A Hybrid Cloud Controller for Vertical Memory Elasticity: A Control-theoretic Approach

Soodeh Farokhi*, Pooyan Jamshidi†, Ewnetu Bayuh Lakew§, Ivona Brandic*, and Erik Elmroth§
*Faculty of Informatics, Vienna University of Technology, Austria.
†Department of Computing, Imperial College London, United Kingdom.
§Department of Computing Science, Umeå University, Sweden.
Email: *{firstname.lastname}@tuwien.ac.at, †p.jamshidi@imperial.ac.uk, §{ewnetu, elmroth}@cs.umu.se

***Abstract*—Web-facing applications are expected to provide certain performance guarantees despite dynamic and continuous workload changes. As a result, application owners are using cloud computing as it offers the ability to dynamically provision computing resources (e.g., memory, CPU) in response to changes in workload demands to meet performance targets and eliminates upfront costs. Horizontal, vertical, and the combination of the two are the possible dimensions that cloud application can be scaled in terms of the allocated resources. In vertical elasticity as the focus of this work, the size of virtual machines (VMs) can be adjusted in terms of allocated computing resources according to the runtime workload. A commonly used vertical resource elasticity approach is realized by deciding based on resource utilization, named capacity-based. While a new trend is to use the application performance as a decision making criterion, and such an approach is named performance-based. This paper discusses these two approaches and proposes a novel hybrid elasticity approach that takes into account both the application performance and the resource utilization to leverage the benefits of both approaches. The proposed approach is used in realizing vertical elasticity of memory (named as vertical memory elasticity), where the allocated memory of the VM is auto-scaled at runtime. To this aim, we use control theory to synthesize a feedback controller that meets the application performance constraints by auto-scaling the allocated memory, i.e., applying vertical memory elasticity. Different from the existing vertical resource elasticity approaches, the novelty of our work lies in utilizing both the memory utilization and application response time as decision making criteria. To verify the resource efficiency and the ability of the controller in handling unexpected workloads, we have implemented the controller on top of the Xen hypervisor and performed a series of experiments using the RUBBoS interactive benchmark application, under synthetic and real workloads including Wikipedia and FIFA. The results reveal that the hybrid controller meets the application performance target with better performance stability (i.e., lower standard deviation of response time), while achieving a high memory utilization (close to 83%), and allocating less memory compared to all other baseline controllers.**

Keywords: cloud computing, feedback control loop, vertical memory elasticity, performance, response time, memory utilization, interactive applications.

I. Introduction

Cloud applications face dynamic and bursty workloads generated by variable numbers of users. Therefore, dynamic resource provisioning is necessary not only to avoid the application performance degradation, but also to efficiently utilize resources. Consequently, infrastructure providers ought to have a resource provisioning technique that allocates resources according to application demands in order to attract customers and to use their resources efficiently.

Since the users of modern interactive applications are becoming increasingly interested to have high and predictable, if not guaranteed, performance, the need for having robust auto-scaling[1] solutions that would meet their service level agreement (SLA) is rising for cloud environments. Otherwise, unexpected workloads can cause a poor service performance that kills user's satisfaction. Several studies have shown that increased response times reduce, revenue [1]. For instance, Amazon found a page load slowdown of just 1sec could cost $1.6 billion in sales each year [2]. Google stated only half a second delay in search page generation time dropped traffic by 20% [3].

Resource elasticity, as one of the main selling points of cloud computing [4, 5, 6], is defined as the degree to which a cloud service is able to accommodate the varying demands at runtime by dynamically provisioning and releasing resources, such that the available resources match the current demands closely [6]. Two types of resource elasticity are defined: horizontal and vertical. While horizontal resource elasticity allows virtual machines (VMs) to be acquired and released on-demand, vertical resource elasticity allows adjusting computing resources (e.g., CPU or memory) of individual VMs to cope with runtime changes. Accordingly, vertical memory elasticity, as the focus of our work, is the case where the size of the allocated memory of the VM is dynamically changed at runtime. Generally speaking, horizontal resource elasticity is coarse-grained, i.e., VMs are considered as resources, which have static and fixed size configurations. Vertical resource elasticity, on the other hand, is fine-grained: the size of the VMs in terms of a particular computing resource such as CPU or memory can be dynamically changed to an arbitrary size for as short as a few seconds [5].

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 SLA violations since the demand cannot always exactly fit the size of the VM. To efficiently utilize resources and avoid SLA violations, horizontal elasticity should be complemented with fine-grained resource allocations

[1]"Auto-scaling" and "Elasticity" are used alternatively in this paper.

where the VM sizes can be dynamically adjusted to an arbitrary allocated computing resources according to runtime demands. Moreover, based on a European Commission report on the future of cloud computing [7], vertical elasticity is one of the areas that is not fully addressed by current commercial efforts, although its importance is acknowledged. For example, vertical elasticity is considered as a key enabling technology to realize resource-as-a-service (RaaS) clouds and one of the main driving features of the second-generation Infrastructure as a Service (IaaS 2.0) [8], in which users pay only for the resources they actually use, and cloud providers can use their resources more efficiently and serve more users [5, 9].

Nevertheless, from the research point of view, in the last decade, most elasticity research has focused on horizontal, while only few research efforts have addressed vertical elasticity [10] due to lack of support from hypervisors. However, vertical scaling of resources has recently started to be supported by the hypervisors such as Xen [11] and KVM [10]. Unlike horizontal elasticity that is widely supported by almost all commercial cloud providers, only a few cloud providers such as *DotCloud*[2] and *ProfitBricks*[3] have started commercial support for vertical elasticity. However, with the current rate of technological developments and user expectations, the support of vertical elasticity techniques will become necessary by any public cloud providers in the future [10].

In theory, one can turn any computing resource, like CPU or memory, vertically elastic if there is a way to measure its behavior continuously over time and if there is at least one knob to change its behavior. However, the practical exploitation of vertical resource elasticity is very challenging due to the following reasons: (i) intrinsically dynamic and unpredictable nature of the and applications' workloads; (ii) the difficulty in determining which resource (e.g., memory or CPU) is the bottleneck [12]; (iii) non-trivial relationship between the performance metrics (e.g., throughput or response time) and the amount of required resources; (iv) detecting when and how much of resources can be added to or removed from the VM while maintaining the desired application performance.

In this work, we used control theory to synthesize a controller for vertical memory elasticity of cloud applications, i.e., the elastic resource is memory and it is adjusted by the controller. The main motivation behind the choice of control theory in our work is to use this well-established theory for modeling and designing feedback loops to make the cloud applications self-adaptive and achieve a proper balance between fast reaction and better stability. Moreover, since the time to adjust memory at runtime is close to instantaneous, control theory is a good fit. The proposed approach, named *hybrid memory controller*, takes the advantages of both the performance-based (PC) and the capacity-based (CC) elasticity control approaches. As commonly used vertical resource elasticity approaches, CC approaches take the resource utilization as a decision making criterion to do the resource elasticity, while PC approaches, give the priority to the application performance and adjust the resources in accordance to the application performance metrics such as response time. However, a CC approach is inadequate to ensure the application performance, and a PC approach may not be able to provide the level of performance assurance that a CC approach can provide efficient resource utilization as the performance of an application can also be affected, for example, by bugs inside the application or by other resources which are not considered by the controller. Therefore, using both the application performance and the resource utilization at the same time would allocate the right amount of resources for the application while preventing both under and over-provisioning. In summary, at the designed feedback loop of the proposed *hybrid memory controller*, scaling up or down of the allocated memory is considered as the control knob parameter, while the application response time (RT) and VM memory utilization are used as feedback variables.

We evaluated the *hybrid memory controller* using RUBBoS [13] as an interactive benchmark application and compare the results of the hybrid controller with a performance-based controller [14, 15] and a capacity-based controller [10]. We validate our approach using synthetic traces generated based on open and closed user loop models [16] along with the real user request traces of Wikipedia [17] and FIFA WorldCup [18] websites, which are the number of requests/ users accessing these two websites per unit time. Results show that the *hybrid memory controller* ensures efficient resource utilization as well as meeting application performance.

Contribution. The contribution of this paper lies in developing and experimentally evaluating a vertical memory controller that takes both memory utilization and application performance as indicators to adjust the memory size of the VM hosting an application at runtime. The goal is to provide guaranteed performance using RT as a key performance indicator (KPI) for cloud applications. We show that the new approach can fulfill the performance guarantees of the application while avoiding over- or under-committing the allocated memory. Specifically, our contributions are:

1) *Performance model design*– a performance model is proposed based on control theory in order to determine how each application is behaving compared to its target performance.
2) *Memory prediction*– using the output from the performance model and the actual memory utilization, the amount of memory to be allocated for each application is predicted.
3) *Performance guarantee and efficient resource utilization*– the proposed approach ensures that the application performance is met while maintaining high utilization of memory.
4) The proposed approach is evaluated using a real application under different workloads. We compare its performance along with resource usage with both performance-based and capacity-based approaches.

The remainder of this paper is organized as follows: Section II motivates our work. Section III provides background regarding memory vertical elasticity and the mechanisms how it can be achieved. Section IV presents the proposed *hybrid memory elasticity* approach, including the design of the system model and the memory controller. The experimental evaluation and discussion of the results are presented in Section V. Key findings, the limitation of our work, and the threats to validity

[2]docCloud: https://www.dotcloud.com/
[3]ProfitBricks:www.profitbricks.com

are presented in Section VI. Finally, Section VII presents the related work, and Section VIII concludes the paper and brings up the opportunities for the future research directions.

II. Motivation

The pervasive and popular architectural patterns for a cloud application is the 3-tier pattern [19]. 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 application may be involved in processing user requests, thus, requiring different resources such as memory. Therefore, resource auto-scaling may be required for one or more tiers in order to meet application performance. In this work, we focus on vertical memory scaling of the BL tier.

Based on the `Apache` Web server performance tuning tip [20], "the single biggest issue affecting web-server performance is RAM. The more RAM your system has, the more processes and threads `Apache` can allocate and use; which directly translates into the amount of concurrent requests/ clients `Apache` can serve.". In order to show that the dynamic thread creation of `Apache` is working as expected in accordance to the allocated memory and its effects the application performance, we conducted an experiment using `ab`, `Apache` HTTP Server benchmarking tool[4] for RUBBoS application. The result is depicted in Fig. 1.

We deployed the BL and DS tiers of RUBBoS on different VMs, while provisioning sufficient memory and CPU cores to the VM hosting the DS tier. We also over-provisioned the VM hosting the BL tier in terms of CPU (8 CPU cores), while we configured three different values for the allocated memory (i.e. 1GB, 2GB, and 4GB) in order to show the effect of memory on the application performance. By using a workload generator tool, `httpmon`[5], we defined variable workload dynamics which stress BL tier for memory during the application life span at runtime by slowly increasing the number of users (i.e., requests/ sec) from 100 to 1000 (with 100 users increment each time). During the experiment, the number of concurrent `Apache` processes was monitored by checking the `Apache` server status[6], which were dynamically varying according to the number of concurrent users and allocated memory size. As shown in Fig. 1, when more memory is allocated to the VM, a better application performance is achieved as `Apache` can create more threads. As the number of concurrent users is increased, the difference in performance (in both throughput and RT) among the three memory configurations also becomes more apparent. The aim of applying an auto-scaling solution for such a scenario is to dynamically adjust the amount of memory to keep the application performance, e.g., RT in this work, under a desired value regardless of the variation in workloads.

