to different linguistic values, each one characterized by a (possibly different) degree of *certainty*. The fuzzifier, therefore, given an input value α, computes the certainty $\mu(\alpha)$ (a value between 0 an 1) that α corresponds to each one of the possible linguistic values assumable by the corresponding linguistic variable.

This is accomplished by using a *Membership Function* (MF) for each (numeric) input variable x that expresses, for each possible linguistic value m it may take, the certainty $\mu(m)$ that the corresponding linguistic variable "x" falls into the range defined by m, and we denote it as $\mu(\text{"}x\text{"}, m)$. In our controller, we use the *triangular-shaped* and *ramp-shaped* MFs, which are the most commonly used membership functions in practice, that are shown in Figure 5.



(a) MF for the "e" input variable. (b) MF for the "C_{res}" input variable.

Figure 5. The MFs for the input variables "e" and "C_{res}".

As shown in this figure, we have two MFs (one for each input variable). In each MF, each possible linguistic value (i.e., *LOW*, *FINE*, and *HIGH* for the "C_{res}" variable, and *NEG*, *OK*, and *POS* for the "e" variable) corresponds to a distinct curve (labeled correspondingly). Each curve is actually a function that, given a value α of the numeric input (reported in the x-axis), yields the certainty that α maps to the corresponding linguistic value.

To exemplify, if $e(k) = 0.2$, the MF in Figure 5a associates to this input the value *OK* with certainty 1.0, since in the MF the value 0.2 projects up to a peak of the membership function corresponding to the linguistic value *OK*. This corresponds to state that the linguistic variable "e" takes the value *OK* with certainty 1.0, i.e., that $\mu(\text{"}e\text{"}, OK) = 1.0$.

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

3.4. The Inference System

The *Inference System* determines the set of rules (the *active rules*) that can be applied given the current values of the input linguistic variables provided by the Fuzzifier. To determine these rules, the Inference System checks whether at least one of the linguistic expressions in its antecedent has a certainty value higher than 0, and, to do so, it applies the *min* or the *max* function depending on whether the antecedents are joined by a conjunction (i.e., "AND") or by a disjunction (i.e., "OR"), respectively.

To exemplify, assume that $\mu(\text{"}C_{\text{res}}\text{"}, HIGH) = 1.0$ and $\mu(\text{"}e\text{"}, OK) = 0.6$ (and that the certainties of all the other linguistic values are 0). By looking at Table I, we see that only the rule corresponding

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

to *(HIGH, OK)* applies, thus resulting in a consequent with a certainty of 0 for all the linguistic values but *DWN*, whose certainty is given by:

$$\mu(\text{“}\Delta C\text{”}, DWN) = \min\big(\mu(\text{“}C_{\text{res}}\text{”}, HIGH), \mu(\text{“}e\text{”}, OK)\big) = \min(1.0, 0.6) = 0.6$$

Conversely, if $\mu(\text{“}C_{\text{res}}\text{”}, FINE) = 0.133$, $\mu(\text{“}C_{\text{res}}\text{”}, LOW) = 0.6$, and $\mu(\text{“}e\text{”}, OK) = 1.0$ (while the certainties of all the other linguistic values are 0), then *two* rules apply, namely *(LOW, OK)* and *(FINE, OK)*, each one with a different certainty degree. According to the Rule Base (see Table I), the consequent of *(LOW, OK)* is *UP*, while the one of *(FINE, OK)* is *STY*. The certainty of these two consequents, namely $\mu(\text{“}\Delta C\text{”}, UP)$ and $\mu(\text{“}\Delta C\text{”}, STY)$, is computed as

$$\mu(\text{“}\Delta C\text{”}, UP) = \min\big(\mu(\text{“}C_{\text{res}}\text{”}, LOW), \mu(\text{“}e\text{”}, OK)\big) = \min(0.6, 1.0) = 0.6,$$
$$\mu(\text{“}\Delta C\text{”}, STY) = \min\big(\mu(\text{“}C_{\text{res}}\text{”}, FINE), \mu(\text{“}e\text{”}, OK)\big) = \min(0.133, 1.0) = 0.133.$$

After the Inference System determines the consequents implied by the active rules, it passes their certainties to the defuzzification process to obtain a single real output value (i.e., ΔC_i in our case).

3.5. The Defuzzifier

Finally, the *Defuzzifier* combines the active rules by means of the *centroid method* [42] and calculates the control output resulting from the combination of the conclusions of all the rules identified by the Inference System.

In the *centroid method*, the numeric value of the control output is computed as a weighted average of the certainty of the fuzzy conclusions, where the weight of each conclusion is the center point of the *output membership function* (i.e., the MF for the output variable). Similarly to what we do for the fuzzifier component, we quantify the linguistic values by means of the *triangular-shaped membership functions* shown in Figure 6.

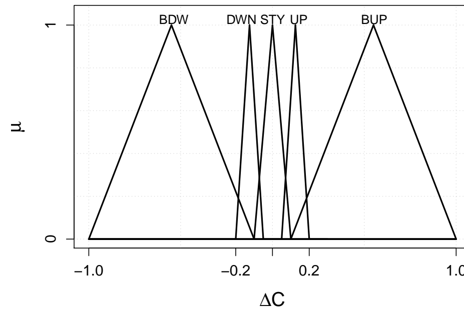


Figure 6. The MF for the output variable “ΔC”.

The centroid method returns the center of the area under the curve of the MFs involved in the active rules. For instance, if $\mu(\text{“}\Delta C\text{”}, UP) = 0.6$ and $\mu(\text{“}\Delta C\text{”}, STY) = 0.133$, the area under these MFs is the gray one shown in Figure 7. As expected, the gray area for the *UP* MF is greater than the gray area for the *STY* MF. This is due to the fact that the degree of certainty of *UP* is higher than the one of *STY*. Figure 7 also shows the centroid of the gray area (i.e., the black-filled circle at $(0.097, 0.168)$). The x-coordinate of the centroid is the output of the defuzzification process which in our case corresponds to the value for $\Delta C_i(k)$. In particular, in our example, $\Delta C_i(k) = 0.097$ which means that the fuzzy controller has to increase the allocated CPU capacity value by 9.7% of the full capacity.

