# Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints

Soodeh Farokhi*‖, Ewnetu Bayuh Lakew†‖, Cristian Klein‡, Ivona Brandic*, and Erik Elmroth†
*Faculty of Informatics, Vienna University of Technology, Austria.
†Department of Computing Science, Umeå University, Sweden.
‡SimScale GmbH, Germany.
Email: *{firstname.lastname}@tuwien.ac.at, †{ewnetu, elmroth}@cs.umu.se, ‡cklein@simscale.com.

***Abstract*—Vertical elasticity is recognized as a key enabler for efficient resource utilization of cloud infrastructure through fine-grained resource provisioning, e.g., allowing CPU cycles to be leased for as short as a few seconds. However, little research has been done to support vertical elasticity where the focus is mostly on a single resource, either CPU or memory, while an application may need arbitrary combinations of these resources at different stages of its execution. Nonetheless, the existing techniques cannot be readily used as-is without proper orchestration since they may lead to either under- or over-provisioning of resources and consequently result in undesirable behaviors such as performance disparity. The contribution of this paper is the design of an autonomic resource controller using a fuzzy control approach as a coordination technique. The novel controller dynamically adjusts the right amount of CPU and memory required to meet the performance objective of an application, namely its response time. We perform a thorough experimental evaluation using three different interactive benchmark applications, RUBiS, RUBBoS, and Olio, under workload traces generated based on open and closed system models. The results show that the coordination of memory and CPU elasticity controllers using the proposed fuzzy control provisions the right amount of resources to meet the response time target without over-committing any of the resource types. In contrast, with no coordinating between controllers, the behaviour of the system is unpredictable e.g., the application performance may be met but at the expense of over-provisioning of one of the resources, or application crashing due to severe resource shortage as a result of conflicting decisions.**

## I. Introduction

Cloud computing provides pools of resources and shares them among multiple customers' applications. Instead of committing to a fixed size of resources on a long-term basis, it offers rapid elasticity, letting customers quickly adjust their level of resource consumption as their applications' requirements change.

Elasticity, as a main selling point of cloud computing, is defined as the ability of the cloud infrastructure to rapidly decide the right amount of resources needed by each application [1]. Two types of elasticity are defined: horizontal and vertical. While horizontal elasticity allows Virtual Machines (VMs) to be acquired and released on-demand, vertical elasticity allows adjusting the resources of individual VMs to cope with runtime changes. Generally speaking, horizontal elasticity is coarse-grained, i.e., VMs with static and fixed size configurations. Vertical elasticity, on the other hand, is fine-grained: the size of the VMs can be dynamically changed to an arbitrary size for as short as a few seconds [2].

‖The first two authors have contributed equally to this paper.

Horizontal elasticity has been widely adopted by commercial clouds due to its simplicity as it does not require any extra support from the Hypervisor. However, due to the static nature and fixed VM size of the horizontal elasticity, applications cannot be provisioned with arbitrary configurations of resources based on their demands. This leads to inefficient resource utilization as well as Service Level Agreement (SLA) violations since the demand cannot always exactly fits the size of the VM. To efficiently utilize resources and avoid SLA violations, horizontal elasticity should be augmented with fined-grained resource allocations where VM sizes can be dynamically adjusted to an arbitrary value according to runtime demands. Nevertheless, in the last decade, most research have focused on horizontal, while only few research efforts have addressed vertical elasticity [3] due to lack of support from Hypervisors. However, vertical scaling of resources have recently started to be supported by Hypervisors such as Xen [2] and KVM [3].

The few research efforts made on vertical elasticity focus either on CPU vertical scaling [2], [4]–[6], or memory vertical scaling [3], [7]–[10] but not both. The underlying assumption made in these works is that the application is either CPU-intensive or memory-intensive, moreover they do not consider application performance (e.g., response time) at all [11], [12]. However, during the application life-time, it can show both characteristics depending on the nature of workload (Section II). For example, a chat application may require techniques such as long-polling to immediate notify the user when a new message has arrived. Long-polling essentially delays the HTTP reply, until there is an event to report, which in turn increases the number of connections on the server, hence increasing its memory requirements, without significantly increasing CPU utilization. On the contrary, the chat application might include a "search in chat history" functionality, which is CPU intensive. Obviously, the application needs to be scaled both CPU- and memory-wise. However, existing techniques cannot readily be used as-is for scaling both memory and CPU at the same time. This is because uncoordinated control actions by different controllers may lead to suboptimal or inconsistent resource allocations which may in turn result in SLA violations [13]. The goal of this work is to design an autonomic controller that dynamically performs coordinated adjustments of CPU and memory allocations in order to meet the performance requirement of a cloud-based application.

In order to address the deficiencies of the existing approaches, and to support the vertical scaling for both CPU and memory while meeting the target response time (RT), in this

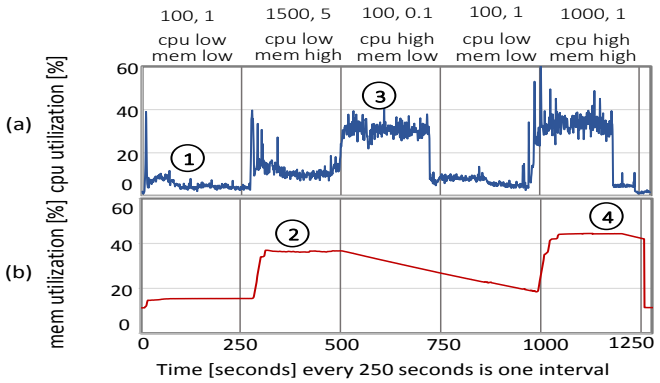



Figure. 1: CPU and memory usage of RUBiS with different workload patterns.

paper we design an autonomic resource controller that allocates the right amount of both resources. The autonomic resource controller (Section III) is composed of three sub-controllers. The first is a *fuzzy controller* (Section IV) based on fuzzy logic to infer the degree of contributions of both CPU and memory to applications' performance change. Using the resource demand indicators generated from *fuzzy controller*, the other two controllers, i.e., *cpu controller* and *memory controller*, determine the amount of CPU and memory needed to meet the application performance target. In general, *fuzzy controller* acts as a coordinator by looking at resource application's demand indicators such as the average of CPU and memory utilization and application's performance so that the control actions of the vertical controllers compliment each other to fulfill the application need. More specifically, we make the following **contributions**:

(i) *Autonomic provisioning of resources:* We propose autonomic controllers that provision the right amount of resources required to meet the target application RT. Given the varying and unpredictable nature of workloads and the infrastructure, the controllers continuously re-adjusts the right amount of resources to meet the demand;

(ii) *Fuzzy controller to coordinate different elasticity controllers:* We propose a fuzzy control approach to coordinate CPU and memory controllers when provisioning CPU and memory respectively for the same application;

(iii) we perform thorough experimental evaluations on the proposed approach using three different benchmark applications. We validate our approach using workload traces generated based on open and closed system models.

The results show that without coordinating auto-scaling controllers, the behaviour of the system is unpredictable. For instance, the application performance may be met but at the expense of over-provision one of the resources or application crashing due to severe resource shortage as a result of conflicting decisions. Moreover, they show that with careful coordination of auto-scaling controllers application performance is met with optimal amount of resources (i.e., achieving high resource utilization) preventing both resource over- and under-provisioning (Section V).

## II. Motivating Scenario

The pervasive and popular architectural patterns for a cloud-based application is the 3-tier pattern [14]. It comprises presentation tier (representing user interface), business logic (BL) tier (featuring the main business logic), and data storage (DS) tier (managing the persistent data). Multiple tiers of a cloud-based application may be involved for processing user requests, thus, requiring different resources such as CPU cores, memory, or both. Therefore, vertical auto-scaling may be required for different resources of one or more tier(s). In this work, we focus on vertical scaling of the BL tier.

To show the BL tier may require different resource configurations, we conducted an experiment on a benchmark application (RUBiS [15]) by injecting variable workloads, which induce the intended behavior, at different time intervals. We deployed the BL and DS tiers of RUBiS on different VMs and over-provisioned both VMs (the VM hosting BL tier: 8 CPU cores and 4 GB memory and the VM hosting DS tier: 6 CPU cores and 10 GB memory). Then, by configuring a workload generator tool, `httpmon`[1], we defined variable workload dynamics which stress BL tier for either CPU, memory, or both resources during its life span. Fig. 1(a) and (b), respectively, depict the experimental results regarding the CPU utilization and memory utilization of the VM hosting `Apache Web Server` as the representative of the BL tier. We set different configuration in `httpmon` to emulate different combinations of CPU or memory demands using a closed-system model [16]. To this aim, we vary the number of concurrent users and *thinktime* at each interval (i.e, every 250 seconds) as shown as two values at the top of Fig. 1, respectively. High *thinktime* resembles the case where many clients are doing long-polling, while high concurrency resembles the case where many clients are intensively interacting with the BL tier. Therefore, changing the number of concurrent users and *thinktime* is a way to emulate variable CPU and memory requirements for an application. For example, to emulate an application with high CPU and low memory requirements, one can set both the number of concurrent users and the *thinktime* to relatively low values (e.g., 1st interval). On the other hand, to induce high memory and low CPU characteristics, one can set high values for both the number of concurrent users and the *thinktime* (e.g., 2nd interval).