Note that allocating more memory to the VM-hosting 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 application RT. However, due to the lack of research on memory elasticity of the BL tier (e.g., for `Apache` Server), and the need for the further application support in case of vertical memory auto-scaling of the DS tier (e.g., for MySQL [21]), we chose to primarily concentrate on BL tier. The proposed solution targets applications that can benefit from live memory elasticity at runtime, i.e. application with dynamic memory requirements, and also applications in which the performance bottleneck is memory not CPU.

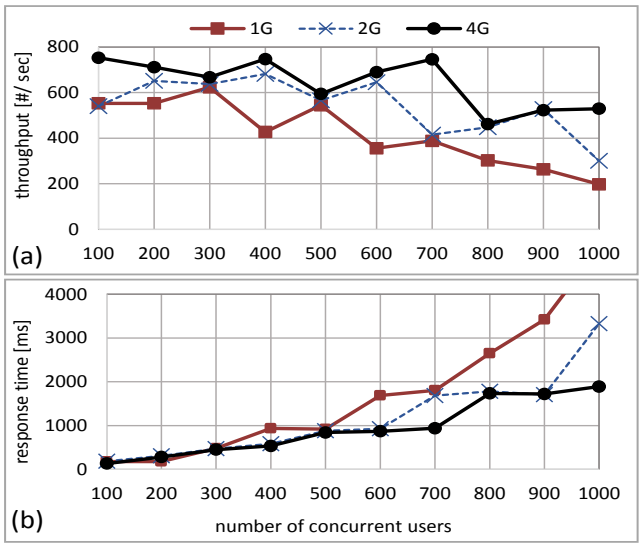


Figure. 1: The effect of vertical memory scaling on the RUBBoS application performance: (a) throughput; (b) response time.

III. Memory Elasticity Background

In this section, in order to explain the logic behind the proposed hybrid approach, we first introduce current approaches for vertical resource elasticity and then explain mechanisms to enforce memory actuation.

A. Vertical Elasticity Approaches

Resource vertical elasticity approaches can be categorized into performance-based, capacity-based, and hybrid approaches:

(i) *Capacity-based approaches*.As the most popular elasticity approach from the cloud provider point of view, this approach uses resource utilization as the decision making criteria to change the amount of resource to be allocated [22, 23, 24]. In other words, the utilization data is used to estimate the required resource at runtime. Efficient use of resources is one of the main advantages of such an approach, but on the other hand, decision making based solely on utilization of the resources can lead to violating performance guarantees of the application as the decisions are oblivious to applications performances [25]. In other words, a sustained high resource utilization can tell us that the system suffering, but it cannot determine by how much. While the whole point of elasticity is to figure out by how much we need to scale up to meet the current demand or scale down to avoid the resource wastage, and consequently taking the appropriate action [26].

(ii) *Performance-based approaches*. Since modern applications are getting more performance sensitive, as a new

[4] `ab`: https://httpd.apache.org/docs/2.2/programs/ab.html
[5] `httpmon`: https://github.com/cloud-control/httpmon
[6] `w3m http://VM-IP/server-status`

trend of the resource elasticity, this category of approaches decide to what extent either increase or decrease the allocated resources based on the application performance properties, such as response time or throughput at runtime [5, 15, 25].

(iii) *Hybrid approaches*. As the main contribution of this work, we bring up a new approach named hybrid in which we leverage from the advantages of both performance-based and capacity-based approaches. The rationale behind such an approach is that while a capacity-based approach is inadequate to ensure the application performance, a performance-based approach may not able to provide a sufficient level of resource efficiency that a capacity-based approach can provide. Therefore, by taking both the application performance and the resource utilization as two decision making criteria would result to satisfy the application users in terms of performance, as well as the application owner in terms of the application resource usage. Detailed explanation of the proposed hybrid approach is presented in Section IV.

B. Hypervisor-level Mechanisms to support Memory Elasticity

The hypervisor is responsible to support vertical elasticity.Vertical memory elasticity operation requires some support from the VM's kernel, hence two mechanisms are commonly mentioned:

(i) *Hot memory add or remove*. Adding or removing resources without having to reboot the system is called hot add or remove. Assuming a kernel supports *hot memory add or remove*, this concept can easily be extended to virtual environments: whenever the hypervisor wants to take memory from a virtual machine, it would request it through a VM-hypervisor interface, and the VM's kernel would be elastic with respect to memory. This mechanism is not widely used since it cannot be supported by the guest operating systems without restarting the VM.

(ii) *Memory ballooning*.

In this mechanism, instead of adding or removing memory, the VM's kernel can ban the usage of a portion of memory in spite of the fact that initially it was allocated to the VM. This is achieved by running a custom device driver, the so-called *ballooning driver*, in the VM's kernel, which creates a bridge between the hypervisor and the VM. Using this mechanism, the VM's kernel is booted with a certain amount of memory. Initially, the balloon would be *deflated*, i.e., the ballooning driver would request no memory from the VM's kernel. Hence, the VM could use all the initial memory. If the hypervisor wants to reduce the memory allocation of the VM, then it would tell the balloon to inflate to that amount. When the balloon expands, the physical memory available in the VM is reduced that compels the guest operating system to reduce the memory footprint of other processes when insufficient free memory is detected; for instance, via passing some of the processes' memory pages to the swap space, or killing some of them in extreme situations. Then, the memory allocated by the balloon process in the guest OS can be reclaimed by the host OS, and can then be used by other co-located VMs, enabling a higher consolidation ratio on the physical host [27]. Finally, if the hypervisor decides to increase the memory allocated to the VM, it would map that amount to the VM address space, in the region allocated by the balloon driver. Now the balloon driver has access to that memory, and can safely release it to the VM's kernel. Despite its complexity, this mechanism reacts almost instantaneously, that is, the guest OS reflects the memory change a few moments after the operation is executed through the hypervisors' API[7] [10].

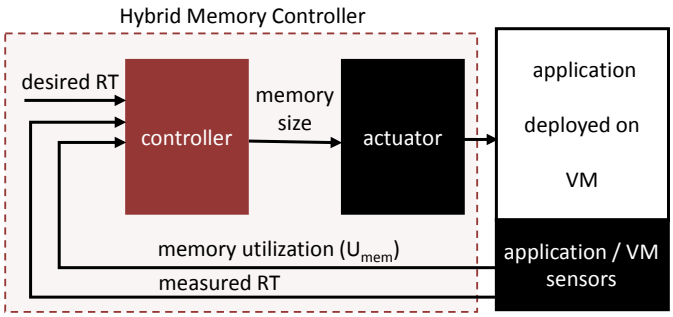


Figure. 2: The architecture of the hybrid memory controller. The colored modules are the contribution of this work.

Based on the above explanation, in contrast to *hot memory add or remove*, *memory ballooning* has some restrictions in order to support memory elasticity: (i) a maximum amount of memory needs to be specified; (ii) some ballooning drivers can only deflate as much as they had been previously inflated. However, in the case of *memory ballooning*, it is supported by all recent Linux kernels and no additional features are required. This makes *memory ballooning* a practical mechanism for realizing vertical memory elasticity, as it is supported by both Xen and KVM hypervisors, while *hot memory add or remove* is currently not supported by any guest OSs without restarting the VM [27]. Therefore, in our work, we utilize the ballooning mechanism provided by the Xen hypervisor via reconfiguration API. The freed memory can then be used by other co-located VMs enabling a higher ratio on the PM [27].

IV. Hybrid Memory Elasticity

In this section, we explain the detailed design of the proposed *hybrid memory controller*.

A. Overview

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

Fig. 2 shows the architecture of the proposed *hybrid memory controller*. It loosely follows a Monitor, Analysis, Planning, and Execution (MAPE) loop based on self-adaptive software terminology [28]. Monitoring gathers information

[7] "`virsh setmem`" in KVM, "`xm mem-set`" in Xen

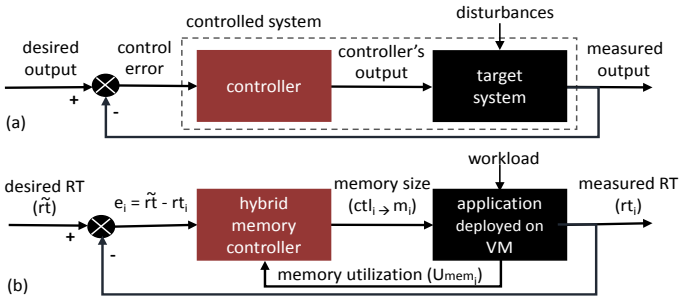


Figure. 3: (a) The standard feedback loop [29] (b) Our feedback loop.

such as the observed RT, average memory utilization from the hosted services at each interval. During analysis, the memory required by an application to meet its performance target is computed using the proposed controller. The goal of the controller is to allocate the right amount of memory in order to meet the application performance target.

The proposed memory controller determines the amount of memory that should be allocated using the application RT and VM memory utilization as decision making criteria. Previous monitoring data is used to fit the model parameters. 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.

- *Controller*. It is an adaptive controller that dynamically tunes the amount of memory required for each application using the values of measured RT given by the application sensor and the desired RT as well as the value of the memory utilization (U_{mem}) given by the VM sensor. It is called adaptive since it dynamically keeps the system model updated at runtime. As the main part of the contribution of this paper, the proposed controller will be discussed in details under Section IV-B.
- *Sensor*. This component gathers the application- and VM-level real time performance information consisting the *mean response time* and the average memory utilization of the application and the allocated VMs, periodically. We refer to this period as the control interval. Memory utilization statistics are monitored over a control interval by using `/proc/meminfo`. These monitoring values are used as a feedback and decision making criteria in *hybrid memory controller* for the next control interval.
- *Actuator*. At each control interval, the controller invokes this module which is the Xen API for memory allocation, to either increase or decrease the allocated memory of the VM hosting the application at runtime.

While *sensor* and *actuator* will be briefly introduced in Section V, the *controller* as the main part of the contribution, will be discussed in details under Section IV-B.

B. System Model and Controller Design

In the context of control theory, a standard feedback control loop is illustrated in Fig. 3 (a) [29]. The controller periodically adjusts the value of the controller's output in such a way that the measured output can stay close to the desired output. The controller aims to maintain the difference between the desired and measured output (referred to as the control error) close to zero, in spite of the disturbances in the target system. The disturbances are what affect the measured output, but they are not controllable [30].