4. SYSTEM IMPLEMENTATION

In order to asses the capability of FC2Q to achieve its design goals, we experimentally evaluate it by using a testbed (whose architecture is shown in Figure 8) that we developed for this purpose.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

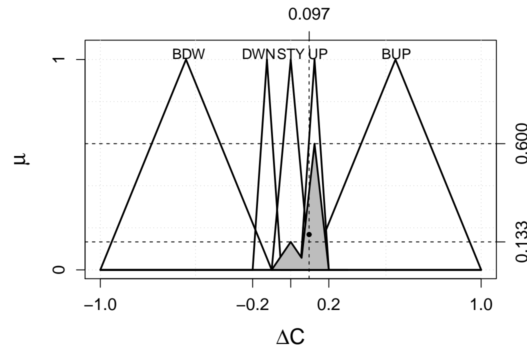


Figure 7. Defuzzification of the aggregate output with the centroid method. The black-filled circle at $(0.097, 0.168)$ is the centroid of the gray area.

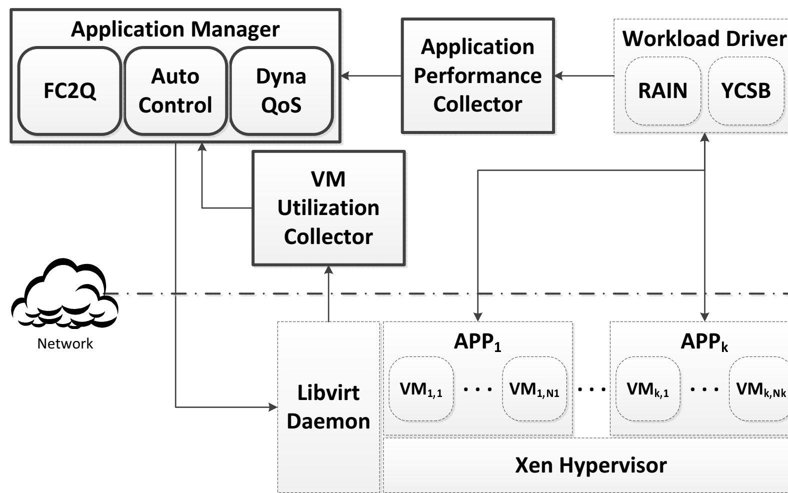


Figure 8. Architectural diagram of our experimental testbed.

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 a feedback controller that allocates CPU capacity to the various tiers on the basis of the inputs provided by the *VM Utilization Collector* and the *Application Performance Collector*. According to the FC2Q approach, each tier is associated to its own controller.

Both the Application Manager and the Workload Drivers are pluggable, that is they allow for an easy replacement of the control algorithm and of the workload generator. As indicated in the figure, we have currently integrated into the testbed three distinct controllers (FC2Q, *DynaQoS* [15], and *AutoControl* [12]), and two distinct workload generators (RAIN [47] and YCSB [48]). These components are used to carry out the experimental evaluation discussed in Section 5.

4.1. Physical Testbed

Our testbed consists of two Fujitsu Server PRIMERGY RX300 S7, connected via a Gigabit Ethernet switch, each one equipped with two 2.4GHz Intel Xeon E5-2665 processors with eight cores each, and with 96GB of RAM. Both machines run the Linux kernel version $3.10.11$ and the Xen [49] version 4.2 virtualization platform.

One of these machine is used to run the virtualized cloud applications, and it is configured to use the Xen *credit scheduler* to enable non-work-conserving scheduling. Under this scheduler, each VM_i is associated with a *weight* w_i (representing the scheduling priority of VM_i with respect to

other VMs) and a *cap* Γ_i, an integer value 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). Thus, to implement non-work-conserving scheduling, at each control interval k we set the weight of all VMs to the same value, and we set the cap assigned to VM_i as $\Gamma_i = \lceil C_i(k) \cdot 100 \cdot m_i \rceil$ (recall that $C_i(k) \in [0, 1]$).

The *libvirt* daemon [50] is also run on this machine in order to enable the Application Manager to remotely interact with the hypervisor to set the cap to all the various VMs, and to the VM Utilization Collector to harvest their utilization values.

The other physical machine is instead used to run the Application Manager, the Application Performance Collector, the VM Utilization Collector, and the Workload Driver.

4.2. Implementation of FC2Q

The various modules composing FC2Q have been implemented in `C++`, and are publicly available on [51].

The Application Performance Collector The Application Performance Collector gathers SLO-defined performance measures (e.g., application response time or throughput) from a specific virtualized application by interacting with the Workload Driver used to generate its workload. In particular, given that both RAIN and YCSB write into their own log files the measures delivered by the application they target, for each application we write a specific module able to extract from these log files the performance measures of interests.

The VM Utilization Collector The VM Utilization Collector collects the information concerning the utilization of the vCPUs of each VM i by using the libvirt API to query Xen for the CPU utilization induced by each one of the VMs running on the testbed.

The Application Manager The Application Manager is responsible to manage the VMs of a virtualized application in order to guarantee the related SLO. It incorporates a controller, and various mechanisms enabling it to interact with both the collectors discussed above to obtain the information that are given as input to the controller. The Application Manager has a pluggable architecture, that allows a simple replacement of the controller used to handle applications. As already mentioned, we have implemented three different controllers, namely our fuzzy controller, as well as *AutoControl* and *DynaQoS*. For the implementation of both FC2Q and DynaQoS, we use the *fuzzylite* library [52].

5. EXPERIMENTAL RESULTS

To assess the ability of FC2Q to use as little CPU capacity as possible to meet the application SLO (so as to maximize the consolidation level), we carry out an extensive experimental evaluation in which several cloud applications, processing a stream of requests generated according to suitably chosen workloads, are executed on our testbed.

In particular, we consider two distinct scenarios. In the first one, named *Isolation*, we run each application alone to verify the performance of FC2Q when no resource contention is present. In the second one, named *Consolidation*, we instead simultaneously run pairs of cloud applications on the testbed in order to assess the ability of FC2Q to effectively multiplex these applications on the same physical infrastructure when resource contention is present.

To compare FC2Q against state-of-the-art alternative solutions, we also run experiments in which the control is carried out by two different controllers, namely the already mentioned *DynaQoS* (which is based on fuzzy control theory) and *AutoControl* (a model-based adaptive feedback linear controller), that we implemented for this purpose.

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, showing that

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

FC2Q outperforms its alternative counterparts in both the scenarios we consider (Section 5.4.1 and Section 5.4.2), and that it is able, at the same time, to closely track workload variations and to precisely allocate to each application tier the amount of CPU capacity it needs to meet the corresponding SLO (Section 5.4.3). Finally, we close this section with a summary discussion of our findings (Section 5.5).