As observed in Fig. 1, there are some intervals where RUBiS needs less of both resources (1th & 4th intervals), more memory and less CPU (2nd interval), more CPU and low memory (3rd interval) as well as more CPU and memory (5th interval). The results clearly show that the application may need an arbitrary combination of these resources during its execution. Thus, designing an autonomic controller that considers multiple resources, in this case CPU and memory to meet application performance targets is a challenging task due to unpredictable workloads and the arbitrary nature of applications' resource requirements.

The more memory the VM hosting the BL tier has, the more processes Web server (e.g. `Apache`) can allocate and use; which directly translates into the amount of concurrent requests/clients that it can serve and consequently faster response time [17]. It is worth mentioning that in general, allocating more memory to the VM hosting the DS tier will also lead to the ability of caching more data into memory, therefore, increasing the probability of a cache hit and consequent enhancement of the performance. However, due to the lack of research on memory elasticity of the BL tier, we chose

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

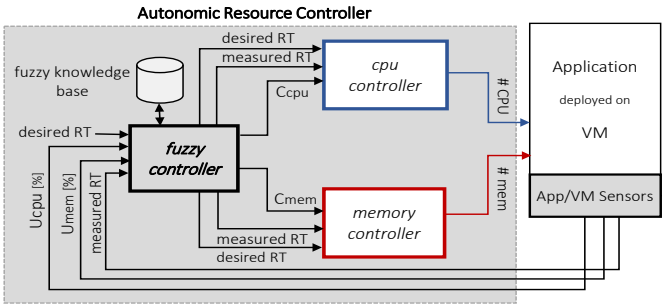



Figure. 2: The architecture of the autonomic resource controller. Modules in "grey" are the contribution of this work.

to primarily concentrate on this tier. Notice that the proposed solution targets applications that can benefit from live memory and CPU elasticity at runtime, i.e. application with dynamic memory and CPU requirements.

## III. Autonomic Resource Controller

In this section, we first give an overview of the proposed approach for autonomic resource control, then we briefly explain the two vertical controllers which are used in this work.

### A. Overview

We consider a cloud infrastructure that hosts interactive applications, each with variable workload dynamics. Each service has an SLA that stipulates a target value expressed as *mean response time*. The goal is to continuously adjust the allocated resources of applications without human intervention, so as to drive applications' performance toward their targets. Specifically, the desired autonomic resource controller should be capable of allocating just the right amount of resources for each application at the right time in order to meet its respective performance target, avoiding both resource under- and over-provisioning.

Fig. 2 shows the architecture of the proposed autonomic resource controller. It loosely follows a Monitor, Analysis, Planning, and Execution (MAPE) loop based on self-adaptive software terminology [18]. Monitoring gathers information such as the observed RT, average CPU and memory utilization from the hosted services at each interval. During analysis, the resources required by an application to meet its performance target are computed using the controllers in two steps.

In the first step, *fuzzy controller* infers the extent of contributions of both CPU and memory to applications' performance change. More specifically, it generates a value $\in$ [-1,+1] indicating the degree of severity that CPU and memory has on the performance of the application. A value closer to -1 indicates resource over-provisioning case, while a value closer to 1 shows under-provisioning situation.

In the second step, using the respective values generated by *fuzzy controller*, the CPU and memory controllers, respectively, determine the number of CPU core(s) and the amount of memory that should be allocated to the application using the application RT as a decision making criterion. Previous monitoring data is used to fit the model parameters, which are similar to the knowledge component of autonomic controllers, shown in Fig. 2. Finally, during the planning and execution phase, Hypervisor is configured to enforce the computed resources. A high level function of each component depicted in Fig. 2 is described as follows:

- *fuzzy controller*. It determines the coefficient with which CPU, memory, or both are the reason for the application performance change. Based on the fuzzy rules specified in the engine, it reasons about the coefficient values using the current state of the system. The output of *fuzzy controller* consists of two coefficients values, one is corresponding to CPU and the other one is related to memory, which indicates either to increase or decrease CPU, memory, or both (See Section IV).
- *cpu controller*. An adaptive controller that dynamically adjusts the CPU capacity that should be allocated to an application based on the values of measured and desired RTs as well as the CPU coefficient ($C_{cpu}$) received from *fuzzy controller*. It allocates the right amount of CPU cores for the applications which guarantees its respective performance target (See Section III-B).
- *memory controller*. Similar to *cpu controller*, *memory controller* is an adaptive controller that dynamically tunes the amount of memory required for each application using the measured and desired RT values as well as the value of the memory coefficient ($C_{mem}$) given by *fuzzy controller* (See Section III-C).
- *sensor*. This component gathers the application and VM level performance information such as *mean response time*, average CPU utilization, and average memory utilization from the application and the allocated VMs, periodically. We refer to this period as the control interval. These monitoring values are used as a feedback and decision making criteria in *fuzzy controller* for the next control interval.

While *cpu controller* and *memory controller* will be briefly introduced in this section, *fuzzy controller* as the main part of the contribution of this paper, will be discussed in details under Section IV. Note that the goal of both vertical controllers is to allocate the right amount of the resources that they control in order to meet the target response time.

### B. cpu controller

*cpu controller* is based on the work presented in [2]. The inverse relationship between *mean response time* $rt_i$ of an application and the number of CPU cores $cpu_i$ allocated to it at each control interval is modeled as:

$$rt_i = \beta / cpu_i, \tag{1}$$

where $\beta$ is a model parameter. The parameter $\beta$ can be estimated online from the past measurements of mean response time and allocated CPU cores, thus compensating dynamically for small non-linearities in the real system. However, to reduce the impact of measurement noise, we use a recursive least square (RLS) filter [19]. In essence, such a filter takes past estimation of $\beta$ and the current product $rt_i * cpu_i$ to output a new value that minimizes the least-squares error. A *forgetting factor* allows to trade the influence of old values for up-to-date measurements. In our experiments, we use a forgetting

factor of $0.45$. Note that in our experiments, based on the available resources, we define a boundary for the minimum and the maximum amount of allocated CPU.

### C. memory controller

*Memory controller* is based on an adaptive version of the work presented in [10], which follows a control synthesis technique. The control formulation, as originally devised in [20], is presented in Eq. (2). The control output at each iteration $ctl_i$ is calculated based on its previous value $ctl_{i-1}$ and a coefficient of the control error ($e_i = \tilde{rt} - rt_i$). The coefficient is based on the value of $\alpha$, a model parameter, and *pole*. The parameter $\alpha$ is estimated online based on the effect of the control output $ctl$ on the measured output $rt$. In this work, we apply the linear regression technique to capture this relationship at each control interval.

$$ctl_i \;=\; ctl_{i-1} \;-\; \frac{1-pole}{\alpha} \;\cdot\; e_i \tag{2}$$

The choice of *pole* influences the stability of the controlled system, and determines how fast the system approaches to its equilibrium. The stability of the controller is ensured as long as $0 \leq pole < 1$ [20]. The value of $pole$ trades responsiveness—how fast the controller reacts— and robustness in the face of noise. Based on analysing various results, we empirically chose 0.9 as the bset value of *pole* in our experiments. At each control interval, the controller's output $ctl \in [0, 1]$ is mapped to a memory size $mem_i \in [mem_{min}, mem_{max}]$ using Eq. (3).

$$mem_i = ctl_i \cdot (mem_{max} - mem_{min}) + mem_{min} \tag{3}$$

where $mem_{min}$ and $mem_{max}$ are the minimum and maximum amount of VM memory sizes expressed by the number of memory units[2] $mem_{unit}$. The computed memory size $mem_i$ at each control interval is used to adjust the size of VM memory, and then the influence of this memory change is reflected on the measured RT of the current iteration.

## IV. Fuzzy Controller Design

Due to the non-deterministic behavior of software systems, it is almost impossible to know with a high degree of confidence, the extent of contributions of different resources to performance degradation of a software application and how much of each resource should be provisioned to alleviate the performance problem. Moreover, the measured data used as decision making criteria in the process of resource allocation such as RT, CPU and memory utilization may include sensory noise. If an autonomic resource controller does not pay attention to such uncertainties, it may cause the oscillations in resources allocations [21], [22]. To address these issues, we propose fuzzy control as it provides a means to reason about uncertainties using highly expressive languages where the treatment of uncertainty and approximate reasoning is performed in a natural and efficient way. With this in mind, in this section, we explain the design of the core part of the proposed autonomic resource controller, *fuzzy controller*. We explain how we developed the fuzzy logic system (FLS) which is responsible for coordination and reasoning of the autonomic resource controller.