We develop *hybrid memory controller* for cloud applications. In an equivalent feedback loop consisting of *hybrid memory controller*, Fig. 3 (b), the target system is a cloud application deployed on a VM. The controller's output at each iteration i is named ctl_i and is mapped to the memory size mem_i to enable elasticity by scaling the VM memory up or down. The desired RT $\tilde{rt}$ and the measured RT rt_i in our control loop are the desired and measured output, respectively. The control error e_i is the difference between these two values at each iteration, as shown in Eq. (1). The number of user requests (workload) and a change in the mix of different request types in the workload are considered as the disturbances. Since the controller cannot control the workload change, it should adjust the application deployment environment in order to meet the desired RT.

$$e_i = \tilde{rt} - rt_i \tag{1}$$

We adopted and customized the control formula originally devised in [31, 32]. As shown in Eq. (2), the controller's output ctl_i is calculated based on its previous value ctl_{i-1} and a coefficient of the control error. The coefficient is based on the value of α, β, and *pole*, which are described as follows.

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

The system model parameter α represents a first order model of the reaction to the controller's output. Similar to [31], in our work α is calculated at each control interval by applying the linear regression technique based on the effect of ctl on rt. The model building is started when the controller passes a minimum number of control intervals to be able to gather enough information to adaptively calculate and update the value of α, e.g., 25 control interval used in our experiments, before that a default value is used for the α (10 in our experiments). Moreover, to improve the stability of the controller, we apply the weighted moving average (WMA) filtering technique on the gathered values before calculating the value of α at each control interval.

The parameter β is a function of memory utilization $U_{mem_i} \in [0, 1]$ of the VM hosting the application and it is calculated adaptively at each control interval. The rationale behind having such a parameter is to have more insight on the current memory required by the application before increasing or decreasing the memory size. β is defined as shown in Eq. (3).

$$\beta_i = \begin{cases} 1 - U_{mem_i} & e_i > 0 \\ U_{mem_i} & e_i \leq 0 \end{cases} \tag{3}$$

When $e_i > 0$, it means the measured RT rt_i is better than the desired RT, $\tilde{rt}$, (see Eq. (1)), which can indicate the over-provisioning situation where the current allocated memory is more than enough for the application to process the current workload. Therefore, the controller should carefully decrease

the memory to some extent to avoid over-provisioning while still meeting $\tilde{rt}$. However, based on our observation during while performing the experiments, there is a period in which in spite of having a reasonable RT, memory utilization is getting very high. Such a period is when the allocated memory is near to its saturation point, but still the remaining memory is good enough for the application in response faster than the desired RT. In this situation, if the controller only reacts based on the control error, it would decrease the memory and this usually leads to a sudden increment of measured RT. To avoid this situation, β is defined as $1 - U_{mem_i}$ when $e_i > 0$. Therefore, β will be very low when the memory utilization is high, and this will lessen the decrement of the memory (see Eq. (2)).

On the other hand, $e_i \leq 0$ represents an overload condition when the application workload is high and more memory is needed to be able to meet the desired RT, so the controller should increase the amount of allocated memory. Nevertheless, if the memory utilization U_{mem_i} is low it can indicate that memory is not the main reason of having a high RT, so the controller should be conservative on adding a large body of memory in this situation. Therefore, β is defined as U_{mem_i} and consequently this will influence the change at the allocated memory.

The choice of *pole* determines the stability of the controlled system, and how fast it approaches to its equilibrium. The stability of the controller is ensured as long as $0 \leq pole < 1$ [31]. In order to develop a more stable and robust controller, we apply the *weighted moving average* error smoothing method used in time-series analysis for calculating control error before using it in Eq. (2). At each control interval, *hybrid memory controller* tracks $\tilde{rt}$ by rejecting the influence of workload fluctuation on rt_i and withstands the control error e_i as long as it is insignificant. Finally, the controller's output $ctl \in (0, 1)$ is mapped to a memory size $m_i \in [m_{min}, m_{max}]$ using Eq. (4).

$$m_i = ctl_i \cdot (m_{max} - m_{min}) + m_{min} \tag{4}$$

where m_{min} and m_{max} are the minimum and maximum amount of VM memory sizes expressed by the number of memory units[8] m_{unit}, which are allowed to be allocated, m_i is the final output of *hybrid memory controller*. For more extensive details about the hybrid controller design methodology refer to [33].

C. Assessment of the Controller Properties

From the perspective of control theory, a controller should be able to provide four main properties: (i) stability; (ii) absence of overshooting; (iii) low settling time; (iv) robustness to model inaccuracies. First, let's consider the static case, in which the values of α and β are fixed based on the results of the experiments and do not change with time. In this case, the control properties are given by [31], and a choice of the pole between 0 and 1 guarantees stability, absence of overshooting, a robustness that depends on the value of the pole (the closer to 1, the more robust), and a settling time that depends on the value of the pole (the closer to 0, the fastest). This allows the controlled system to trade off robustness for settling time, which is usually done in control theory.

In the case of our proposed controller, where we also consider adaptation of the controller, recalling Eq. (2) what changes is the weight that is given to the error. That is indeed changed by a factor that is given by the changes of β and α. Similar to [31], since the WMA is used for the value of α, so it will not change too fast, so it does not affect the stability of the closed user loop model. Actually, it even improves convergence, it is because by using updated values of α, the system model is a more precise representation of what is happening at the current time. The value of β here is always between 0 and 1, therefore, it affects the settling time but not the stability. Clearly, if a controller acts on less than the error that it experiences, it will be slower in reacting in case the model is correct. However, it also increases robustness, because the controller would take smaller steps towards the satisfaction of the goals. No matter how β is changed, the fact that its value is between 0 and 1 makes it possible to state that stability is preserved, settling time is increased, but with an increase in robustness. This is preferable because apparently the model may not be very precise around some of the operating points - due to memory saturation or some other runtime situation that happens in software systems.

V. Experimental Evaluation

In this section, we present the experimental performance evaluation of the proposed *hybrid memory controller* (HMC) and compare it against two baseline approaches: *performance-based memory controller* (PMC) [14, 15] and *capacity-based memory controller* (CMC) [10]. In what follows, we first describe these baseline approaches, then we explain the experimental setup and finally, we report and discuss the evaluation results.

A. Baseline Approaches

CMC. This approach, as discussed in Section III-A, uses memory utilization as its decision criterion. We used the solution presented in [10] which proposes a vertical elasticity manager (VEM) based on memory utilization. The VEM encompasses an elasticity rule that is applied based on a concept called memory over-provisioning percentage (MOP) $\in [0, 1]$. The idea behind MOP is to avoid thrashing so that to keep the VM memory size beyond the memory used by the application. The proposed elasticity rules enable CMC to decide when to scale up or scale down by monitoring the memory usage $mem_{used} \in [0, 1]$ and the calculated MOP at each control interval (every 5 seconds used in our experiments). As shown in Eq. (5) the mem_{used} is estimated by using the values reported at `meminfo`[9] file, which includes the information about the Linux system's memory at runtime. Similar to HMC and PMC, we define a minimum amount of memory where the controllers cannot shrink the memory size of the VM below this amount. This will allow the guest OS to properly operate and avoid experiencing unexpected application crashes due to the lack of memory.

$$mem_{used} = \text{total} - (\text{free} + \text{cached} + \text{buffers}) \tag{5}$$

The proposed elasticity rule is applied if the free memory of the VM, $mem_{free} = 1 - mem_{used}$, is smaller than 80% or

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

[9] `/proc/meminfo`

greater than 120% of the MOP. Under such conditions, CMC dynamically adjusts the memory size of the VM using Eq. (6).

$$mem_{size} = mem_{used} \cdot (1 + \text{MOP}) \quad (6)$$

This rule implies decreasing or increasing the memory size depends on the behavior of the application deployed on the VM, and the magnitude of the memory changes depends on how fast or slow the application requests or releases the memory. A lower value of MOP aims at reducing the unused memory of the VM, i.e., achieving higher utilization, but has a higher chance to incur in thrashing if the application memory consumption grows faster than the rate at which CMC increases the memory size. In contrast, a higher MOP aims at reducing the chance of thrashing if the memory consumption grows rapidly, but surely at the expense of wasting more memory. Based on our experimental setup (i.e., the used benchmark application and the workload pattern), as suggested in the original work [10], we use 0.1 for the Wikipedia workload trace and 0.2 for the FIFA workload trace as values for MOP. This is because of the different nature of these workloads. A larger value of MOP leads to achieving better result in case of unpredictable and sudden workload such as FIFA workload.

PMC. This approach (see Section III-A), is based on our previously designed memory controller presented in [14]. It is also an adaptive version of the work proposed in [15], which follows a control synthesis technique. As shown in Eq. (7), the control formulation is roughly the same as what we proposed in Eq. (2), but without including the parameter β.

$$ctl_i = ctl_{i-1} - \frac{1 - pole}{\alpha} \cdot e_i \quad (7)$$

At each control interval, similar to HMC, PMC's output $ctl \in [0, 1]$ is mapped to a memory size $mem_i \in [mem_{min}, mem_{max}]$ using the same mapping formula that we used in our work, Eq. (4).

B. Experimental Setup

The experiments were conducted on a physical machine (PM) equipped with 32 cores[10] and 56 GB of memory. To emulate a typical virtualized environment and easily perform vertical elasticity, we used Xen hypervisor. The benchmark application, as shown in Fig. 4, 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 behavior 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 (10 GB) and CPU (10 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 (e.g., `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 of the experiments in our work.

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

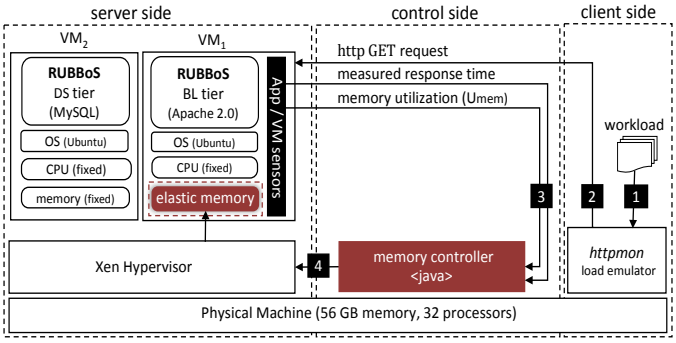


Figure. 4: Experimental setup for the evaluation scenarios. HMC, PMC, or CMC can be used as the memory controller. Numbers indicate the experiment's process.

Benchmark Application. To test the applicability of our contribution, we performed experiments using RUBBoS as an interactive benchmark application. It is an open source multi-tier interactive application. It implements the essential bulletin board features of Slashdot site[11] and has been widely used in cloud research community [23, 34, 35]. It includes two tiers: a front-end tier, which is a Web server that performs the BL by using PHP server side scripts; a back-end tier that stores user information, stories, and related comments. We use an updated PHP version of RUBBoS[12]. The benchmark includes two kinds of workload modes: browse-only and write interaction. We use browse-only workload in all the experiments to more stress the BL tier with read-only `http GET` requests.