5.1. Performance Metrics

In our experiments, controller performance is assessed by computing two distinct metrics, namely: (a) the *Mean CPU Capacity* (MCC) allocated to each application tier, defined as the average of the CPU capacity allocated in each control interval, and (b) the *percent error* $\widehat{E}$, defined as the relative percentage change between the achieved value $\hat{Y}$ and the SLO value r, that is:

$$\widehat{E} = \begin{cases} 100\frac{\hat{Y}-r}{|r|}, & \text{for response time SLOs,} \\ 100\frac{r-\hat{Y}}{|r|}, & \text{for throughput SLOs.} \end{cases} \quad (4)$$

These metrics are used to rank controllers as follows: if $\widehat{E} > 0$, than the controller is violating the SLO, and as such is not considered to be suitable to solve the problem addressed in this paper. Conversely, if $\widehat{E} \leq 0$, the lower the achieved MCC value, the better the controller (i.e., among two controllers that both meet the SLO, the best one is that which uses less CPU capacity).

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

5.2. Cloud Applications and Workloads

We consider three applications representative of those that run on today's CIP virtualized infrastructures [53], namely:

- *RUBiS* [54], a two-tier Web 1.0 Internet application that implements an auction site prototype modeled after eBay.com. For our experiments, we choose the RUBiS PHP version developed by the *OW2 Consortium* [55], that we patched in order to fix some bugs we found during our experimentation (we release the patched version in the public domain [51]).
- *Olio* [46], a two-tier Web 2.0 Internet application for social events modeled after the Yahoo! Upcoming service. For our experiments, we choose the PHP-based Olio from the *Apache Software Foundation* [56], that we patched in order to fix some bugs we found during our experimentation (we release the patched version in the public domain [51]).
- *Cassandra* [57], a Java-based NoSQL data serving application (originally developed by Facebook) designed to handle large amounts of data across many commodity servers.

We deploy and run each application tier inside a separate VM. Specifically, for both RUBiS and Olio, we setup two VMs, one for the Web tier and another one for the DB tier, each one with 1 vCPU, 2GB of RAM, and 30GB of disk space. Instead, for Cassandra, being a single-tier application, we setup a unique VM equipped with 10 vCPUs, 16GB of RAM, and 30GB of disk space.

To drive the workload of the above applications, we rely on the *RAIN* toolkit [47] (for both RUBiS and Olio) and the *Yahoo! Cloud Serving Benchmark* (YCSB) [48] (for Cassandra), two widely used workload generators for cloud applications that we extended whenever needed to make them fit within our testbed (these extensions are now part of their official repositories).

In particular, for both Olio and RUBiS we use the "default mix" matrix of RAIN (that includes a mix of the various operations supported by the benchmark) with a negative exponentially distributed think-time whose mean is set to 7 seconds. For Cassandra we instead use an YCSB setup featuring the loading of $5,000,000$ records and $100,000$ operations per second as target.

To reproduce realistic operational conditions, characterized by time-varying and bursty workloads, for each cloud application we create a step-like workload with three phases (one phase for each burst), each one characterized by a specific intensity and duration, as shown in Table II. As shown in this table, each workload starts with a medium intensity (*Phase 1*), then continues with a higher intensity phase (*Phase 2*), and then it ends with a lower intensity phase (*Phase 3*).

Copyright © 0000 John Wiley & Sons, Ltd.

Cloud Application	1st Phase Load	1st Phase Duration	2nd Phase Load	2nd Phase Duration	3rd Phase Load	3rd Phase Duration
RUBiS	30 usr	750 sec	45 usr	350 sec	15 usr	750 sec
Olio	100 usr	750 sec	150 usr	350 sec	50 usr	750 sec
Cassandra	12 thd	$30,503,520$ op	3 thd	$13,816,440$ op	6 thd	$11,556,540$ op

Table II. Experimental setup – Three-phase workload parameters. The abbreviations "op", "usr" and "thd" mean "number of operations", "number of users", and "number of threads", respectively.

The SLO (p, r) for each cloud application, shown in Table III, has been set as follows to make it feasible with the physical resources available in our testbed. Each application has been profiled by exclusively assigning a distinct physical CPU core to each one of its vCPUs, and by running it in isolation under the maximal intensity workload (i.e., under *Phase 2*). In each of these profiling experiments, we computed the empirical distribution of the values of the SLO performance metric of interest (i.e., the response times for Olio and RUBiS, and the throughput for Cassandra), from which we determined the value r corresponding to the p-th percentile, and we set the SLO to (p, r).

In particular, for Olio and RUBiS we chose $p = 95$. For Cassandra we instead chose $p = 5$ since, being the throughput the performance metric of interest, the higher the throughput, the better the performance. Thus, if the 5th percentile of the throughput distribution is T, it means that 95% of the observed throughput values are greater than or equal to T.

Application	p	r
RUBiS	95	0.6186 sec
Olio	95	0.1292 sec
Cassandra	5	$6,348.504$ op/sec

Table III. Experimental setup – Application SLOs.

5.3. *Controller Parameters*

Each controller we consider in our evaluations needs specific parameters to be set. The values used in our experiments are reported below.

All the controllers require to set the value of the *sampling time* s (representing the distance in time between two consecutive measurements of the SLO performance metric of interest), and of the *control time* c (representing the distance in time between two consecutive controller activations). In our experiments, we set $s = 2$ seconds and $c = 10$ seconds for all the controllers.

Furthermore, for each controller we set its specific parameters as follows. For FC2Q, we set $\beta = 0.9$ (where β is the smoothing factor, see Section 3.1). For DynaQoS, we set the *discount factor* γ to 0.8, as suggested in [15]. Finally, for AutoControl, we set the *stability factor* q to 2 (as suggested in [12]) and the *forgetting factor* λ, used by the RLS algorithm, to 0.98 (a commonly used value which means that the weight of past observations decays very rapidly with time).

5.4. *Results*

Let us now discuss the results collected in our experiments. We start with those obtained with the Isolation scenario (Section 5.4.1), and then we move to those corresponding to the Consolidation scenario (Section 5.4.2). Finally, we show and discuss, for selected applications and scenarios, how FC2Q is able to track workload variations and to precisely allocate to each application tier the amount of CPU capacity it needs to meet the corresponding SLO (Section 5.4.3).

5.4.1. The Isolation Scenario As already anticipated, in the *Isolation* scenario we execute a single application at a time, and, for every associated VM, we pin each vCPU to a dedicated physical CPU core (the remaining CPU cores are allocated to Xen's Domain-0).