One of the most well-known applications of fuzzy logic is Fuzzy control, in which the controller decides based on the defined fuzzy rules. A typical fuzzy *IF-THEN* rule *R* is expressed as:

$$\begin{aligned} R: \quad \text{IF} \quad & \underbrace{x_1 \text{ is } F_1 \ldots \text{ and } x_p \text{ is } F_p}_{\text{antecedent: input variables and fuzzy linguistic terms}} \\ \text{THEN} \quad & \underbrace{y_1 \text{ is } G_l \ldots \text{ and } y_q \text{ is } G_q}_{\text{consequent: output control variable}} \end{aligned} \tag{4}$$

where antecedent is compound of a number of input variables, and the consequent is composed of a number of output control variables. In the case where there are multiple input and output variables, similar to our case, the system is called *multi-input-multi-output* (MIMO) fuzzy system.

**Elasticity Reasoning using FLS**. A fuzzy knowledge-base has the information of how best scale the target system in terms of a set of linguistic rules (i.e., rule (4)). In our FLS, average RT, average CPU utilization ($U_{cpu}$), and average memory utilization ($U_{mem}$) are the input variables, while CPU coefficient ($C_{cpu}$) and memory coefficient ($C_{mem}$) are the output control variables. Therefore, the fuzzy system used in this paper is a MIMO FLS. In this work, the linguistic terms representing the values of the input variables are divided into three levels. For RT they are: slow (S), medium (M), and fast (F). Similarly, for $U_{cpu}$ and $U_{mem}$ they are: low (L), medium (M), and high (H). As mentioned, the designed FLS consists of two output control variables, $C_{cpu}$ and $C_{mem}$. These values indicate two numbers $\in [-1, 1]$ as the degree of severity that CPU and memory effect on the application performance change at each control interval.

**Extracting Fuzzy Rules**. In general for an FLS there are two ways to design a fuzzy knowledge-base including fuzzy rules and membership functions (MFs): collecting data from the system behavior or using human experience. In our work, we first applied the later by following the approach of Jamshidi et al. [21] to extract the fuzzy rules from the experts and identify the MFs and then we empirically update these values by carefully monitoring and analysing the behavior of the controller. To design the fuzzy rules, we collected the required data by performing a data collection among 7 experts who had deep knowledge on cloud resource allocation and performance modeling. We prepared several questions to extract the required knowledge such as the following sample question:

$$\begin{aligned} R^{25}: \quad \text{IF} \quad & \underbrace{RT \text{ is } S \text{ and } U_{cpu} \text{ is } H \text{ and } U_{mem} \text{ is } L}_{\text{antecedent}} \\ \text{THEN} \quad & \underbrace{C_{cpu} \text{ is +1 and } C_{mem} \text{ is -0.3}}_{\text{consequent}} \end{aligned} \tag{5}$$

The experts were asked to determine the consequent (output control variables) using a number $\in [-1,1]$ for each fuzzy rule. We initially used the mean values from the experts' responses for our experiments, and then we empirically tuned some of the rules taken from experts based on analysing the behavior of *fuzzy controller*. The Final fuzzy rules which are used in our experiments (Section V) are presented in Table I.

**Constructing Membership Functions**. A MF defines the degree of truth with a value between 0 and 1. In an FLS, there is a MF for each linguistic term used for defining each input variable. In our FLS, there are three linguistic terms for each input variable, so in total we need to define 9 MFs. We use both trapezoidal and triangular MFs, as shown in Fig. 3. We used trapezoidal MFs to represent "Low" ("Fast"), and

[2]memory unit is a discrete block of memory, e.g 64MB in this work.

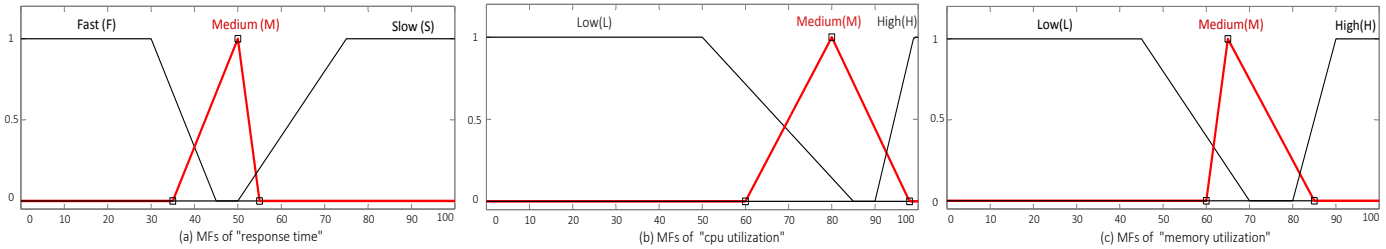



Figure. 3: Membership functions for each linguistic term of (a) response time, (b) CPU utilization, and (c) memory utilization.

Table. I: The defined fuzzy rules.

| Rule (R) | Antecedents | | | Consequent | |
|---|---|---|---|---|---|
| | RT | $U_{cpu}$ | $U_{mem}$ | $C_{cpu}$ | $C_{mem}$ |
| 1 | Fast | Low | Low | -1.0 | -1.0 |
| 2 | Fast | Low | Medium | -1.0 | -0.6 |
| 3 | Fast | Low | High | -1.0 | 0.0 |
| 4 | Fast | Medium | Low | -0.9 | -0.9 |
| 5 | Fast | Medium | Medium | -0.5 | -0.7 |
| 6 | Fast | Medium | High | -1.0 | 0.0 |
| 7 | Fast | High | Low | 0.1 | -1.0 |
| 8 | Fast | High | Medium | 0.1 | -0.7 |
| 9 | Fast | High | High | 0.1 | 0.1 |
| 10 | Medium | Low | Low | -0.9 | -0.7 |
| 11 | Medium | Low | Medium | -0.8 | -0.5 |
| 12 | Medium | Low | High | -0.8 | 0.5 |
| 13 | Medium | Medium | Low | -0.6 | -0.6 |
| 14 | Medium | Medium | Medium | -0.9 | -0.5 |
| 15 | Medium | Medium | High | -0.9 | 0.5 |
| 16 | Medium | High | Low | 0.5 | -0.8 |
| 17 | Medium | High | Medium | 0.4 | -0.5 |
| 18 | Medium | High | High | 0.5 | 0.5 |
| 19 | Slow | Low | Low | -1.0 | -0.3 |
| 20 | Slow | Low | Medium | -1.0 | 0.7 |
| 21 | Slow | Low | High | -1.0 | 1.0 |
| 22 | Slow | Medium | Low | 0.6 | -0.2 |
| 23 | Slow | Medium | Medium | 0.5 | 0.5 |
| 24 | Slow | Medium | High | 0.4 | 1.0 |
| 25 | Slow | High | Low | 1.0 | -0.3 |
| 26 | Slow | High | Medium | 1.0 | 0.5 |
| 27 | Slow | High | High | 1.0 | 1.0 |

"High" ("Slow"), and triangular MFs to represent "Medium". Similar to the fuzzy rules, we initially asked the experts to locate an interval $\subseteq$ [0,100] for each linguistic term used as input variable, and then we tuned the extracted intervals empirically. The MFs are presented in Fig. 3 for each input variable. While CPU and memory utilization do not need further normalization since the utilization for each resource is expressed in percentage, RT value should be mapped to a value $\in [0, 100]$. In order to do so, we normalize the measured response time with respect to a reference value as a coefficient of the target RT. Measured RT values closer to this reference which are further up away from the target value are set to a value close to 100. On the contrary, the RT values further down are set to a value closer to 0 depending on the magnitude of the value relative to the reference.