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 user loop models [16]. A closed user loop model is defined when the arrival of new requests is only triggered by previous request completions, followed by a delay according to *thinktime*. The effective average request inter-arrival time is the sum of the average *thinktime* and the average response time of the application, hence dependent on the performance of the evaluated application. While, an open user loop model is defined when new requests arrive independently of the previous request completions, typically modeled as Poisson process, requests are issued with an exponentially-random inter-arrival time, characterized by a rate parameter, without waiting for requests to complete. For open clients, we change 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 are varied. The change in arrival rate or number of users is made instantly. This makes it possible to meaningfully compare the system behavior under these two client models. For instance, to increase the number of requests by five-folds or ten-folds to understand the behavior of our solution; to induce memory 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 both open and closed user loop models, we used the `httpmon` workload generator tool. We also kept constant the number of

[11]Slashdot: http://slashdot.org
[12]RUBBoS: https://github.com/cristiklein/brownout/tree/rubbos-icse2014

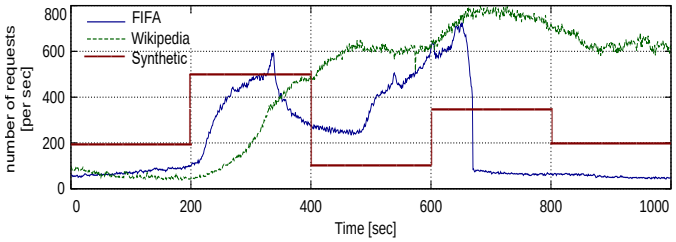


Figure. 5: Workload patterns based on the user requests accessing real Websites: Unpredictable–FIFA-based and Predictable–Wikipedia-based, and synthetic (open and closed user loop models) workload.

requests for some time to study the behavior of the models under both the steady- and transient-states.

In addition, to validate the applicability of our approach against real-life situations, we used two real workloads: (i) the Wikipedia workload [17] extracted from a trace of 10% of all user requests issued to Wikipedia Website (in German language) during the period between September 19, 2007 and January 2, 2008; (ii) the FIFA WorldCup Website access logs [18], named as FIFA workload in our work, that includes all the requests made to the 1998 WorldCup Website between April 30, 1998 and July 26, 1998. As reported in [18], during this period of time the Website received 1,352,804,107 requests. These two traces were selected due to their complementary nature. While the Wikipedia workload shows a steady and predictable trend, the FIFA workload has a bursty and an unpredictable pattern. Fig. 5 depicts the scaled patterns of these two real workloads as well as the synthetic workload used– generated based on open and closed loop models. The scaling of the real workloads is according to the capacity of our experimental setup, while keeping the original patterns. These workloads finally are mapped to the number of concurrent users who send `http GET` requests to the used benchmark application. To facilitate the reproduction of our research, we released the csv files of these workloads[13].

Metrics. The response time 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. Note that, as pointed in [26], suitable auto-scaling metrics are the ones which are related to a single application tier. Such metrics can be tracked, patterns can be learned, statistics can be calculated, and intelligent elasticity controllers then can use them to ensure the user satisfactions in terms of the SLA. This is one of the reasons that in our work we only focus on the BL tier of the cloud application, so based on our experimental setup (see Section V-B), the defined metric is only dependent on the memory resource of the VM hosting the BL tier of the application.

Experiment Process. As shown in Fig. 4 by the numbers, 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 `http GET` requests to the application under test (2). In each control interval, 5 seconds in our experiments, the application sensor observes the average RT and the VM sensor[14] measures and sends the average of memory utilization (3). Both sensors send their monitored information via `TCP/IP` connection to the controller. Depending on the evaluation setup, either HMC, PMC, or CMC is used as the memory controller and then it will dynamically adjust the memory size of VM_2. While for HMC both the memory utilization and measured RT are required, in the case of the other controllers only one of these measurements is enough for the corresponding controller to decide the memory size. Finally, the used memory controller invokes the memory actuator, i.e., the Xen API for memory allocation, to either increase or decrease the allocated memory of VM_1 at runtime (4).

C. Experimental Results

In this section, the evaluation results will be presented and discussed as follows:

(i) *non-adaptive scenarios*. Analyzing the time-series results of a non-adaptive RUBBoS under both Wikipedia and FIFA workloads. Non-adaptive scenarios consisting of over- and under-provisioning the memory;

(ii) *HMC time-series analysis*. Analyzing the time-series results of HMC under various workloadtraces consisting open closed user loop models, Wikipedia and FIFA workloads. The goal here is to show that the proposed controller is able to meet the desired RT without over-provisioning while achieving a relatively high memory utilization;

(iii) *Time-series comparison*. Presenting the results related to the behavior of CMC and PMC under Wikipedia and FIFA workloads and Comparing them with the results of the proposed controller, HMC, with a similar experimental setup;

(vi) *Aggregate analysis*. Reporting the aggregate results related to a self-adaptive RUBBoS equipped with the three controllers under different scenarios as well as a non-adaptive RUBBoS with under- and over-provisioned memory.

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. In each figure, the upper graph plots *mean response times*, the bottom graph plots the memory and CPU utilization of the VM hosting the BL tier (i.e., VM_1) of the application under test. The rationale behind reporting the CPU utilization is to show that in the experiments, CPU has not been the bottleneck resource. Finally, the middle graph plots the amount of memory required in GB computed by the respective controller and allocated to VM_1 over the next 5 seconds as the default control interval.

Non-adaptive scenarios. Figs. 6 and 7 presents the behavior of a non-adaptive RUBBoS, with over-provisioning and under-provisioning of allocated memory under Wikipedia and FIFA workloads, respectively. The aim of these scenarios is to show the application measured RT when the allocated memory

[13]https://gitlab.com/soodeh/workloads/tree/master

[14]https://gitlab.com/soodeh/monitoring-scripts/tree/master

is static under different workloads. It can be observed that in the over-provisioning (Figs. 6a and 7a), the target RT easily was met with allocating 4GB of memory, but with the expenses of wasting the memory most of the time during the experiments and consequently a high resource cost, while achieving relatively low resource utilization. On the other hand, in the case of under-provisioning experiments (Figs. 6b and 7b), the measured RT roughly follows the workload pattern and is much higher than the target RT from when the workload started to increase while achieving a very high resource utilization. However, the application is unable to respond quickly and handle the peak periods, so all requests after this time face the measured RTs far higher than the desired RT. More precisely, under sever lack of memory and a large amount of user requests, the Web server enters a state where it is unable to respond to any single request. If this happens, the reaction of the VM to memory increment will become much slower than normal. Avoiding this situation is a challenge for the controller. In general, these results reveal the need for self-adaptive solutions, i.e., using memory elasticity controllers.

HMC Time-Series Analysis. To show the behavior of HMC under various runtime conditions, the diagrams of Fig. 8 show results of different scenarios in which RUBBoS application has been equipped with HMC under various workloads: (i) synthetic workloads consisting open and closed user loop models (Figs. 8a and 8b); (ii) real workloads consisting Wikipedia and FIFA traces (Figs. 8c and 8d). In general, the measured RTs remain lower than or relatively close to the target values under both user loop models (see Figs. 8a and 8b) and the RTs converge to the target values immediately, mostly without being noticed, after detecting a sudden increase or decrease in workload, e.g., from the 1st interval (200 requests/ users) to the 2nd interval (500 requests/ users). This can be seen also in scenarios with real workloads (see Figs. 8c and 8d while considering Fig. 5). The used pole value for all scenarios is 0.9, but in the case of Wikipedia due to the slow, incremental nature of the workload where it is not required to have a quick reaction from the controller, so we set the pole value 0.99 empirically which lead to the lower control error for this workload.

The other important point to note is that HMC properly detects and adapts to the memory capacity required to meet the target RTs for both open and closed user loop models. Indeed, as it can be observed from the plots presented in Figs. 8a and 8b, a close observation of the results reveals that the memory allocated is slightly higher under the open user loop model compared to the closed user loop model for similar configurations (i.e., under the same workload as written on top of the first diagram in Figs. 8a and 8b). This is because the number of `Apache` processes created is slightly higher under the open than the closed user loop 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. In general, since the nature of the memory allocation is quite different with the other resource such as CPU cores in terms of responsiveness, i.e., memory is released or reclaimed by the application slowly, a memory controller should have enough time to monitor the correct reaction of the application to the change and then decide the right amount of the memory for the next control interval.

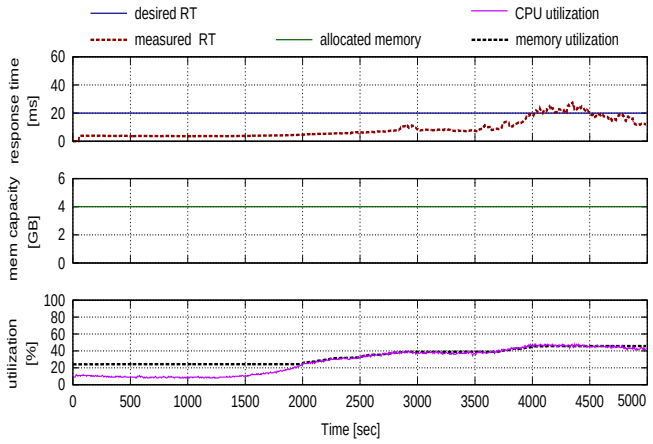
Time–Series Comparison of CMC, PMC, and HMC. To compare the behavior of the proposed controller with the two baseline controllers, Figs. 9 and 10 present the results achieved by using these two baseline controllers, while considering the results achieved by HMC (Figs. 8c and 8d) using the same benchmark application under Wikipedia and FIFA workload, respectively.

As shown in Figs. 9b and 10b, while CMC as a capacity-based controller was able to highly utilized the allocated memory, it is obviously inadequate to ensure the performance. Based on the results of Figs. 9b and 10a, the values of the measured RT exactly follow the application workload pattern (see Fig. 5 for the Wikipedia and FIFA workload patterns, respectively). This is because that capacity-based approaches are oblivious to the observed performance of services and only tries to adjust the allocated resources in a way to achieve a high resource utilization. Therefore, the important point is that the decisions of such a controller may lead to SLA violations, so it is not sufficient for performance-sensitive applications such as interactive applications.