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

The results we collect for each applications are shown in Tables IV (Olio), V (RUBiS), and VI (Cassandra), where rows correspond to applications, and columns report the values of $\widehat{E}$ (together with a textual annotation indicating whether the SLO has been satisfied or not) and of MCC attained for each application.

	SLO		MCC	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)
AutoControl	No	1041.26	86.21	64.94
DynaQoS	No	26.09	82.92	82.92
FC2Q	Yes	−44.55	76.45	61.95

Table IV. Isolation scenario – Results for the Olio application.

	SLO		MCC	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)
AutoControl	Yes	−52.99	100.00	100.00
DynaQoS	No	1099.41	62.36	62.36
FC2Q	Yes	−24.42	55.78	65.25

Table V. Isolation scenario – Results for the RUBiS application.

These results indicate that FC2Q outperforms both DynaQoS and AutoControl for all the applications we consider, since it is always able to meet the corresponding SLOs ($\widehat{E}$ is negative) by allocating the smallest MCC value.

The ranking between DynaQoS and AutoControl varies according to the specific cloud applications.

For Olio (Table IV), DynaQoS ranks to the second place since, although it meets the SLO, its MCC value is much higher than FC2Q, while AutoControl ranks to the last place, since it achieves a very large value of $\widehat{E}$ (corresponding to an achieved $\widehat{Y}$ value 10 times larger than the target SLO value) and a MCC value larger than the other two controllers. This is due to the use of a linear model to describe the relationships between controller inputs and output, that are instead strongly nonlinear. As a consequence, in many cases AutoControl generates negative values of $C_i(k+1)$ that (given that the CPU capacity assigned to a VM cannot be negative) are translated into no control action (the alternative would be that of lowering $C_i(k+1)$ to 0, which is clearly unacceptable).

For RUBiS (Table V) the ranking between DynaQoS and AutoControl is the opposite. AutoControl ranks to the second place, as it meets the SLO but over-allocates CPU capacity, as demonstrated by its high MCC value (that translates into a large negative value of $\widehat{E}$, corresponding to an achieved value $\widehat{Y}$ much lower than necessary). Conversely, DynaQoS ranks to the last place since it does not meet the SLO and allocates more CPU capacity than FC2Q to both application tiers. This is due to the problem already discussed in Section 3.1, i.e., to the inability of Δe to quickly track changes in workload intensity.

	SLO		MCC
	Satisfied?	$\widehat{E}$	Single Tier (%)
AutoControl	No	91.49	45.79
DynaQoS	No	98.28	58.17
FC2Q	Yes	−0.29	67.84

Table VI. Isolation scenario – Results for the Cassandra application.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

Finally, for Cassandra (Table VI), both controllers ranks to the last place, since they both do not meet the SLO of the application, since they do not allocate enough CPU capacity to the corresponding VMs.

Thus, we can conclude that FC2Q outperforms its counterparts in scenarios where applications run in isolation, i.e., when no resource contention is present.

5.4.2. The Consolidation Scenario As already anticipated, in the *Consolidation* scenario 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 three distinct pairs that can be obtained by combining the three cloud applications. To reduce space, in this paper we report the results for two of these pairs namely, $\langle RUBiS, Olio \rangle$ and $\langle RUBiS, Cassandra \rangle$ (however, the results for the third pair 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 $\langle RUBiS, Olio \rangle$ pair, we pin the VMs running the same application tier (either *Web* or *DB*) to the same physical CPU core (e.g., the two VMs running the Web tiers were assigned to the physical CPU core 0). Conversely, for the $\langle RUBiS, Cassandra \rangle$ pair, we pin the two VMs of RUBiS to two of the 10 physical CPU cores used by the Cassandra's VM (recall that this VM has 10 vCPUs).

To provide a more thorough comparison, we add a static capacity provisioning approach (that we name *Static*), in which each application tier receives a fixed CPU capacity, to AutoControl and DynaQoS. The capacity allocated to each tier is set to the 75th percentile of the CPU utilization measured for that tier during the experiment performed to compute the SLO of the application (as discussed in Section 5.2). We consider such a percentile a good compromise between the need of achieving a reasonable consolidation level and of allocating enough CPU capacity to each application to meet its SLO.

The results corresponding to the $\langle RUBiS, Olio \rangle$ and to the $\langle RUBiS, Cassandra \rangle$ pairs are shown in Table VII and Table VIII, respectively.

(a) RUBiS Application.

	SLO		*MCC*	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)
AutoControl	Yes	-23.66	95.77	88.37
DynaQoS	No	762.81	42.39	42.38
FC2Q	Yes	-16.07	60.03	70.10
Static	Yes	-21.83	58.00	82.00

(b) Olio Application.

	SLO		*MCC*	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)
AutoControl	No	220.99	91.02	97.23
DynaQoS	No	2459.59	46.29	46.29
FC2Q	Yes	-12.52	56.83	42.36
Static	No	140.67	31.00	12.00

Table VII. Consolidation scenario – Results for RUBiS and Olio.

From these results, we see that in both cases FC2Q is the best controller, being it the only one able to meet the SLO of both applications. Conversely, none of the other alternatives is able to do the same: each one of them is able to meet at most the SLO for one of the applications, but not for the other one. In particular, DynaQoS fails to satisfy the SLO for both applications, while AutoControl

Copyright © 0000 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

DOI: 10.1002/cpe

(a) RUBiS Application.

	SLO		MCC	
	Satisfied?	$\widehat{E}$	Web (%)	DB (%)
AutoControl	Yes	-53.28	99.66	99.95
DynaQoS	No	1691, 18	63.88	63.88
FC2Q	Yes	-17.69	55.05	64.53
Static	Yes	-19.97	58.00	82.00

(b) Cassandra Application.