**Fuzzy Controller**. Having designed the FLS with the MFs and the set of fuzzy rules, the controller can then perform the elasticity reasoning. *Fuzzy controller* works based on the following steps: (i) the measured values of input variables (i.e., RT, $U_{cpu}$, and $U_{mem}$) are first fuzzified using the defined MFs (shown in Fig. 3); (ii) the FLS inferences and reasons according to the given fuzzified input variables using the designed fuzzy rules (Table I) to produce the outputs consisting $C_{cpu}$ and $C_{mem}$; (iii) the output control variables, $C_{cpu}$ and $C_{mem}$, are fed into *cpu controller* and *memory controller*, respectively, to compute CPU $cpu_i$ and memory $mem_i$ for the next control interval. Fig. 4 (a) and (b) show the outputs of *fuzzy controller* ($C_{cpu}$ and $C_{mem}$ $\in$ [-1,1]) results in a hyper-surface corresponding to all possible normalized values

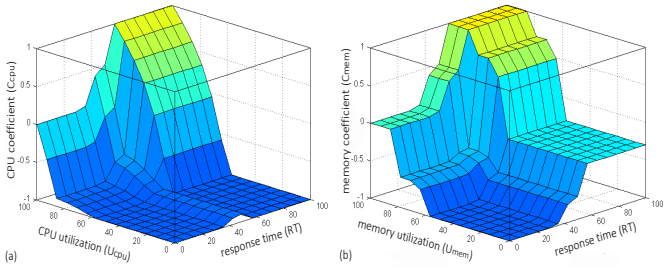



Figure. 4: Output of control variables of *fuzzy controller* in accordance with the input variables, (a) CPU coefficient, (b) memory coefficient. Notice that 3 inputs were modeled in two separated diagrams with 2 inputs for the sake of visibility.

of the input variables (RT, $U_{cpu}$, and $U_{mem}$ $\in$ [0,100]). These diagrams reveal a more conservative behavior of *fuzzy controller* for controlling the memory allocation in comparison with CPU due to the fact that applications may crash as a result of memory shortage.

**Coordination of Controllers Using Fuzzy Control.** CPU core(s) and memory sizes are allocated to each VM with seamless coordination of the two controllers. *Fuzzy controller* generates coefficients that signify the extent to which each resource is needed by the application. Using CPU coefficient $C_{cpu}$ and memory coefficient $C_{mem}$, the other two controllers determine the actual amount of each resource that should be allocated to meet the application's performance target. Then *cpu controller* and *memory controller* first compute the values of $cpu_i$ and $mem_i$ based on Eq. (1), and Eqs. (2) and (3), respectively. Finally, both controllers apply the coefficient values to calculate the respective resource using Eqs. (6) and (7). In these equations, $mem_i$ and $cpu_i$ are the outputs of *memory controller* and *cpu controller*, respectively, while $CPU_i$, and $MEM_i$ are the final amount of resources that should be allocated at control interval $i$. Indeed, *fuzzy controller* mitigates the change on the number of CPU cores or the amount of memory, as the direct outputs of *cpu controller* and *memory controller*, at the current interval relative to the previous one by looking at the utilization of both resources as well as the measured RT at the same time.

$$CPU_i = CPU_{i-1} + C_{cpu} \cdot |cpu_i - CPU_{i-1}| \quad (6)$$

$$MEM_i = MEM_{i-1} + C_{mem} \cdot |mem_i - MEM_{i-1}| \quad (7)$$

To clarify the reasoning behind values of $C_{cpu}$ or $C_{mem}$, a brief highlight is provided. Zero value for $C_{cpu}$ or $C_{mem}$ implies that the performance target is met, thus, there is no need to change resources and the actions made by *cpu controller* and *memory controller* are bypassed. A value of 1 or -1 for $C_{cpu}$ or $C_{mem}$ indicates that the values computed by these two controllers should be fully allocated to the VM hosting

the application. However, when the value of $-1 < C_{cpu} < 1$ or $-1 < C_{mem} < 1$, the actual CPU core(s) or memory size allocated to the VM is proportional to the respective coefficient values produced by *fuzzy controller* instead of allocating the amount of resources computed by these controllers directly. In general, a positive coefficient value indicates the need for additional resources while a negative coefficient value indicates that the need to reduce resources.

## V. Performance Evaluation

In this section, we present the experimental performance evaluation of the proposed autonomic resource controller based on a *fuzzy controller (FC)* as compared to a *non-fuzzy controller (NFC)*. The NFC is based on by simply running the previous independent work for *cpu controller* [2] and *memory controller* [10] in parallel without having any synchronization between the two controllers during the resource provisioning decision. Indeed, the outputs of these two controllers will be directly applied on the VM without any influence of *fuzzy controller*'s outputs, i.e., coefficient values. In this case, the inputs for both *cpu controller* and *memory controller* are only the desired and measured RT. In NFC evaluation scenario, it has been assumed that an application is either CPU- or memory-intensive, so there is no need for coordination between controllers which control these resources separately. The evaluation goal is to show which approach, FC or NFC, meet the desired RT by predicting the right amount of resources and avoiding under- or over-provisioning. A controller is said to be better if the desired RT is met without over-provisioning any of the resources. In what follows, we first describe the experimental setup and then, we report and discuss the results.

### A. Experimental Setup

The experiments were conducted on a Physical Machine (PM) equipped with 32 cores[3] and 56 GB of memory. To emulate a typical cloud environment and easily perform vertical elasticity, we used Xen Hypervisor [23]. Each benchmark application, as shown in Fig. 5, was deployed on two separate VMs. $VM_1$ runs a web server, `Apache 2.0` with PHP enabled, and $VM_2$ runs the application database, `MySQL`. To emulate long connections that induce memory-intensive behaviour on $VM_1$, as would be the case with techniques such as long-polling, we set *keep alive timeout* to 10 seconds. To avoid $VM_2$ being a bottleneck, we provisioned sufficient memory and CPU cores for that during the experiments. Besides, we set our experimental setup in a way that there is no memory consumption limit for `Apache` running on $VM_1$. We used `Apache MPM prefork` module, which is thread safe and therefore suitable to be used with PHP applications. We set parameters regarding `Apache` processes, for example `MaxClients` and `ServerLimit`, to relatively high values, i.e., 2000 in our experiments. This value is well above the number of concurrent requests that `Apache` has to deal with during any experiments in our work.

Note that, in general for using *memory controller*, one should configure the Web server in a way that it would consume all memory it needs to reach peak performance, and let the controller find the right amount of memory and adjust it at each control interval in accordance to the workloads and measured RT. In other words, it is the controller's job

[3]Two AMD Opteron™6272 processors, 2100 MHz, 16 cores each.

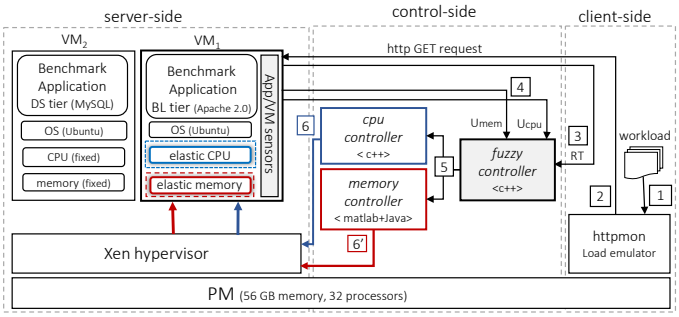



Figure. 5: Experimental setup for FC evaluation scenario. Numbers indicate the experimental process.

to provision the right amount of memory to the VM hosting the Web server in order to meet the performance constraints. This way the released memory can be used by other VMs or application which are at the same PM.

**Benchmark Applications**. To test the applicability of our contribution with a wide range of interactive applications, we performed experiments using three benchmark applications: RUBiS, RUBBoS [24], and Olio [25]. These applications are widely-used cloud benchmarks (see [26]–[28]) and represent an eBay-like e-commerce application, a Slashdot-like bulletin board, and an Amazon-like book store, respectively.

**Workloads**. Experiments were performed using different workloads to characterize the controller's responses to performance changes. We evaluated the controller using workload generated based on the open and closed system models [16]. The generated workloads gave us freedom of evaluating parameters. For instance, to increase the number of requests by five-folds or ten-folds to understand the behavior of our solution; to induce memory and/or CPU intensive behavior by varying different parameters such as *thinktime*, and the number of concurrent users accessing the systems. To emulate the users accessing the applications, under the generated workload, we used our custom `httpmon` workload generator tool, which supports both open and closed system model client behaviors. By using `httpmon`, a low *thinktime* would induce high CPU utilization. On the contrary, a long *thinktime* with a high number of concurrent users would induce a high memory utilization while keeping the CPU utilization low. We also kept the number of requests constant for some time to study the behavior of the models under both the steady- and transient-states. We test the controller under more extreme scenarios than can be found in real world traces to stress-test the resulting system. For open clients, we changed the arrival rate and inter-arrival time during the course of the experiments as required to stress the system. For the closed model, *thinktime* of each client as well as the number of concurrent users were varied. The change in arrival rate or number of users was made instantly. This made it possible to meaningfully compare the system's behavior under the two client models.

**Metrics**. The RT of a request is defined as the time elapsed from sending the first byte of the request to receiving the last byte of the reply. In this work, we are mostly interested in the average RT over 20 seconds (4 control intervals), which is a long enough period to filter measurement noise, but short enough to highlight the transient behavior of an application.