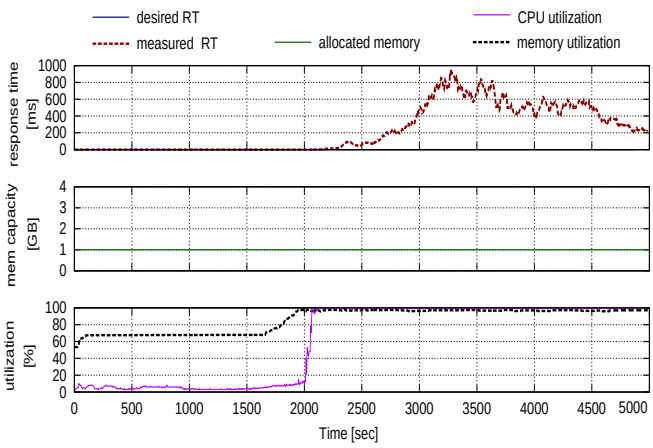
The results of PMC, Figs. 9a and 10a, on the other hand, reveals that taking the application response time as an indicator of the memory scarcity is enough to meet the application performance requirements. However, a performance-based approach such as PMC sometimes decides inefficiently as it does have any feedback regarding the resource utilization. This can lead to either over- or under-provisioning. As an example, a common problem of such approaches is when the application performance is close to the saturation point, but still the measured RT is far better that the desired RT, therefore, PMC decides to decrease the allocated memory to avoid the memory wastage. While the controller in this situation is oblivious about the memory utilization, depends on the intensity of the workload at that moment, any decrement of memory may suddenly enter the application into a memory saturation circumstance and consequently achieving a sudden peak at the measured RT and sometimes the SLA violation. For instance, this problem be seen in Fig. 9a around time 4200sec with the sudden RT peaks.

Nevertheless, results of Figs. 8c and 8d show that HMC remains stable in terms of both achieving performance targets and avoiding resource over- and under-provisioning. In compared to CMC based on the utilization plots (considering the last diagram of Figs. 8c and 9b under Wikipedia workload, and Figs. 8d and 10b under FIFA workload), while it is clear that the achieved memory utilization using HMC is relatively lower than by using CMC, the measured RT is kept under the desired RT in case of HMC. In comparison with PMC (Figs. 9a and 10a), HMC was able to meet the desired RT with less oscillations, low values for the standard deviation (SD) of RT as reported in Table I which will be discussed later, and achieving a higher memory utilization. These comparisons are more obvious when analyzing the aggregate results of the remaining of this section.

Aggregate Analysis. To assess the aggregate behaviors of HMC in comparison with PMC, and CMC over the course

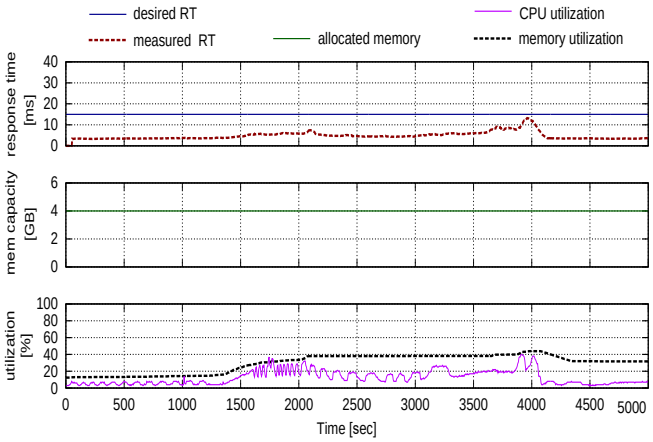


(a) **Over-provisioning** the memory, Wikipedia workload, 20ms target RT

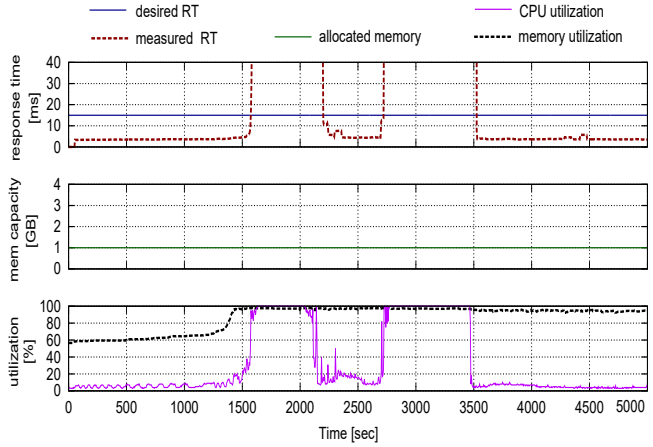


(b) **Under-provisioning** the memory, Wikipedia workload, 20ms target RT

Figure. 6: **Non-adaptive RUBBoS**, under **Wikipedia** workload with 20ms target RT.



(a) **Over-provisioning** the memory, FIFA workload, 15ms target RT

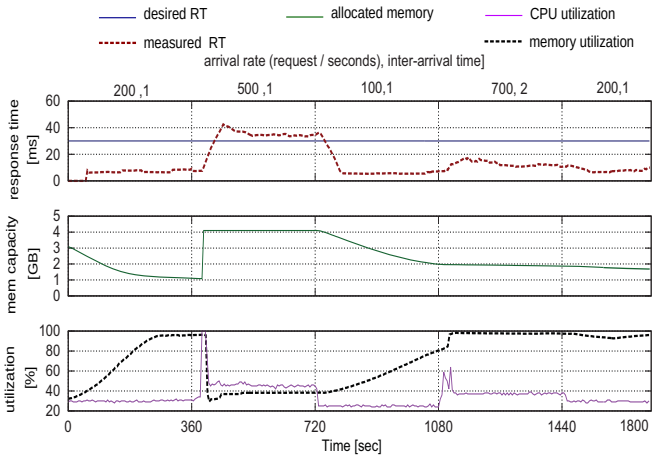


(b) **Under-provisioning** the memory, FIFA workload, 15ms target RT

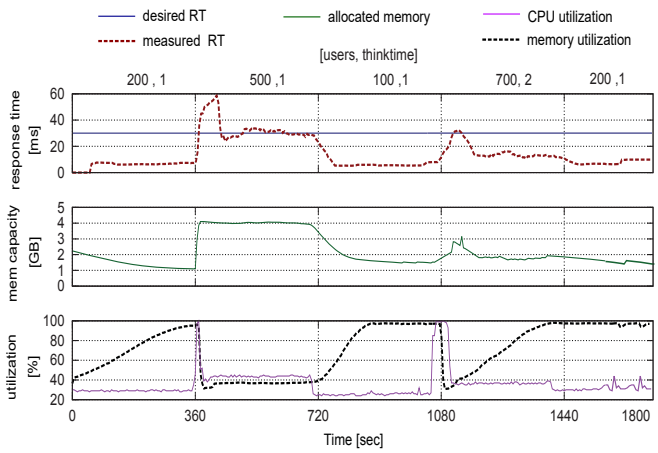
Figure. 7: **Non-adaptive RUBBoS**, under **FIFA** workload with 15ms target RT.

Table. I: **Self-adaptive** and **non-adaptive aggregate results** of RUBBoS under the two synthetic workload models, as well as Wikipedia and FIFA workloads.

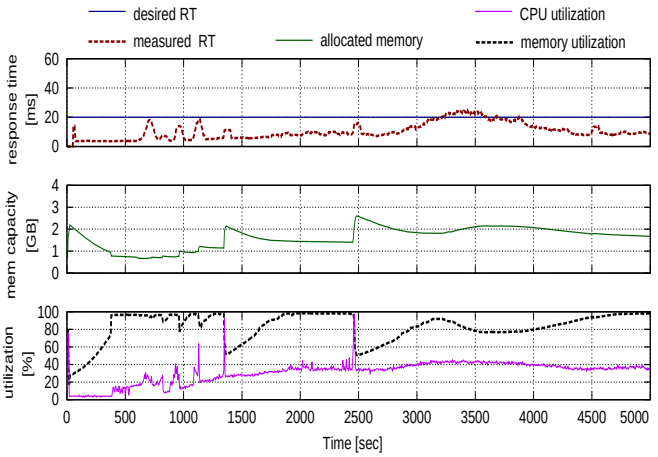
Scenario definition	**Workload** ($\tilde{rt}$)	**Controller**	**ISE**	**ISTE**	**Memory usage (mean) [GB]**	**RT (mean) [ms]**	**RT (SD) [ms]**	**U$_{cpu}$ [%]**	**U$_{mem}$ [%]**
non-adaptive	Wikipedia (20ms)	Over-provisioned	2.86	18.97	4	8.7	0.7	27.97	33.67
		Under-provisioned	94.4	85.63	1	234.0	305.35	60.45	85.97
	FIFA (15ms)	Over-provisioned	0.11	0.07	4	1.91	4.80	12.51	29.62
		Under-provisioned	126.34	231.81	1	110.77	12.36	30.99	86.88
aggregate results of HMC	open (30ms)	HMC (pole 0.9)	81.6	55.56	2.42	14.61	13.62	34.63	69.62
	closed (30ms)	HMC (pole 0.9)	82.48	55.53	2.15	13.02	11.46	34.91	68.85
controllers' comparisons	Wikipedia (20ms)	HMC (pole 0.99)	36.4	23.45	1.60	10.45	6.93	31.05	83.36
		PMC (pole 0.99)	38.13	25.33	2.32	15.0	64.73	19.47	59.74
		CMC (MOP 0.1)	30.5	22.25	2.06	176.15	173.19	42.02	90.97
	FIFA (15ms)	HMC (pole 0.9)	0.11	0.06	1.34	8.12	7.71	38.99	82.18
		PMC (pole 0.9)	0.17	0.10	2.08	11.39	11.26	18.65	57.39
		CMC (MOP 0.2)	12.13	4.25	1.36	98.99	63.33	18.02	90.93



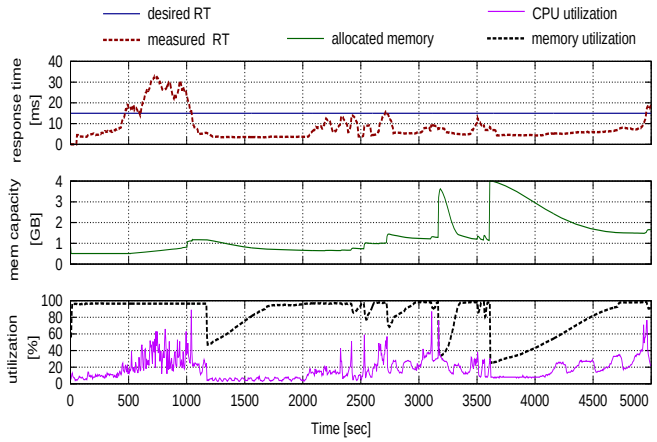
(a) **Open system model**, 30 ms target RT, 0.9 pole, interval 5sec



(b) **Closed system model**, 30ms target RT, 0.9 pole, interval 5sec

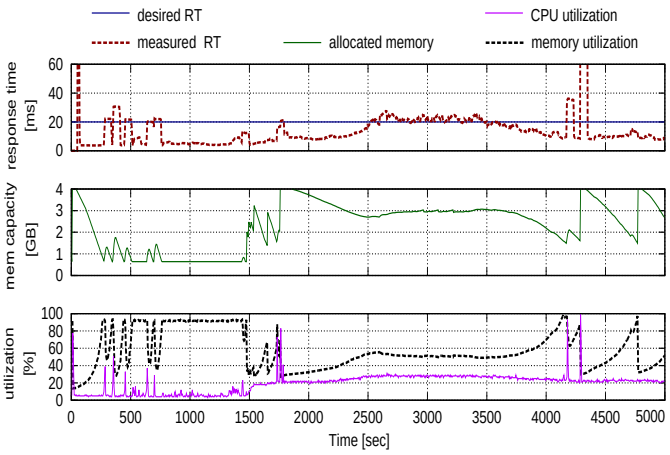


(c) **Wikipedia workload**, 20ms target RT, **0.99 pole**, interval 5sec

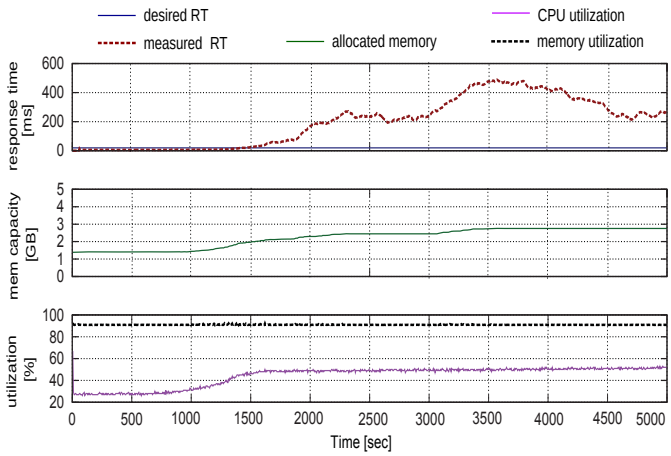


(d) **FIFA workload**, 15ms target RT, **0.9 pole**, interval 5sec

Figure. 8: **Self-adaptive RUBBoS** equipped with **HMC**, under **different workload traces** and configuration parameters.