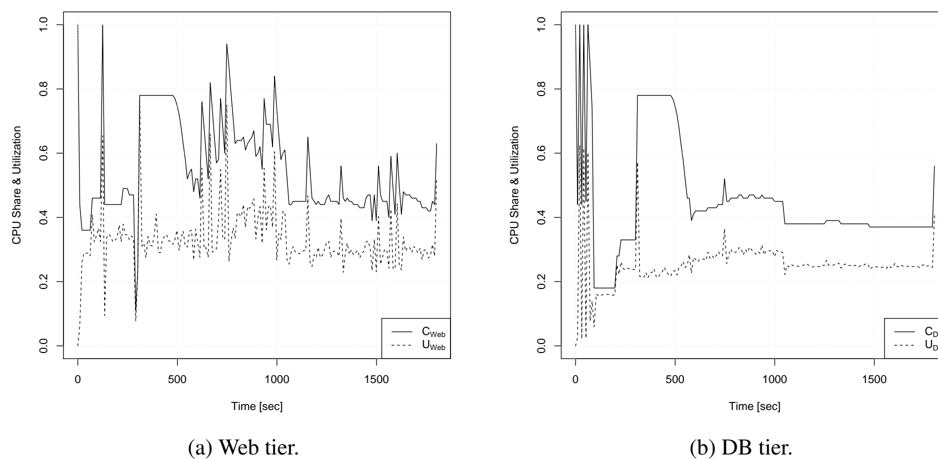
	SLO		MCC
	Satisfied?	$\widehat{E}$	Single Tier (%)
AutoControl	No	88.74	51.10
DynaQoS	No	98.47	38.03
FC2Q	Yes	-0.62	78.88
Static	No	1.06	54.00

Table VIII. Consolidation scenario – Results for RUBiS and Cassandra

and Static are able to do so only for RUBiS, but by allocating an MCC value much higher than FC2Q to both tiers (AutoControl), or to the *DB* tier (Static).

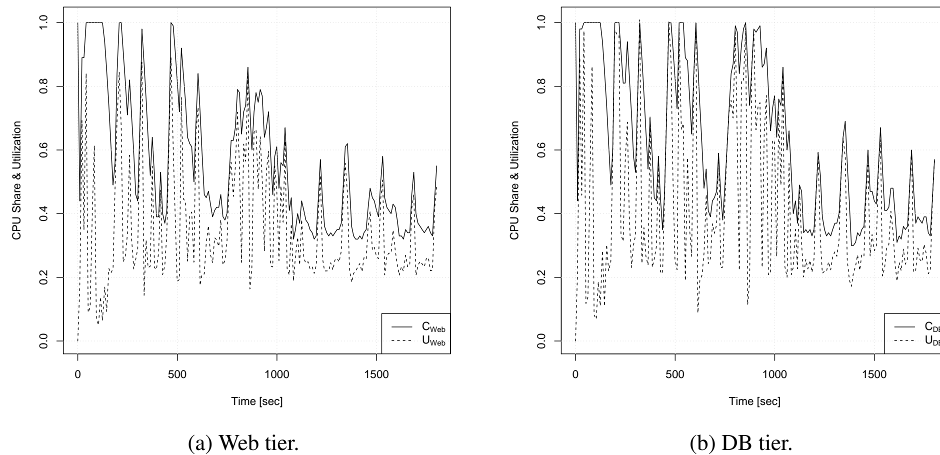
Thus, we can conclude that FC2Q outperforms its counterparts also in scenarios where resource contention is present.

5.4.3. Workload Tracking and Allocation Precision To show that FC2Q is able to track changes in workload variations, i.e. to increase or decrease the amount of CPU capacity allocated to each tier in response to increases or decreases in workload intensity, in this section we report the graphs in which we plot – for selected applications and scenarios – the CPU share allocated to each tier, as well as the actual CPU utilization of that tier. In particular, we report the graphs for Olio (Figure 9) and RUBiS (Figure 10) in the Isolation scenario, and those for the pair $\langle RUBiS, Cassandra \rangle$ (Figure 11) in the Consolidation scenario. The results for the other cases, however, do not significantly differ from those reported here.



(a) Web tier. (b) DB tier.

Figure 9. Isolation scenario – Olio: CPU share and utilization as observed under the control of FC2Q.



(a) Web tier.

(b) DB tier.

Figure 10. Isolation scenario – RUBiS: CPU share and utilization as observed under the control of FC2Q.

By comparing, for each tier, the allocated CPU share and its actual CPU utilization, it is possible to determine whether FC2Q is able to track changes in the workload. The actual CPU utilization of a tier represents indeed the smallest amount of CPU capacity that must be allocated in order to meet the corresponding application SLO. Thus, its evolution over time (the dashed line in each graph) represents the value that must be tracked by FC2Q when allocating the CPU share to the corresponding tier.

As shown in all the figures, the CPU share allocated by FC2Q to each tier (the solid line) as time goes by exhibits practically the same profile of the corresponding actual CPU utilization. This means that FC2Q is able to determine the actual capacity needs of each tier, and to allocate it a suitable CPU share.

However, at any time instant t, the two curves present a practically constant offset, meaning that FC2Q allocates to each tier more CPU capacity than strictly needed. In particular, at any time t, the difference between allocated and used CPU capacity is reasonably small and, for all tiers and applications, it ranges between 20% and 25%. This effect is not unexpected, but it is instead a design choice. Our rules (and, more precisely, the membership functions) have been indeed designed in such a way to allocate to each tier a small amount of extra CPU capacity with respect to the optimal value, so that small workload variations can be tolerated without violating the SLO before the next activation of FC2Q takes place.