**Experiment Process.** As shown in Fig. 5, the experiment starts with feeding the workload traces into `httpmon` (1),

and based on the workload at each control interval, `httpmon` emulates a specific number of concurrent users to send `GET` requests to application under test (2). In each control interval, 5 seconds in our experiments, the application sensor observes the average RT (3), while the VM sensor measures the average of CPU and memory utilization (4). Both sensors send their monitored information via `TCP/IP` connection to the controller. Depending on the evaluation setup, either FC or NFC are use and the amount of resources required to meet the demand is computed. In the case of FC, *fuzzy controller*, implemented using *fuzzylite* library [29], computes the corresponding coefficients for CPU $C_{cpu}$ and memory $C_{mem}$, and the result is fed to the respective controllers (5). Then the CPU and memory controllers compute the amount of the respective resource. Finally, each controller invokes the corresponding actuator, i.e., the Xen API for CPU and memory allocation, to either increase or decrease the allocated memory or CPU of $VM_1$ at runtime (6, 6').

### B. Time-Series Analysis

To evaluate the controllers, we injected a variable load, so as to test how each controller reacts during sudden workload spikes (i.e., under extreme conditions) under both open and closed system models. Furthermore, the target RTs used in the experiments were varied from relatively high to small target values in order to assess the controllers' behavior under different scenarios.

The plots in this section are structured as follows. Each figure shows the results of a single experiment. Note that we performed a number of experiments and found similar patterns in the results and then presents one of them. The bottom x-axis represents the time elapsed since the start of the experiment. The upper graph in each figure plots *mean response times*, while the lower graphs plot the number of CPU cores and the amount of memory required in GB as were computed by the respective controllers and allocated to the VM hosting the BL tier (i.e., $VM_1$) of the application under test over the next 5 seconds as the control interval.

Figs. 6(a) to 6(f) show the results for FC and NFC scenarios with different target RTs under open and closed system models for Olio application. In general, the measured RTs remain stable and close to the target values under both system models. Moreover, the RTs converge to the target values immediately, mostly without being noticed, after detecting a sudden increase or decrease in workload for FC and NFC scenarios. However, NFC most of the time over-provisions both memory and CPU. The reason for this is due to asynchronous decision making by the CPU and memory controllers. As a result, each controller assumes that the performance degradation is due to the lack of resource which is controlled by itself. Since *cpu controller* is more reactive than *memory controller* due to the different nature of the controlled resources, it can adapt the allocated CPU more quickly with the right number of cores. While *memory controller* is more conservative for memory adjustment and it remains in over-provisioning state because it needs to consider the performance stability of its decision. In general, FC allocates the right amount of both resources without over- or under-provisioning any of the resources.

The other important point to note is that both FC and NFC properly detect and adapt to the CPU capacity required to meet the target RTs for both open and closed model systems. Indeed, as it can be observed from the plots presented in Fig. 6, the open system model requires more capacity compared to the closed system model for similar configurations. This is because the closed system model waits for the *thinktime* after getting response from the system which reduces the intensity of the workload. However, memory is over-provisioned under NFC for the reason explained above. On the contrary, FC allocates the right amount of memory required to meet the target RT for both open and closed system models due to the coordination of the memory and CPU controllers. Besides, close observation of the results reveals that the memory allocated using FC is slightly higher under open system model compared to closed system model for similar configurations. This is because that the number of `Apache` processes created are slightly higher under open system model than closed system model. Moreover, there is a slight increase in memory usage as the number of users or arrival rates increase because of the equivalent number `Apache` processes created. However, memory is not immediately released unlike CPU cores as the number of users or arrival rates decrease. This is because the idle `Apache` processes are not garbage collected immediately.

We also run the experiments with RUBiS and RUBBoS applications to observe the behaviour of the proposed coordination approach with applications which have different types of resource needs. However, due to the lack of space, we only present time series plots for target RT of $0.5$sec and $1.0$sec for RUBiS and RUBBoS, respectively. As can be observed from Figs. 7 and 8, while NFC does not behave well as it over-provisions memory for RUBiS application, FC remains stable since provisioning of the resources is synchronized. The over-provisioning of both CPU and memory was lower in case on RUBBoS.

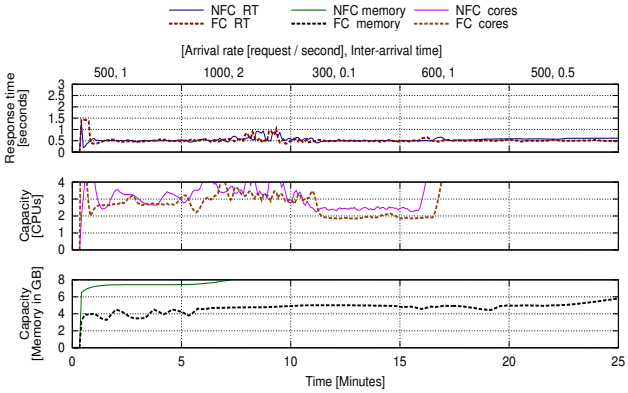
In general, FC remains stable both in terms of achieving performance targets and avoiding resource over- and under-provisioning irrespective of the target values under both system models. On the other hand, most of the time, NFC makes inconsistent decisions. In general, it is only *cpu controller* that does its job under NFC while *memory controller* is usually over-provisioning the memory because of its slow reaction due to the fact that memory is released or reclaimed by the application slowly. Moreover, there were some instances where the experiments were not able to be completed under NFC as a result of application crashing due to low memory allocation (under-provisioning). Thus, the behavior of NFC is non-deterministic from one run to the other for the same workload pattern and with the same configurations.

### C. Aggregate Analysis

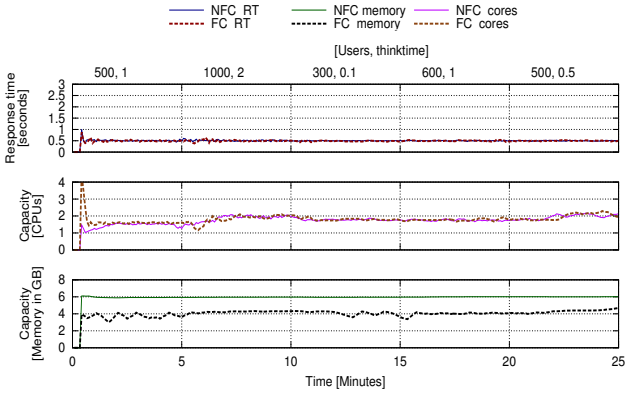
To assess the aggregate behaviors of FC in comparison with NFC over the course of the experiments, we report the mean values for the CPU and memory allocations along with the values for the mean and standard deviation (SD) of RT. Moreover, we also present two control theoretic metrics which measure the total error observed during the life span of the system under test. These metrics are Integral of Squared Error (ISE) and Integral of the Absolute Error (IAE) which are computed as shown in Eqs. (8) and (9), where $e(t) = \tilde{rt} - rt(t)$.