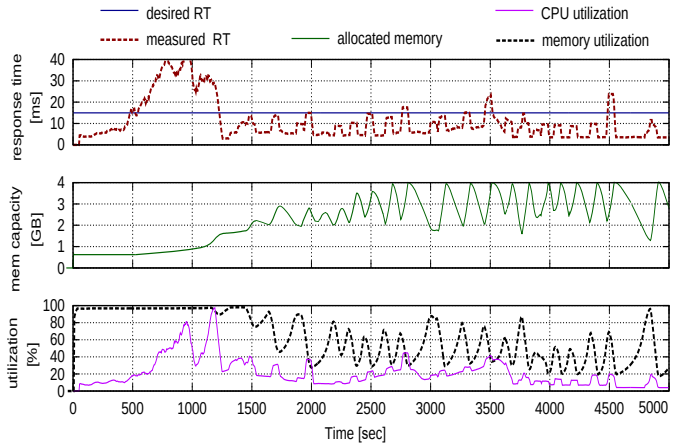


(a) **PMC**: Wikipedia workload, 20ms target RT, 0.99 pole, interval 5sec

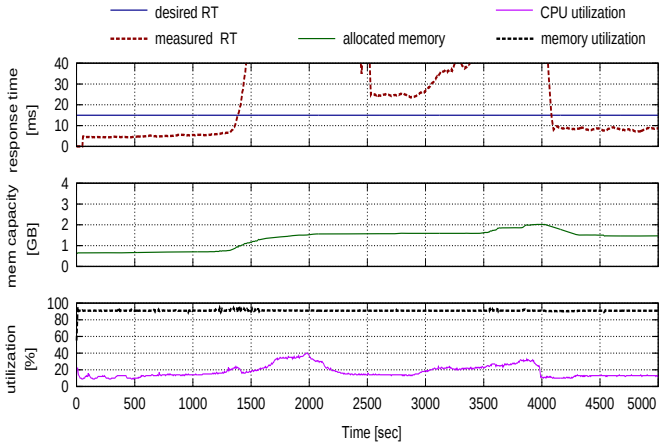


(b) **CMC**: Wikipedia workload, MOP 0.1, interval 5sec

Figure. 9: **Self-adaptive RUBBoS** equipped with **PMC** (a) and **CMC** (b), under **Wikipedia workload**.



(a) **PMC**: FIAF workload, 15ms target RT, 0.9 pole, interval 5sec



(b) **CMC**: FIFA workload, MOP 0.2, interval 5sec

Figure. 10: **Self-adaptive RUBBoS** equipped with **PMC** (a) and **CMC** (b), under **FIFA workload**.

of the experiments, in Table I, we report the mean values of the allocated memory, memory utilization, and CPU utilization, as well as the values for the mean and SD of RT for different scenarios. Moreover, we also present two control theoretic metrics, Integral of Squared Error (ISE) and Integral Squared of Timed Error (ISTE) [36], which represent the observed error during the life span of the system under test. These metrics are computed as shown in Eqs. (8) and (9) [36], where $e(t) = \tilde{rt} - rt(t)$. ISE metric reports how much the measured RT is close to the desired RT (set point). ISTE is a timed variant of the ISE, which weights the error over time and reduces the effect of the initial transient phase. ISTE would result in better values for a control theoretical solution, where the transient phase is the price to pay for modeling and controlling the system [36]. Note that although achieving a lower measured RT seems preferable from the end users' point of views, an elasticity controller should be able to achieve a measured RT that is close to the desired RT, not far better since this can implicitly indicate the resources over-provisioning.

$$ISE = \frac{1}{n}\sum (e(t))^2 \tag{8}$$

$$ISTE = \frac{1}{n}\frac{\sum (t \cdot e(t))^2}{\sum t} \tag{9}$$

Table I shows these aggregated results of the non-adaptive RUBBoS application without using any controller, as well as the self-adaptive RUBBoS equipped with different controllers under different test scenarios. As shown in this table, the first set of results is related to non-adaptive experiments under Wikipedia and FIFA workload where we have not used any of the controllers. The goal of such experiments was to show that with a static memory allocation strategy, we either should sacrifice the application performance (see under-provisioning results), or we can meet the desired performance with the expense of memory wastage to be able to handle the workload peak (see the over-provisioning results). The rational behind having a relatively high CPU utilization in the case of under-provisioning case under Wikipedia workload in compared to all other scenarios is that, when the memory is not sufficient for the application to handle the workload, the VM starts using thrashing which is a CPU intensive task.

The second set of scenarios (the second row of Table I) is related to HMC under open and closed user models. The aggregate results using HMC reveal that while HMC was able to keep the measured RT less than the desired RT, it uses reasonable memory with relatively low error values, while the memory utilization is close to 70% in these two experiments. Note that achieving a relatively low CPU utilization in all scenarios indicates that CPU in not a resource bottleneck in any of the scenarios (i.e., CPU has been over-provisioned).

The last set of experiments (the last row of Table I), compare the aggregate results of HMC to the two baseline controllers, PMC, and CMC, under the Wikipedia and FIFA workload with similar test conditions. A controller is said to be better if the desired RT is met with less error and without memory over-provisioning. In the case the three first experiments under Wikipedia workload, while HMC uses the lease amount of memory (1.60GB), it could meet the application performance target with low control errors and high memory utilization (83.36%). Although among the all, CMC achieved the highest memory utilization (90.97%), the measured application RT is very high (173.19ms) compared to the desired RT (20ms), and it used even more memory (2.06GB) than the proposed HMC. In the case the last three experiments under FIFA workload, similar results were achieved. HMC uses the least amount of memory (1.34GB), while meeting the application performance target with the lowest control errors and relatively high memory utilization (82.18%). Again, among all the controllers CMC achieved the highest memory utilization (90.93%), but the desired RT was violated (98.99ms) compared to the desired RT (15ms), while using almost the same memory (1.36GB) as the proposed HMC.

Figures 11 and 12 respectively visualises the aggregated results presented in Table I under Wikipedia and FIFA workloads. Specifically, each of these figures depicts the aggregate results obtained under the respective workloads for 5 different experiments, including self-adaptive scenarios, (using HMC, CMC, and PMC presented with reddish plots), as

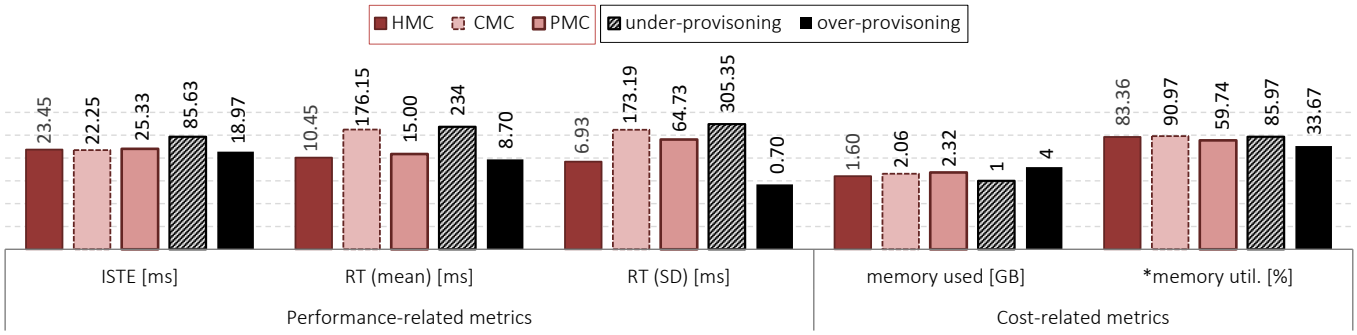


Figure. 11: **Comparison** of the **aggregate results** for **self-adaptive** RUBBoS under **Wikipedia** workload using CMC, PMC, and HMC, and **non-adaptive** RUBBoS, based on the values reported in Table I. *Note that, except the other metrics, achieving a higher value is preferred in the case of memory utilization.

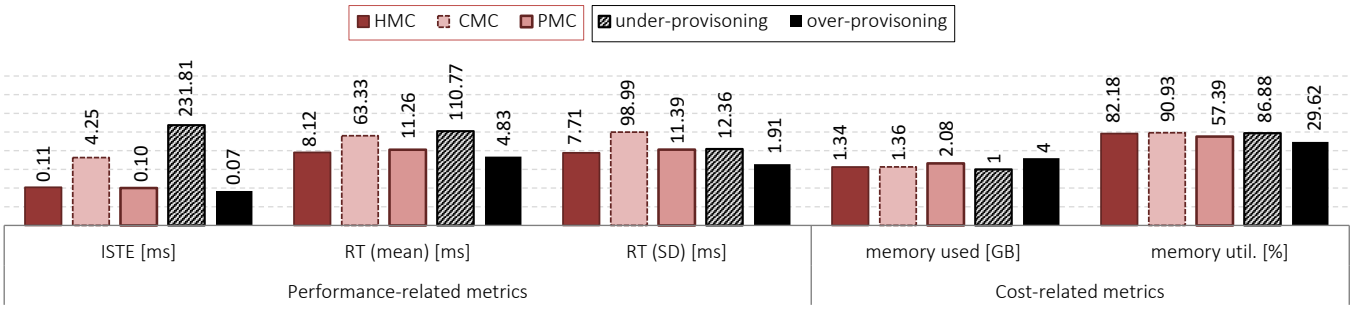


Figure. 12: **Comparison** of the **aggregate results** for **self-adaptive** RUBBoS under **FIFA** workload using CMC, PMC, and HMC, and **non-adaptive** RUBBoS, based on the values reported in Table I.

well as non-adaptive scenarios (under- and over-provisioning, presented with back plots) in terms of either performance-related metrics, or cost-related metrics. These two diagrmas are illustrated in details in next section.