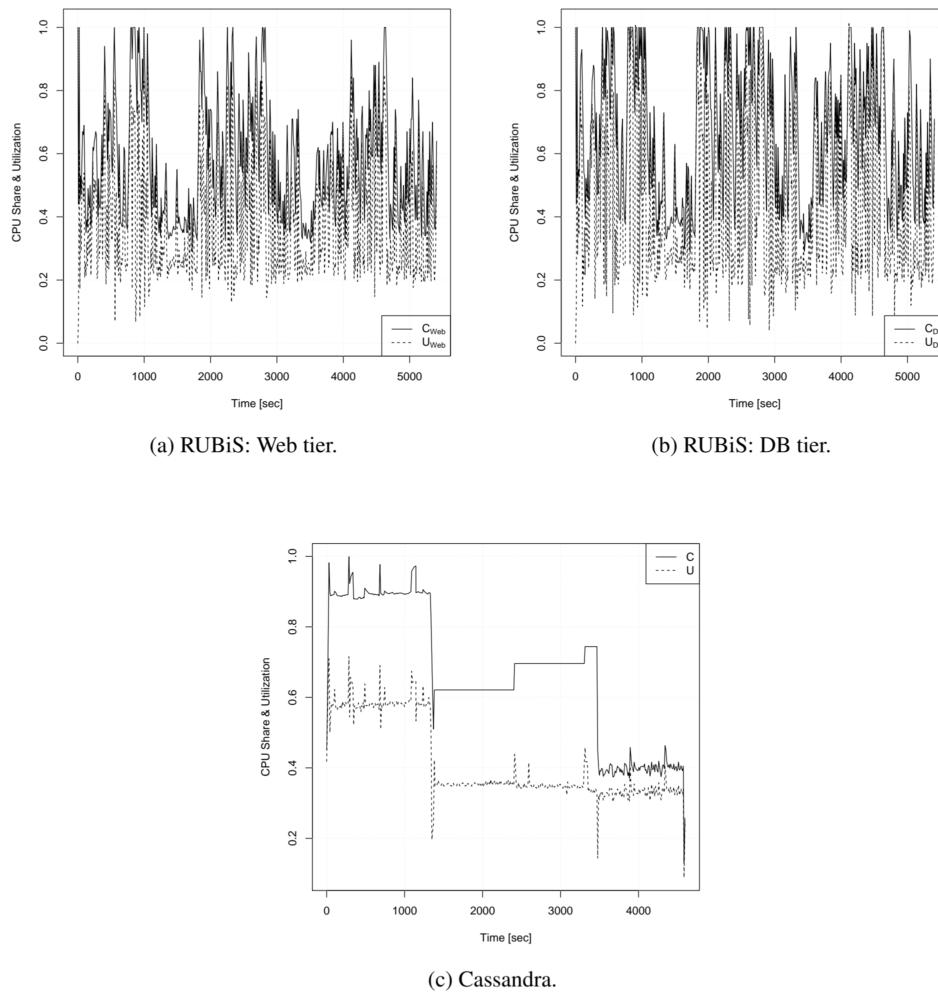
Indeed, the membership function for the "C_{res}" input variable (see Figure 5b) implies that if the residual CPU capacity lies in between 10% and 40% (i.e., it is *FINE*), then no CPU share adjustments are made if the SLO is being met (i.e., "e" is *OK*). This means that an average over-allocation of 25% (corresponding to the middle point of the triangle representing the value *FINE*) is considered to be normal by FC2Q.

In case a smaller average over-allocation is desired, the triangle corresponding to the *FINE* fuzzy value in the membership function for "C_{res}" needs to be changed (in particular, its base has to be shortened accordingly).

5.5. Discussion

The main reason of the ability of FC2Q to suitably allocate resources to application tiers stems from its ability to rapidly determine whether the SLO is being met or not, and whether too much capacity has been allocated to an application tier or not. This is achieved by the use of fuzzy control (which, as discussed in Section 6 below, unlike approaches based on linear control, is able to deal with non-linearities) and by the information used as control inputs, namely the "relative error" $e(k)$ and the "residual capacity of tier i" $C_{\text{res},i}(k)$ (which, as discussed in Section 3.1, are used both to assess

Copyright © 0000 John Wiley & Sons, Ltd.



(a) RUBiS: Web tier.

(b) RUBiS: DB tier.

(c) Cassandra.

Figure 11. Consolidation scenario – $\langle RUBiS, Cassandra\rangle$: CPU share and utilization as observed under the control of FC2Q.

the SLO achievement and to determine if the CPU capacity of each tier is either well-provisioned, over-provisioned, or under-provisioned).

In summary, from the results obtained in the experimental evaluation we can conclude that FC2Q, with respect to state-of-the-art works, provides the following benefits:

- Unlike approaches that use model-based linear control (e.g., [12, 58, 59, 60]), FC2Q is able to deal with non-linearities in the controlled system and to properly handle non-stationary workloads thanks to the use of fuzzy control (thus avoiding any linearization operation [16] that would introduce inaccuracies).
- Unlike approaches that do not consider the sign of the performance deviations (e.g., [12, 15, 19, 58, 59, 60]), FC2Q is able to distinguish between situations where the deviation of performance is due to a SLA violation or to resource over-provisioning.
- Unlike approaches that do not consider resource utilization at each application tier (e.g., [15]), FC2Q is able to take into consideration resource bottlenecks at different application tiers, thus assigning them a different resource capacity in order to meet SLAs.
- In all the experimental scenarios we consider, FC2Q outperforms the other state-of-the-art approaches, as it is able to achieve a better consolidation level without violating any

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe

application SLAs, unlike competing solutions that fail to achieve either one of these goals, or both of them.

6. RELATED WORKS

The dynamic management and provisioning of physical resources for virtualized cloud applications with SLO constraints is a topic that has been actively investigated in the past years. Existing approaches can be classified into three categories, namely *VM placement*, *VM horizontal scaling* (also known as VM provisioning) and *VM vertical scaling* (also referred to as VM sizing). However, to the best of our knowledge, only few works focus also on the maximization of the consolidation level (as we instead do).

VM placement approaches [35, 37, 61, 62] exploit live VM migration to dynamically move VMs to the set of physical machines that allow them to meet their SLAs and, possibly, to increase the consolidation level. VM placement techniques, however, are unsuitable to work at a short time scale, given the relatively high costs of VM migration both in terms of performance and energy, that can become prohibitive if migrations occur too frequently.

VM horizontal scaling approaches [63, 64, 65] focus on dynamically varying the *number* of VMs provisioned to a given application in order to provide it with more or less CPU power to match workload demand. As VM placement techniques, horizontal scaling is unsuitable to operate at short time scales, because of the large overhead of starting and stopping a VM, that must be amortized over a relatively long time scale.

Dynamic VM vertical scaling approaches (like ours) focus 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.

Utility-driven approaches [11, 66] base their decisions on the maximization of a suitable utility function that quantifies the goodness of each possible decision, and typically rely on analytical models of applications and computing resources to develop a suitable optimization problem. The main problems of these approaches are the difficulty to build accurate models of applications and systems, and the possibly high computation time required to perform optimization (that makes them suitable only to work at a long or medium time scale).

Time-series based approaches [67, 13] use historical data to build black-box models to mimic the behavior of the application, and to forecast its resource requirements, so that it can be adjusted at run time. However, to be able to produce accurate forecasts for a short time scale, these techniques usually require that specific patterns (e.g., trends and/or seasonality) or stationarity are present in the historical data at that time scale, a characteristics that the workloads we consider in this work not always exhibit. Thus, the scarcity (or even the lack) of such properties can introduce delays in the reaction time, thus making such approaches ineffective.

Machine learning approaches [14, 68], as time-series ones, use historical data too to build models able to accurately capture system behavior without any explicit performance model. However, the training process of these models tends to be very long in time before a sufficiently accurate model is learned, and this may be unacceptable for applications with strict SLO requirements.

Finally, control-theoretic approaches able to work at a short time scale (as the one proposed in this paper) have been proposed as well in the literature.