$$ISE = \sum \left(e(t)\right)^2 \tag{8}$$

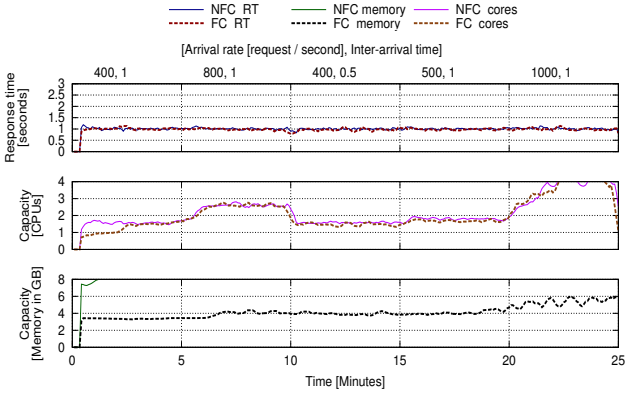
$$IAE = \sum |e(t)| \tag{9}$$



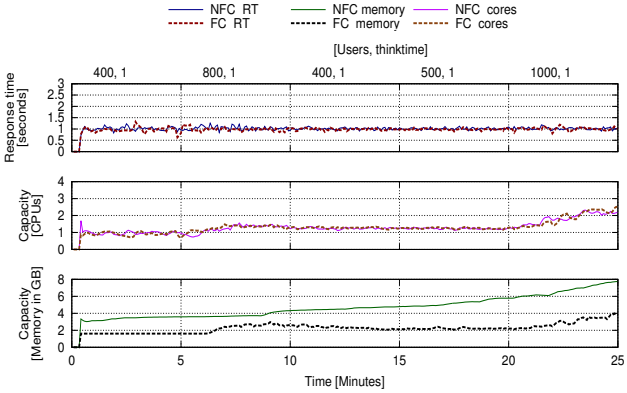

((a)) open system model, 0.5sec target



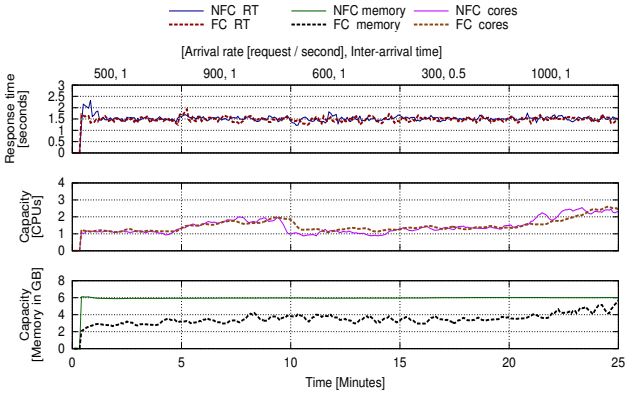

((b)) closed system model, 0.5sec target



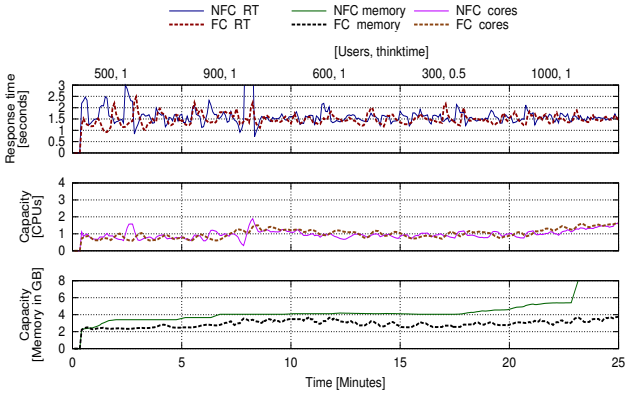

((c)) open system model, 1.0sec target




((d)) closed system model,1.0sec target




((e)) open system model, 1.5sec target

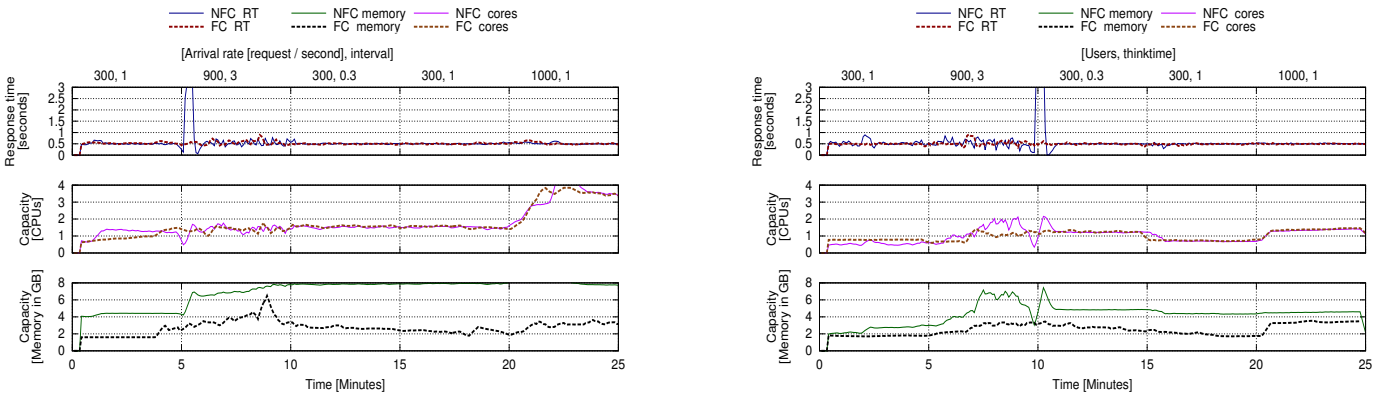



((f)) closed system model, 1.5sec target

Figure. 6: Olio– under open and closed system models with with 0.5sec, 1.0sec and 1.5sec target response times.

Table. II: Aggregated results under the two system models for Olio, RUBiS and RUBBoS with various target RTs.

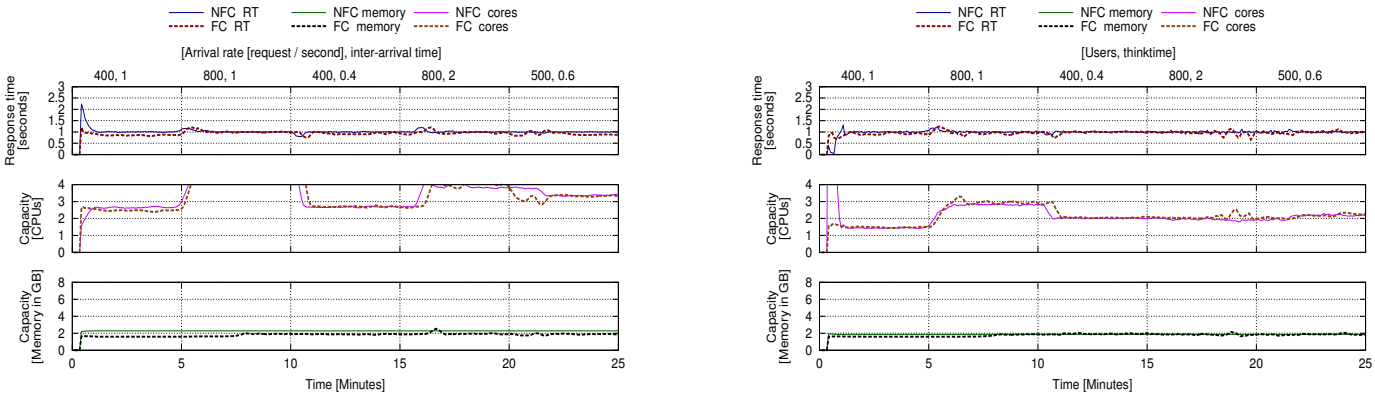
| **Application** (target RT) | **System Model** | **Control Mode** | **ISE** | **IAE** | **Memory usage (mean) [GB]** | **CPU usage (mean) [core]** | **RT (mean) [sec]** | **RT (SD) [sec]** |
|---|---|---|---|---|---|---|---|---|
| Olio (0.5sec) | open | FC | 16.65 | 41.68 | 3.70 | 4.44 | 0.56 | 0.22 |
| | | NFC | 34.69 | 56.10 | 4.57 | 10.21 | 0.59 | 0.32 |
| | closed | FC | 14.32 | 34.50 | 4 | 1.99 | 0.50 | 0.21 |
| | | NFC | 24.03 | 52.47 | 4.73 | 2.04 | 0.52 | 0.27 |
| Olio (1sec) | open | FC | 7.60 | 38.15 | 3.70 | 2.21 | 0.98 | 0.15 |
| | | NFC | 37.39 | 48.15 | 4.57 | 2.34 | 1.01 | 1.29 |
| | closed | FC | 28.12 | 66.11 | 4.00 | 1.39 | 0.98 | 0.29 |
| | | NFC | 34.90 | 75.21 | 4.73 | 1.37 | 1.04 | 0.33 |
| Olio (1.5sec) | open | FC | 40.45 | 86.24 | 3.93 | 1.86 | 1.46 | 0.35 |
| | | NFC | 34.07 | 74.91 | 8.08 | 1.92 | 1.50 | 0.32 |
| | closed | FC | 42.36 | 89.69 | 2.25 | 1.80 | 1.47 | 0.36 |
| | | NFC | 39.48 | 84.17 | 4.81 | 1.86 | 1.51 | 0.35 |
| RUBiS (0.5sec) | open | FC | 25.25 | 42.43 | 2.67 | 1.70 | 0.55 | 0.27 |
| | | NFC | 190.66 | 60.10 | 6.81 | 1.75 | 0.56 | 0.76 |
| | closed | FC | 16.52 | 43.15 | 2.19 | 0.91 | 0.50 | 0.22 |
| | | NFC | 355.89 | 81.94 | 4.19 | 0.90 | 0.56 | 1.04 |
| RUBBoS (1sec) | open | FC | 7.04 | 32.51 | 1.78 | 3.50 | 0.93 | 0.14 |
| | | NFC | 10.95 | 21.32 | 2.23 | 3.50 | 1.02 | 0.18 |
| | closed | FC | 13.04 | 40.52 | 1.76 | 2.16 | 0.95 | 0.20 |
| | | NFC | 15.61 | 39.45 | 1.83 | 2.33 | 0.98 | 0.22 |




((a)) open system model

((b)) closed system model

Figure. 7: RUBiS– under open and closed system models with 0.5sec target response time.




((a)) open system model

((b)) closed system model

Figure. 8: RUBBoS– under open and closed system models with 1sec target response time.