VI. Discussions and Limitations

In the remainder of this section, we discuss the results, limitations, threats to the validity of the findings, some insights and future work.

A. Discussion

Recalling Figs. 11 and 12, in general, the usage of the proposed HMC controller (the dark red plots) leads to better results in both performance and cost aspects. More precisely, in the case of Wikipedia workload (see Fig. 11 achieved results using HMC compared to the results using one of the baseline controllers reveal that, while the observed control error (ISTE) over the experiment period was almost comparable, the stability in RT is less for CMC compared to HMC (173.19 SD vs. 6.93 SD) by even allocating more memory (2.06GB vs. 1.60GB), leading to SLA violation. This shows that a capacity-based approach such as CMC in spite of achieving a higher resource utilization (90.97% vs. 83.36%) compared to our proposed hybrid-approach, is oblivious to the application performance, achieving at least 10 times worse RT compared to HMC (176.19ms vs. 10.45ms), and it obviously violated the SLA (20ms).

In the case of FIFA workload (see Fig. 12, the observed control error (ISTE) over the experiment period were similar for PMC and HMC, but it is much higher for CMC. After the over-provisioning scenario, the stability in RT is the minimum for the proposed HMC (7.71) compared to other two controllers by even allocating the minimum memory. Although again CMC achieved the highest memory utilization (90.93% vs. 82.18) compared to our proposed hybrid-approach and performance-based approach, is oblivious to the application performance, violating the SLA (15ms) compared to HMC (63.33ms vs. 8.12ms).

On the other hand, a performance-based approach such as PMC could achieve almost the same measured RT compared to the proposed hybrid approach (HMC) with comparable control error under both workloads, but with higher memory usage (Wikipedia workload: 2.32GB vs. 1.60GB; FIFA workload: 2.06GB vs. 1.34GB), less memory utilization (Wikipedia workload: 59.74% vs. 83.36%; FIFA workload: 57.93.74% vs. 82.18%), and much less stability in measured RT based on the value of SD under Wikipedia workload (64.73ms vs. 6.93msc) and comparable values in case of FIFA workload (11.39ms vs. 8.12ms). The more stability was achieved by the hybrid approach due to the insight that it has into resource utilization in comparison to a performance-based approach that only decides based on the application performance, i.e., response time, at runtime. That is why HMC achieved the best stability (i.e., the lowest SD value) in RT among all other baseline controllers.

Results related the comparison between two non-adaptive scenarios and self-adaptive RUBBoS equipped with our proposed hybrid controller (HMC) show the benefit of leveraging elasticity controllers that dynamically adjust the allocated

resources, rather than allocating static amount of resources. Compared to HMC, at the over-provisioning scenario, the memory utilization is lower (Wikipedia workload: 33.67% vs. 83.36%; FIFA workload: 29.62% vs. 82.18%), and allocating much more memory in average (4GB vs. 1.60GB–Wikipedia and 1.34GB–FIFA). While achieving the lowest average RT, which is far better than the desired RTs (20ms and 15ms), with the expense of wasting resources. On the other hand, while in the under-provisioning experiment only 1GB of memory was allocated statically with a high utilization (85.97.97%–Wikipedia, 86.88%), due to the lack of memory during the workload peak, the mean and SD of measured RT are not acceptable (see Figs. 11 and 12) and it lead to SLA violation.

In summary, the trade off between the importance of saving more on memory usage (i.e., achieving a higher memory utilization) or having less performance violation is subjective and is up to the owner of the cloud application. Moreover, the behavior of the hybrid controller under all experiments reveals that HMC behaves as intended. After identifying the system model and updating it, if it is required, at each control interval, it adjusts the right amount of memory in order to keep the measured RT close to the desired RT without over-committing the memory by considering the memory utilization as another decision making indicator along with the application response time.

B. Limitations

Reactive rather than proactive. Although the violation rate by using HMC, as a reactive approach, was very low (recalling the first plots of Figs. 8a to 8d), even this amount might be unacceptable for the industrial usages, as the money which can be saved by using such controllers may not compensate the degradation in their users' satisfactions. Therefore, using a proactive approach to address this concern would be superior. However, one can configure HMC with a value lower than the desired output. This way, before the controlled output goes outside of its desired regime, corrective action can be triggered by the controller, i.e., explicitly, the controller can behave proactively.

Using a linear regression technique to build the system model. Although as mentioned in [31], the system model used in the control formula does not necessary have to capture the exact relationship between the controller's output and the measured output, but a rough estimation is enough to tune the controller, the fact that the model rebuild mechanism is linear regression limits its applicability in highly dynamic situations such as burstly workload.

Using empirical tuning for configuring the controller rather than running sensitivity analysis. For the reported experimental scenarios, we empirically set the pole value by monitoring the control error for different value and then we chose the value which lead to the lowest error for that scenario. However, enhancing the controller to adaptively find the best value for pole can be one of the future directions.

C. Threats to Validity

Evaluation with a limited virtulized environment. Since vertical elasticity is not yet supported widely by public infrastructure providers, HMC was evaluated on a virtulized environment using Xen hypervisor (i.e., roughly similar to a private cloud). However, changing the allocated memory on-demand has been possible very recently in some public clouds such as *ProfitBricks*[15]. Therefore, evaluating HMC using public clouds services is considered as a future task.

Correlating application RT with the memory size of the VM hosting the BL tier. In order to observe the influence of the memory allocation of the VM hosting the BL tier, we set our experiments by considering: (i) assigning the proper number of CPU cores; (ii) focusing on a memory-intensive application scenario; (iii) the assignment of enough memory to the VM hosting the DS tier to be able to cache the whole application database; (iv) stressing the application by read-only requests. Since fetching required data from the cache is greatly faster than processing each request by the BL tier, the changes in the measured RT can be considered independent of the DS tier, i.e., it can be assumed that the VMs used for different application tiers behave independently.

Technical constraints of vertical elasticity. Apart from the challenges addressed throughout the controller designing process, there are still several technical constraints that should be carefully considered while focusing on the memory elasticity: (i) depending on the memory allocation strategy used in hypervisors, reducing memory size may not be beneficial for the host OS. However, in the case of Xen hypervisor and using ballooning mechanism, the released VM memory size can be used for other co-located VMs; (ii) even when the memory size can be changed at the OS level, some applications cannot still support the dynamic memory allocation and eventually need to be restarted to take advantages of the new allocated memory, such as `JVM` applications. However, we focused on `Apache` Web service that can leverage the new allocated memory in a dynamic and live manner.

VII. Related Work

The field of elastic systems in general and auto-scaling approaches, in particular has been gaining momentum in cloud computing [4], and several approaches based on different frameworks, models and techniques have been applied, turning it into a mature field. In this section, instead of reviewing the breadth of this field, which has been reviewed in [37, 38], we summarize the work on vertical elasticity in cloud computing, and then carefully look at the most relevant control-theoretic approaches that have been applied to enable auto-scaling of cloud applications. By following this approach, not only can we position our work into this narrow filed in cloud computing, but we can also provide a clear view of the impact of control theory in this field.

A. Vertical Elasticity Approaches

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 memory, we mainly review work related to vertical memory elasticity and briefly explore the efforts for other resources such as CPU. The research work in the area of vertical elasticity is presented in two categories: capacity-based and performance-based approaches.

[15] www.profitbricks.com/help/Live_Vertical_Scaling

Capacity-based approaches. Baruchi et al. [22] compare two techniques for memory elasticity: (i) based on the concept of EMA, which was also used in our work; (ii) based on Page Faults. They experimentally show that when Page Faults are used to scale memory, the performance is improved in comparison with the EMA technique. Dawoud et al. [23] propose the concept of Elastic VM that supports dynamic resource scaling feature without rebooting the system. They experimentally demonstrated that Elastic VM architecture requires less consumption of resources and avoids scaling-up overhead while guaranteeing SLAs. The authors claim it is more suitable for memory scaling with lower costs and complexity. The authors of [10] use elasticity rules to adapt the VM memory size to the application requirements. A mechanism is proposed to monitor the VM memory and apply vertical elasticity rules in order to dynamically change its memory size by using the memory ballooning technique provided by KVM hypervisor. Similar to us, they show that it is possible to adapt the VM memory size while maintaining the performance level of the running application. Molto et al. [10] present a mechanism for adapting the VM memory size to the memory consumption pattern of the application by using a simple elasticity rule. Kalyvianaki et al. [39] supports vertical elasticity by adopting Kalman filtering and statistical approaches to track and control the CPU utilization in virtualized environments in order to guide capacity allocation. In [40], the authors use two layers of controllers, one to regulate the relative utilization for each tier of a RUBBiS Web application, and a second one to further adjust the allocations in cases of CPU contention. Baruchi et al. [22] compare two techniques for memory elasticity: exponential moving average (EMA), and page faults. They demonstrate that scaling memory using page faults improves the performance as compared to the EMA technique.

Performance-based approaches. The authors of [5] and [15] 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 [14] the authors propose an autonomic resource controller consisting two controllers for CPU and memory vertical elasticity, and a higher level fuzzy controller as a coordination between these two controllers. Their proposed approach considers the application performance as the main elasticity reasoning and resource utilization to infer the degree of contribution of each resource on application performance change. Spinner et al. [27] proposes a proactive vertical memory approach which takes the application performance and the change at the workload in order to adjust the allocated memory at runtime. In [41] the authors present a performance model which captures the relationship between the CPU allocation and the application performance is automatically extracted and updated online using resource demand estimation techniques. This model is then used in a feedback controller to dynamically adapt the number of virtual CPUs of individual VMs.

B. Control-Theoretic Approaches on Auto-scaling

Controller synthesis. Our work is inspired by the idea of simple yet general controller synthesis proposed in [31, 32] that reduces the need for strong mathematical background to devise ad-hoc control solutions. There are some approaches based on control theory (e.g., [35, 42]) that can enhance cloud applications with the capability to adjust their resources based on changing environmental conditions. These approaches typically synthesize an elasticity controller to automatically decide when to activate some optional features. The benefit of such approaches is that they allow guaranteeing some specific desirable properties. Although such controllers are resilient against stationary noises, they are proved to be robust against non-stationary uncertainties.

Classic control. Some approaches (e.g., [30, 43, 44]) employ class control techniques, such as PID, to construct autonomic controllers for adjusting resources. Other work use feedback control to realize power management [45, 46] to guarantees power consumption while maximizing performance subject to the power budget. These controllers depend on simple mathematical models and provide formal guarantees on the properties of the controller, but they need to consider some assumptions that constrain their adoption in highly dynamic and volatile environments such as clouds.

Advanced control. Other approaches (e.g., [34, 36, 47]) try to build on classic control theory or classic queuing models, by proposing parametric models in which part of the parameters are unknown at design time and can be adjusted by adopting adaptive filters such as Kalman filtering.