Model-based approaches relying on linear feedback control and using adaptive and optimal control to cope with time-varying workloads [12, 58, 59, 60] fail to properly handle non-stationary workloads [15, 69], as effect of the errors induced by the linearization [16] they have to perform when dealing with non-linear systems (like those addressed in this paper). Furthermore, they do not take into account the sign of the error between the observed and the reference value, and are therefore unable to differentiate a SLO violation from a resource overallocation.

In contrast, FC2Q is able to deal with non-linearities in the controlled system thanks to the use of fuzzy control, that does not require any linearization operation, and to the consideration of the sign of the error, that makes it able to differentiate between SLO violations and CPU capacity overallocation and, therefore, to make better control actions.

Copyright © 0000 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Fuzzy control has been also used in [15], that proposes the already-discussed DynaQoS, a fuzzy proportional-derivative controller. However, as discussed in previous sections, this controller is unable to react to workload changes because of the choice of using Δe as control input. Conversely, FC2Q has been shown to be able to work well under these conditions, and to outperform DynaQoS in all the experimental scenarios considered in this paper.

In [19], a neural-fuzzy model is used to describe the relationship between the application performance measure (the control output) and the resource allocations (the control inputs), and to design a model predictive controller that aims at reducing the control error by minimizing the allocated physical resources. However, the above input-output model 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. Furthermore, it needs to use linearization to design the model predictive controller, with consequent errors deriving from this operation, and does not consider the sign of the error.

In contrast, FC2Q is able to take into consideration resource bottlenecks at different application tiers (thus assigning them a different resource capacity), and to make different decisions according to the sign of the control error (thus differentiating between deviations from the SLO value due to SLO violations and ones due to SLO achievement but with over-provisioned resource).

7. CONCLUSIONS AND FUTURE WORKS

In this paper we presented FC2Q, a dynamic vertical scaling framework, based on fuzzy control, that is able to achieve the best consolidation level that can be attained on a physical server without violating the SLAs of the applications running on it. FC2Q works by continuously monitoring application performance and resource usage in order to determine whether, when, and how to adjust the amount of physical CPU capacity allocated to each application tier, so that it is able to properly cope with the time-varying and bursty workloads that characterize contemporary cloud applications.

To assess the efficacy and performance of FC2Q, we implemented it on a real testbed, that we used in an extensive experimental evaluation in various scenarios, involving three real-world cloud applications (namely, RUBiS, Olio, and Cassandra), that can be considered representative of typical cloud applications that run today on Internet data centers, exposed to time-varying and bursty workloads both in absence and in presence of resource contention. Furthermore, we compare FC2Q against two existing, state-of-the-art controllers tailored to virtualized cloud applications with SLA constraints, namely DynaQoS and AutoControl, as well as against a static approach.

Or results show that, in all the experimental scenarios we considered, FC2Q 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.

Future works are planned along two distinct directions. First, we plan to extend FC2Q to incorporate also other types of physical resources (e.g., memory and disk bandwidth) in addition to CPU. Second, we plan to investigate the use of adaptive control techniques (like the adaptive neuro-fuzzy inference framework [70]) and evaluate its possible benefits in obtaining a better consolidation level, while still achieving application SLAs.

REFERENCES

1. Amazon Web Services I. Amazon Web Services. Online: `https://aws.amazon.com` 2014.
2. Rackspace. Rackspace: The Open Cloud Company. Online: `http://www.rackspace.com` 2014.
3. Vogels W. Beyond server consolidation. *ACM Queue* Jan/Feb 2008; .
4. 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).
5. Maltz DA. Challenges in cloud scale data centers. *SIGMETRICS Performance Evaluation Review* Jun 2013; **41**(1).
6. Rolia J, Cherkasova L, Arlitt M, Andrzejak A. A capacity management service for resource pools. *Proc. of the* 5th *International Workshop on Software and Performance (WOSP)*, 2005; 229–237.
7. 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* 2013; In Press.

Copyright © 0000 John Wiley & Sons, Ltd.

8. Wang Y, Chen S, Pedram M. Service level agreement-based joint application environment assignment and resource allocation in cloud computing systems. *Proc. of the IEEE Green Technologies Conference*, 2013; 167–174.
9. Singh R, Sharma U, Cecchet E, Shenoy P. Autonomic mix-aware provisioning for non-stationary data center workloads. *Proc. of the 7th International Conference on Autonomic Computing (ICAC)*, 2010.
10. Mi N, Casale G, Cherkasova L, Smirni E. Injecting realistic burstiness to a traditional client-server benchmark. *Proc. of the 6th International Conference on Autonomic Computing (ICAC)*, 2009.
11. Menasce D, Bennani M. Autonomic virtualized environments. *Proc. of the International Conference on Autonomic and Autonomous Systems (ICAS)*, 2006.
12. Padala P, Hou KY, Shin KG, Zhu X, Uysal M, Wang Z, Singhal S, Merchant A. Automated control of multiple virtualized resources. *Proc. of the 4th ACM European Conference on Computer Systems (EuroSys)*, 2009; 13–26.
13. Shen Z, Subbiah S, Gu X, Wilkes J. CloudScale: Elastic resource scaling for multi-tenant cloud systems. *Proc. of the 2nd ACM Symposium on Cloud Computing (SoCC)*, 2011; 5:1–5:14.
14. Kundu S, Rangaswami R, Gulati A, Zhao M, Dutta K. Modeling virtualized applications using machine learning techniques. *Proc. of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2012; 3–14.
15. 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).
16. Hellerstein J, Diao Y, Parekh S, Tilbury D. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.
17. 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.
18. Kosko B. Fuzzy systems as universal approximators. *IEEE Transactions on Computers* Nov 1994; **43**(11):1329–1333.
19. Lama P, Guo Y, Zhou X. Autonomic performance and power control for co-located web applications on virtualized servers. *Proc. the IEEE/ACM 21st International Symposium on Quality of Service (IWQoS)*, 2013; 1–10.
20. 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 3rd International Cloud and Green Computing Conference (CGC)*, 2013; 159–166.
21. 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.
22. 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 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007; 205–220.
23. 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.
24. 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.
25. Lama P, Zhou X. PERFUME: Power and performance guarantee with fuzzy MIMO control in virtualized servers. *Proc. of the IEEE 19th International Workshop on Quality of Service (IWQoS)*, 2011; 1–9.
26. Xiao Z, Chen Q, Luo H. Automatic Scaling of Internet Applications for Cloud Computing Services. *IEEE Transactions on Computers* Nov 2012; In Press.
27. Amazon. Amazon EC2 Pricing. Online: `https://aws.amazon.com/ec2/pricing/` 2014.
28. Rackspace. Rackspace Cloud Servers Pricing. Online: `http://www.rackspace.com/cloud/servers/pricing/` 2014.
29. W Lloyd and S Pallickara and O David and J Lyon and M Arabi and K Rojas. Performance implications of multi-tier applications deployments on Infrastructure-as-a-Service: Towards performance modeling. *Future Generation Computer Systems* July 2013; **29**(5).
30. Nathuji R, Kansal A, Ghaffarkhah A. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. *Proceedings of 5th ACM European Conference on Computer Systems (EuroSys '10)*, ACM Press: Paris, France, 2010.
31. Kambadur M, Moseley T, Hank R, Kim M. Measuring Interference Between Live Datacenter Applicatins. *Proc. of Supercomputing '12, the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE: Salt Lake City, Utah, US, 2012.
32. Borgetto D, Casanova H, Costa GD, Pierson JM. Energy-aware service allocation. *Future Generation Computer Systems* 2012; **28**(5):769–779.
33. Gmach D, Rolia J, Cherkasova L, Kemper A. Resource pool management: Reactive versus proactive or let's be friends. *Computer Networks* Dec 2009; **53**(17):2905–2922.
34. Lama P, Zhou X. Efficient server provisioning with control for end-to-end response time guarantee on multitier clusters. *IEEE Transactions on Parallel and Distributed Systems* 2012; **23**(1):78–86.
35. Verma A, Ahuja P, Neogi A. pMapper: Power and migration cost aware application placement in virtualized systems. *Proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 2008; 243–264.
36. 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.
37. Liu H, He B. VMbuddies: Coordinating live migration of multi-tier applications in cloud environments. *IEEE Transactions on Parallel and Distributed Systems* 2014; In Press.
38. Bhadauria M, McKee S. An approach to resource-aware coscheduling for CMPs. *Proc. of the International Conference on Supercomputing (ICS)*, ACM, 2010.
39. Eyerman S, Eeckhout L. Probabilistic job symbiosis modeling for SMT processor scheduling. *ACM SIGPLAN Notices* March 2010; **45**.
40. Almeida J, Almeida V, Ardagna D, talo Cunha, Francalanci C, Trubian M. Joint admission control and resource allocation in virtualized servers. *Journal of Parallel and Distributed Computing* 2010; **70**(4):344–362.