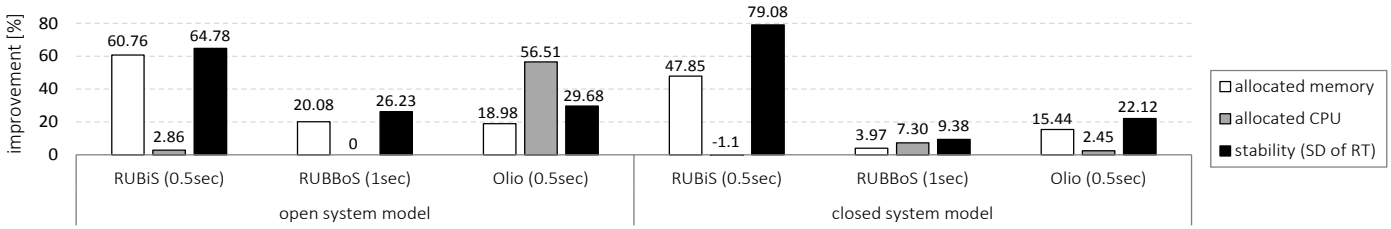



Figure. 9: Comparison of the aggregated results under FC compared to NFC for the allocated memory, allocated CPU, and stability of RT, based on the values reported at Table II.

Table II shows these aggregated results of the three applications under different target RTs and system models. As shown in Table II, for almost all the benchmark applications the ISE and IAE are relatively smaller for FC compared to NFC under both system models. This implies that our novel FC takes better decisions than NFC. In some of instances where NFC has smaller values of the aggregate errors, it has higher resource allocations and standard deviation indicating that the allocations were either over-provisioned or under-provisioned during the experiment. Generally, the average resources allocated under FC is almost always less than under NFC. This difference is significant in some experiments for either of the resources under FC in comparison with NFC. Fig. 9 presents the percentage of improvements for FC compared to NFC in terms of allocated resources and stability of the RT, i.e., lower value for the standard deviation. In general, it shows that FC is more stable and allocates less resources compared to NFC.

As shown in Fig. 9, under RUBiS open system model experiment, the allocated memory under FC was less by 60.76% (2.67 GB vs. 6.81 GB) compared to NFC for similar workload pattern while having 64.78% more stability in RT. In case of CPU, while the values for both FC and NFC were comparable, we observed less CPU cores in some experiments under FC such as Olio (0.5 sec, open model) which has allocated 56.51% (4.44 CPU cores vs. 10.21 CPU cores) less CPU cores compared to NFC while again having improvement in both allocated memory (18.98%) and stability of RT (29.68%). Besides, even though the aggregated mean RTs were relatively comparable for both FC and NFC, the standard deviation for FC is relatively smaller than NFC, i.e., more stability in case of RT. This implies that the resources allocated under NFC were less well matched with the needs compared to FC. Moreover, it also indicates that there were more oscillations under NFC than FC due to improper prediction of the resources.

In general, in all scenarios under NFC, more CPU and memory were allocated on average during the experiment than with FC under similar configurations despite the fact that the aggregate mean RTs are comparable. These results clearly reveal that by using our novel FC the target RTs were met with lower oscillation (lower values for the standard deviation of RT) while avoiding under- and over-provisioning of either of the resources. This happens due to *fuzzy controller* that coordinated the level of requirements of each resource. Thus, FC is an effective approach that meets the target RTs slightly better, using a substantially lower amount resources.

### D. Discussion

Experiments highlighted the need for coordination of the two vertical controllers acting on different resources for the same goal–meeting applications performance demands. Specifically, coordination among the different controllers prevents conflicting decisions such as under- or over-provisioning of one or more of the resources while avoiding performance violations. The experimental results revealed that under open system model relatively more improvement was achieved in terms of allocation of both resources and stability in term of application RT (see Fig. 9). On the contrary, uncoordinated decision making can lead to non-deterministic behavior due to the fact that each controller makes decision in isolation without considering the effect of the other. That is why the novel FC was able to predict the right amount of resources required to satisfy applications' demands under a variety extreme conditions using workloads generated based on open and closed system models. On the other hand, the behavior of NFC, as a baseline approach, was non-deterministic leading to either resource over-provisioning or under-provisioning. In general, FC was able to adapt the resources by observing the applications' performance, and average resources' utilization without needing to be explicitly notified about changes in the workload patterns. In summary, the experimental results reveal the following **key findings**:

(i) the behavior of NFC was non-deterministic leading to either over provisioning or under-provisioning of memory. The applications performance was met during over-provisioning of memory as the application performance was literally controlled by *cpu controller*. However, this leaves ample unused memory that could have been used by other VMs of the same PM. On the other hand, the most serious issue was when memory was under-provisioned which led the application to crash;

(ii) FC was able to meet applications performance while resources were efficiently utilized. This is because FC is able to reason about the contribution of each resource under uncertainty (using fuzzy logic) by observing the corresponding average utilization values and application RT;

(iii) FC maintains high utilization of resources. The proposed FC not only guarantees the application performance but also achieves high utilization of resources as they are used as the decision making criteria. Thus FC provides a win-win scenario for both application owners and also cloud infrastructure providers.

In summary, depending on the nature of the workload, an application can intensively need arbitrary combinations of resources (e.g., CPU or memory) at different stages of its execution. Therefore, uncoordinated deployment of resource elasticity controllers that control different resources for the same application may lead to unpredictable behavior such as resource over-provisioning, which forces customers to pay for unused resources, or application crashing due to severe resource shortage as a result of conflicting decisions. To overcome such issues, careful coordination of controllers ensures application performance is met with optimal amount of resources, preventing both resource over- and under-provisioning.

## VI. Related Work

In theory, any resource could be elastic, however, the practical exploitation depends on the type of the resource, cost and complexity of the implementation. Since the focus of this paper is on vertical scaling of CPU and memory, we review work related to these resources. Moreover, we briefly introduce work on fuzzy control, as a used approach in our work.

### A. CPU Elasticity

Kalyvianaki et al. [5] design a controller using Kalman filtering to control allocation based on the CPU utilization. The authors in [4] propose a two stage controller to allocate CPU cores for different tiers of a multi-tier application. While the first controller regulates the relative CPU utilization of each tier, the second controller adjusts the allocations in cases of CPU contention. Yazdanov and Fetzer [30] develop a specific solution for vertical scaling of CPU resources. Their solution are built on top of Xen Hypervisor using combination of on-the-fly plugging CPU and tuning virtual CPU power to provide a finer grain control of the physical resources associated to the VM. Spinner et al. [6] propose a model-based approach that uses the relationship between the resource allocation and the observed application performance to automatically extract and update the model using resource demand estimation techniques. Lakew et al. [2] present two generic response time performance models, queue length based and inverted response time, which map performance to CPU capacity and provide performance guarantees for interactive applications deployed in the cloud. Indeed, this is the *cpu controller* that served as a building-block in the present contribution.

### B. Memory Elasticity

Baruchi et al. [7] compare two techniques for memory elasticity: exponential moving average (EMA), and page faults. They demonstrate that scaling memory using page faults improve performance as compared to the EMA technique. Wang et al. [9] propose a mechanism to dynamically set the amount of memory required to meet the performance of an application using sampling techniques. The authors in [3] use rules to adapt the VM memory size to the application requirements. The application's memory usage is monitored and then vertical elasticity rules are applied in order to dynamically change the memory size by using the memory ballooning technique provided by KVM Hypervisor. Molto et al. [3] present a mechanism for adjusting the VM memory size based on the memory consumption pattern of the application using a simple elasticity rule.

### C. CPU and Memory Elasticity

Lu et al. [31] develop a tool to automatically set resource control for both VMs and resource pools to meet performance of the application level, as well as resource pool level. For the former, they translate performance objectives into the appropriate resources, consisting memory and CPU, by controlling setting of the individual VMs running that application. At the resource pool level, they ensure that all important applications within the resource pool can meet their performance targets by adjusting controls at the resource pool level. In [11] a MIMO controller is designed to regulate server CPU and memory utilization within specified QoS value for `Apache`. They show that the MIMO control technique is able to handle the trade-offs between speed of metric convergence and sensitivity to random fluctuations while enforcing the desired policies. Apache CloudStack as a recent open source software [12] tries to add the ability to scale up CPU and/or memory for running VMs based on the predefined compute offerings for different Hypervisors (Xen, VMware, and KVM). Some more recent approaches to the problem considered herein are either focusing on single resource (CPU or Memory) or the other which consider both CPU and memory [11], [12] use resource utilization as a decison making criteria which is oblivious to application performance [32]. The work proposed in [31] is an application-driven model that tries to ensure response time below a certain threshold. However, the approach may lead to resource over-provisioning. The authors of [2] and [10] propose a significantly faster average response time models by using parameter estimation techniques for CPU and memory, respectively. These models require only minimal training or knowledge about the hosted applications while simultaneously reacting as quickly as possible to changes in workloads. In this paper, we combine these two techniques to meet applications' response time targets. Moreover, in order to efficiently utilize the resources while meeting the targets, fuzzy control is employed. The goal of this work is to efficiently utilize the resources (i.e., maintain high utilization for both CPU and memory) while meeting the RT targets at the same time.