Adaptive control. Adaptive control addresses some of the shortcomings of fixed gain controllers by dynamically estimating the model parameters and adjusting the gains of the controller to better estimate the set point. Therefore, changes in the system model are detected on the fly and incorporated into the controller. A relevant example of such adaptive controller has been employed in [48].

Knowledge-based control. The main difference between the traditional model-based control theory approaches and knowledge-based control approaches is that model-based approaches assume that a precise mathematical model of the system to be controlled is explicitly available. Whereas, the knowledge-based control does not make such an assumption, but rely on expert knowledge [4]. Deriving an accurate mathematical model of the underlying software is a daunting task due the non-linear dynamics of real systems [30, 49]. Fuzzy control is a known knowledge-based control approach which has been applied for dynamic resource allocation in cloud [50]. 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. Although this facilitates knowledge elicitation from users, but such approaches are still dependent on users' inputs. Some approaches tackle this problem by entangling the fuzzy control with machine learning techniques [51, 52].

Black box control. Classic control approaches rely on the use of mathematical models that is inherently limited to the domain where it is possible to accurately define a model structure and estimate model parameters. Black box and surrogate models address this challenge by constructing the models from input-output data collected over time, and thus obtaining models that resemble the system by construction. Interestingly, such black box approaches have been adopted quite a lot in the context of cloud auto-scaling, e.g., [53, 54].

Online learning approaches. Some other established techniques, such as machine learning, have been exploited to enhance classic controllers. This approach allows the control solutions to deal with unseen and emerging behaviors that may differ from the design-time assumptions. Machine learning approaches can be categorized as model-based and model-free, depending on the use of analytical models. The most popular model-based approaches use artificial neural networks (ANN) [36], while popular model-free techniques use clustering to discover new control rules [50]. In model based approaches, the accuracy of the control actions is proportionally related to the model structure and the training data [36]. In model-free solutions, the accuracy depends on the learning rate and the size of the action-configuration space [38].

C. Concluding remarks

The literature on auto-scaling is abundant. However, our approach has several distinguishing aspects: first, because of the special challenges in memory elasticity, research on this topic is scarce compared to other resource elasticity research; second, among the work exploring memory elasticity, most of them [55, 56] look at the DS tier, as the effect of memory on retrieving data is clear; therefore, being concerned with the memory elasticity of the BL tier has not yet been well investigated; third, our proposed hybrid approach consider both application performance and resource utilization as decision making criteria to scale up or scale down the memory in order to leverage the benefits of both performance-based approaches and capacity-based approaches.

VIII. Conclusion

This paper proposes a hybrid vertical memory elasticity controller that leverages the benefits of both performance-based and capacity-based elasticity approaches. The hybrid controller was designed by using the concept of the feedback control loop. It scales up or down the allocated memory as a control knob, and takes the application performance and VM memory utilization as feedback parameters to satisfy the application performance constraints in spite of the varying workload. For the evaluation, the results achieved by the hybrid controller are compared in an experimental setup with the results of a performance-based controller and a capacity-based controller using RUBBoS as an interactive benchmark application deployed in a virtualized environment using Xen hypervisor under synthetic and real workloads including Wikipedia and FIFA. The results reveal that the hybrid controller achieves a relatively high memory utilization (close to 83%), while allocating the lowest amount of memory, and having a high performance stability (i.e., standard deviation of response time) compared to the two baseline controllers.

Generally speaking, such a controller can be used to make a cloud application self-adaptive (memory-wise) and to guarantee the desired performance while decreasing the cost in terms of resource usage, i.e., achieving a high resource utilization, for the application owner. We envision the future work as follows: (i) improving *hybrid memory controller* by using techniques such as Kalman Filtering to more precisely rebuild the system model and hereby be able to more robustly handle the workload fluctuations; (ii) considering the application workload prediction to proactively realize the memory elasticity.

Acknowledgment

The authors would like to thank Martina Maggio for her constructive comments on the control related content of the paper, Toni Mastelić for his technical points regarding the experiments, and Cristian Klein for his valuable points regarding the memory virtualization. This work was partially supported by the doctoral college "Adaptive Distributed Systems", 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.

References

[1] F. Nah, "A Study on Tolerable Waiting Time: How Long are Web Users Willing to Wait?" *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004.

[2] K. Eaton. (2012) How One Second Could Cost Amazon $1.6 Billion In Sales. Available online: http://www.fastcompany.com/1825005/impatient-america-needs-faster-intertubes/.

[3] J. Grossklags and A. Acquisti, "When 25 Cents is Too Much: An Experiment on Willingness-To-Sell and Willingness-To-Protect Personal Information," in *Workshop on the Economics of Information Security (WEIS)*, 2007.

[4] P. Jamshidi, A. Ahmad, and C. Pahl, "Autonomic resource provisioning for cloud-based software," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2014, pp. 95–104.

[5] 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.

[6] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What it is, and What it is not," in *International Conference on Autonomic Computing (ICAC)*, 2013, pp. 23–27.

[7] L. Schubert, K. G. Jeffery, and B. Neidecker-Lutz, *The Future of Cloud Computing: Opportunities for European Cloud Computing Beyond 2010*. European Commission, 2010.

[8] Neovise. (2013) Second-generation Cloud Computing - IaaS Services, What It Means, and Why We Need It Now.

[9] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "The Resource-as-a-service (RaaS) Cloud," in *Conference on Hot Topics in Cloud Computing*, 2012, p. 12.

[10] G. Moltó, M. Caballer, E. Romero, and C. de Alfonso, "Elastic Memory Management of Virtualized Infrastructures for Applications With Dynamic Memory Requirements," *Procedia Computer Science*, vol. 18, pp. 159–168, 2013.

[11] 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," *SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[12] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Setection and Bottleneck Identification," *Computing Surveys (CSUR)*, vol. 48, no. 1, p. 4, 2015.

[13] (2014) Rubbos. Available : http://jmob.ow2.org/rubbos.html.

[14] S. Farokhi, E. B. Lakew, C. Klein, I. Brandic, and E. Elmroth, "Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints," in *International Conference on Cloud and Autonomic Computing (ICCAC)*. IEEE, 2015, pp. 69–80.

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

[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] Wikipedia Access Traces. Available online: http://www.wikibench.eu/?page_id=60, Last visit 2016-02-10.

[18] FIFA 1998 Web site Page View Statistics. Available online: http://ita.ee.lbl.gov/html/contrib/WorldCup.html, Last visited 2016-02-10.

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

[20] Apache Performance Tuning. Available online: http://www.devside.net/articles/apache-performance-tuning, Last visit 2016-02-10.

[21] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application Level Ballooning for Efficient Server Consolidation," in *European Conference on Computer Systems*. ACM, 2013, pp. 337–350.

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

[23] 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.

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

[25] E. B. Lakew, "Autonomous Cloud Resource Provisioning: Accounting, Allocation, and Performance Control," Ph.D. dissertation, Umeå University, Department of Computing Science, 2015.

[26] (2015) Metric Choice Matters for Intelligent Autoscaling. Available online: http://elastisys.com/2015/08/24/metric-choice-matters-for-intelligent-auto-scaling/.

[27] S. Spinner, N. Herbst, S. Kounev, X. Zhu, L. Lu, M. Uysal, and R. Griffith, "Proactive Memory Scaling of Virtualized Applications," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2015, pp. 277–284.

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

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

[30] 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.

[31] A. Filieri, H. Hoffmann, and M. Maggio, "Automated Design of Self-Adaptive Software with Control-Theoretical Formal Guarantees," in *International Conference on Software Engineering (ICSE)*, 2014.

[32] A. Filieri, M. Maggio, and et. al, "Software Engineering Meets Control Theory," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2015.

[33] S. Farokhi, "Quality of Service Control Mechanisms in Cloud Computing Environments," Ph.D. dissertation, Vienna University of Technology, Faculty of Informatics, 2016.

[34] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, Model-driven Autoscaling for Cloud Applications," in *International Conference on Autonomic Computing (ICAC)*. IEEE, 2014, pp. 57–64.

[35] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez, "Brownout: Building More Robust Cloud Applications," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 700–711.

[36] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva, "Comparison of Decision-making Strategies for Self-optimization in Autonomic Computing Systems," *Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 7, no. 4, p. 36, 2012.

[37] G. Galante and L. C. E. d. Bona, "A Survey on Cloud Computing Elasticity," in *IEEE Conference on Utility and Cloud Computing (UCC)*, 2012, pp. 263–270.

[38] A. Gambi, G. Toffetti, and M. Pezzè, "Assurance of Self-adaptive Controllers for the Cloud," in *Assurances for Self-Adaptive Systems*. Springer, 2013, pp. 311–339.

[39] 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 (ICAC)*. IEEE, 2009, pp. 117–126.

[40] 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.

[41] 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 *International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE, 2014, pp. 157–166.

[42] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck, "From Data Center Resource Allocation to Control Theory and Back," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2010, pp. 410–417.

[43] M. Maggio, C. Klein, and K. Arzen, "Control Strategies for Predictable Brownouts in Cloud Computing," in *International Federation of Automatic Control (IFAC)*, 2014.

[44] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, Robust Capacity Management for Multi-tier Data Centers," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 4, p. 14, 2012.

[45] H. Hoffmann and M. Maggio, "PCP: A Generalized Approach to Optimizing Performance under Power Constraints Through Resource Management," in *International Conference on Autonomic Computing (ICAC)*, 2014, pp. 241–247.

[46] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-aware Computing," in *ACM SIGPLAN Notices*, vol. 46, no. 3. ACM, 2011, pp. 199–212.

[47] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using Control Theory to Achieve Service Level Objectives in Performance Management," *Real-Time Systems*, vol. 23, no. 1-2, pp. 127–141, 2002.

[48] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A Multi-model Framework to Implement Self-managing Control Systems for QoS Management," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2011, pp. 218–227.

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

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

[51] 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)*. IEEE, 2011, pp. 1–9.

[52] P. Lama and X. Zhou, "Autonomic Provisioning with Self-

adaptive Neural Fuzzy Control for Percentile-based Delay Guarantee," *Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 2, p. 9, 2013.

[53] G. Toffetti, A. Gambi, M. Pezzé, and C. Pautasso, *Engineering Autonomic Controllers for Virtualized Web Applications*. Springer, 2010.

[54] A. Gambi, G. Toffetti, C. Pautasso, and M. Pezze, "Kriging Controllers for Cloud Applications," *Internet Computing*, vol. 17, no. 4, pp. 40–47, 2013.

[55] A. Gupta, E. Ababneh, R. Han, and E. Keller, "Towards Elastic Operating Systems," in *Conference on Hot Topics in Operating Systems*, 2013, p. 16.

[56] H. C. Lim, S. Babu, and J. S. Chase, "Automated Control for Elastic Storage," in *International Conference on Autonomic Computing (ICAC)*. IEEE, 2010, pp. 1–10.