Copyright © 0000 John Wiley & Sons, Ltd.

41. 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.
42. Chen G, Pham T. *Introduction to Fuzzy Sets, Fuzzy Logic, and Fuzzy Control Systems*. CRC Press, 2001.
43. Mamdani E. Application of fuzzy logic to approximate reasoning using linguistic synthesis. *IEEE Transactions on Computers* 1977; **26**(12):1182–1191.
44. Zadeh LA. Fuzzy sets. *Information and Control* 1965; **8**(3):338–353.
45. Chen F, Lambert D, Pinheiro JC. Incremental quantile estimation for massive tracking. *Proc. of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2000; 516–522.
46. 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.
47. 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.
48. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. *Proc. of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010; 143–154.
49. 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 19th ACM Symposyum on Operating Systems Principles (SOSP)*, 2003; 164–177.
50. Red Hat. Libvirt virtualization API. Online: `http://libvirt.org` 2014.
51. Anglano C, Canonico M, Guazzone M. Repository for the code used in our experimental evaluation. Online: `https://github.com/sguazt/ccpe2014` 2014.
52. Rada-Vilela J. fuzzylite: A fuzzy logic control library written in `C++` 2014. URL `http://www.fuzzylite.com`.
53. 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 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012; 37–48.
54. 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.
55. OW2 Consortium. RUBiS: Rice university bidding system. Online: `http://rubis.ow2.org/index.html` 2008.
56. Apache Software Foundation. Olio web 2.0 toolkit. Online: `https://incubator.apache.org/projects/olio.html` 2011.
57. Apache Software Foundation. The Cassandra project. Online: `http://cassandra.apache.org` 2014.
58. Liu X, Zhu X, Singhal S, Arlitt M. Adaptive entitlement control of resource containers on shared servers. *Proc. of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, 2005; 163–176.
59. Guazzone M, Anglano C, Canonico M. Energy-efficient resource management for cloud computing infrastructures. *Proc. of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011.
60. Wang X, Wang Y. Coordinating power control and performance management for virtualized server clusters. *IEEE Transactions on Parallel and Distributed Systems* Feb 2011; **22**(2):245–259.
61. Wood T, Shenoy P, Venkataramani A, Yousif M. Sandpiper: Black-box and gray-box resource management for virtual machines. *Computer Networks* 2009; **53**(17):2923–2938.
62. Song W, Xiao Z, Chen Q, Luo H. Adaptive resource provisioning for the cloud using online bin packing. *IEEE Transactions on Computers* 2013; In Press.
63. Tesauro G, Das N, Bennani M. A hybrid reinforcement learning approach to autonomic resource allocation. *Proc. of the 3rd International Conference on Autonomic Computing (ICAC)*, 2006.
64. Lama P, Zhou X. Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee. *ACM Transactions on Autonomous and Adaptive Systems* Jul 2013; **8**(2):9:1–9:31.
65. Kaur PD, Chana I. A resource elasticity framework for qos-aware execution of cloud applications. *Future Generation Computer Systems* 2014; **37**(0):14–25.
66. Ghanbari H, Simmons B, Litoiu M, Iszlai G. Feedback-based optimization of a private cloud. *Future Generation Computer Systems* 2012; **28**(1):104–111.
67. Meng X, Isci C, Kephart J, Zhang L, Bouillet E, Pendarakis D. Efficient resource provisioning in compute clouds via vm multiplexing. *Proc. of the 7th International Conference on Autonomic Computing (ICAC)*, 2010; 11–20.
68. Minarolli D, Freisleben B. Distributed resource allocation to virtual machines via artificial neural networks. *Proc. of the 22nd Parallel, Distributed and Network-Based Processing (PDP)*, 2014; 490–499.
69. Lama P, Zhou X. Coordinated power and performance guarantee with fuzzy MIMO control in virtualized server clusters. *IEEE Transactions on Computers* 2013; In Press.
70. Jang JS. ANFIS: Adaptive-network-based fuzzy inference system. *IEEE Transactions on Systems, Man and Cybernetics* May 1993; **23**(3):665–685.

Copyright © 0000 John Wiley & Sons, Ltd.

DOI: 10.1002/cpe