### D. Fuzzy Control

The main difference between the traditional model-based control and knowledge-based control is the assumption of the former of the availability of a precise and explicit mathematical model of the system under control. Whereas, the knowledge-based control does not make such an assumption but rely on expert knowledge, instead [21]. Deriving an accurate mathematical model of the underlying software system is a daunting task due the non-linear dynamics of real systems [33], [34]. Fuzzy control is a known knowledge-based control approach which has been applied for dynamic resource allocation in cloud [35]. In fuzzy control, which is typically called as model-free approach, such non-linear functions of the target system is implicitly constructed through fuzzy rules and fuzzy inference by imitating human control knowledge [36], [37]. In this work, we used fuzzy control to reason and determine the contribution of each resources to the performance changes. This in turn guides the allocation decision of the controllers for each resource.

## VII. Conclusion and Future Work

In this paper, we proposed an autonomic resource controller consisting of three sub-controllers: *fuzzy controller*, *cpu controller*, and *memory controller*. *Fuzzy controller* acts as a coordinator so that the control actions of the cpu and memory vertical controllers complement each other in order to fulfill the application's performance requirements expressed in terms of response time. *Cpu controller* and *memory controller* allocate the right amount of CPU and memory respectively using the inputs provided by *fuzzy controller*. In general, the proposed fuzzy control approach can be used as a coordination technique among distributed controllers. We evaluated the proposed solution using RUBiS, RUBBoS, and Olio– three widely used cloud benchmark interactive applications– on a virtualized environment using Xen Hypervisor. Different experiments were conducted under workload traces generated

based on open and closed system models. The results show that the proposed coordination solution was able to maintain the target response time with fewer control errors and more efficient resource use, e.g., up-to 60% less memory usage in case of the RUBiS experiment or up-to 56% less CPU usage in the experiment with Olio compared to a non-fuzzy approach used as a baseline. We envision extending the work in several ways: (i) focusing on tail response time values instead of mean values; (ii) enhancing *fuzzy controller* with features such as online learning for self-adaptation of the fuzzy rules and membership functions; (iii) extending the proposed solution to complement vertical elasticity with horizontal elasticity.

## ACKNOWLEDGMENT

The authors would like to thank Pooyan Jamshidi for his constructive comments. This work was partially supported by the HALEY project, the Vienna Science and Technology Fund (WWTF) through the PROSEED grant, the Swedish Research Council (VR) project Cloud Control, and the Swedish Government's strategic effort eSSENCE. This collaborative research was initiated with an STSM granted by IC1304 COST-ACROSS action.

## REFERENCES

[1] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building more robust cloud applications," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 700–711.

[2] E. B. Lakew, C. Klein, F. Hernandez, and E. Elmroth, "Towards Faster Response Time Models for Vertical Elasticity," in *IEEE Conference on Utility and Cloud Computing (UCC)*, 2014, pp. 560–565.

[3] Moltó, Germán and Caballer, Miguel and Romero, Eloy and de Alfonso, Carlos, "Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements," *Procedia Computer Science*, vol. 18, pp. 159–168, 2013.

[4] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem, "Adaptive control of virtualized resources in utility computing environments," in *SIGOPS Operating Systems*, vol. 41, no. 3, 2007, pp. 289–302.

[5] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters," in *International conference on Autonomic computing*, 2009, pp. 117–126.

[6] S. Spinner, S. Kounev, X. Zhu, L. Lu, M. Uysal, A. Holler, and R. Griffith, "Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation," in *8th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, 2014, pp. 157–166.

[7] A. Baruchi and E. T. Midorikawa, "A survey Analysis of Memory Elasticity Techniques," in *Euro-Par 2010 Parallel Processing Workshops*, 2011, pp. 681–688.

[8] W. Dawoud, I. Takouna, and C. Meinel, "Elastic Virtual Machine for Fine-grained Cloud Resource Provisioning," in *Global Trends in Computing and Communication Systems*. Springer, 2012, pp. 11–25.

[9] Y. Wang, C. C. Tan, and N. Mi, "Using Elasticity to Improve Inline Data Deduplication Storage Systems," in *IEEE International Conference on Cloud Computing (Cloud)*, 2014, pp. 785–792.

[10] S. Farokhi, P. Jamshidi, D. Lucanin, and I. Brandic, "Performance-based Vertical Memory Elasticity," in *12th IEEE International Conference on Autonomic Computing (ICAC)*, 2015, pp. 151–152.

[11] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server," in *Network Operations and Management Symposium*, 2002, pp. 219–234.

[12] (2014) Dynamic scaling of CPU and RAM. Available online: https://cwiki.apache.org/confluence/display/CLOUDSTACK/Dynamic+scaling+of+CPU+and+RAM, Visited 2015-05-29.

[13] F. A. de Oliveira Jr and T. Ledoux, "Self-management of Cloud Applications and Infrastructure for Energy Optimization," *SIGOPS Operating Systems Review*, vol. 46, no. 2, p. 10, 2012.

[14] N. Grozev and R. Buyya, "Multi-cloud Provisioning and Load Distribution for Three-tier Applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 3, p. 13, 2014.

[15] (2014) RUBiS: Rice University Bidding System. Available online: http://rubis.ow2.org.

[16] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open Versus Closed: A Cautionary Tale," in *Networked Systems Design and Implementation (NSDI)*, vol. 6, 2006, pp. 18–32.

[17] Apache Performance Tuning. [Online]. Available: http://www.devside.net/articles/apache-performance-tuning

[18] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[19] W. Liu, J. C. Principe, and S. Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*. John Wiley & Sons, 2011, vol. 57.

[20] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *36th International Conference on Software Engineering (ICSE)*, 2014, pp. 299–310.

[21] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic Resource Provisioning for Cloud-based Software," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2014, pp. 95–104.

[22] S. Farokhi, P. Jamshidi, I. Brandic, and E. Elmroth, "Self-adaptation Challenges for Cloud-based Applications: A Control Theoretic Perspective," in *10th International Workshop on Feedback Computing*, 2015.

[23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," vol. 37, no. 5. ACM, 2003, pp. 164–177.

[24] (2014) Rubbos. Available : http://jmob.ow2.org/rubbos.html.

[25] (2014) Olio. Available online: http://incubator.apache.org/projects/olio.html.

[26] Z. Gong, X. Gu, and J. Wilkes, "PRESS: Predictive Elastic ReSource Scaling for cloud systems," in *International Conference on Network and Service Management (CNSM)*, 2010, pp. 9–16.

[27] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems," in *2nd ACM Symposium on Cloud Computing*, 2011, p. 5.

[28] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "DejaVu: Accelerating Resource Allocation in Virtualized Environments," vol. 40, no. 1. ACM, 2012, pp. 423–436.

[29] J. Rada-Vilela, "Fuzzylite: a Fuzzy Logic Control Library," 2014. [Online]. Available: http://www.fuzzylite.com

[30] L. Yazdanov and C. Fetzer, "Vertical Scaling for Prioritized VMs Provisioning," in *Cloud and Green Computing*, 2012, pp. 118–125.

[31] L. Lu, X. Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni, "Application-driven Dynamic Vertical Scaling of Virtual Machines in Resource Pools," in *Network Operations and Management Symposium (NOMS)*, 2014, pp. 1–9.

[32] E. B. Lakew, C. Klein, F. Hernandez-rodriguez, and E. Elmroth, "Performance-Based Service Differentiation in Clouds," in *15th IEEE/ACM International, Symposium on Cluster Computing and the Grid (CCGrid)*, 2015, pp. 505–514.

[33] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[34] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and P. Padala, "What Does Control Theory Bring to Systems Research?" *SIGOPS Operating Systems*, vol. 43, no. 1, pp. 62–69, 2009.

[35] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif, "On the Use of Fuzzy Modeling in Virtualized Data Center Management," in *4th International Conference on Autonomic Computing (ICAC)*, 2007, pp. 25–25.

[36] J. Rao, Y. Wei, J. Gong, and C.-Z. Xu, "DynaQoS: Model-free Self-tuning Fuzzy Control of Virtualized Resources for QoS Provisioning," in *International Workshop on Quality of Service (IWQoS)*, 2011, pp. 1–9.

[37] P. Lama and X. Zhou, "Autonomic Provisioning with Self-adaptive Neural Fuzzy Control for Percentile-based Delay Guarantee," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 2, p. 9, 2013.