LOOPS: A Holistic Control Approach for Resource Management in Cloud Computing

Auday Al-Dulaimy
Mälardalen University, Västerås, Sweden
auday.aldulaimy@mdh.se

Javid Taheri
Karlstad University, Karlstad, Sweden
javid.taheri@kau.se

Alessandro V. Papadopoulos
Mälardalen University, Västerås, Sweden
alessandro.papadopoulos@mdh.se

Thomas Nolte
Mälardalen University, Västerås, Sweden
thomas.nolte@mdh.se

ABSTRACT

Resource sharing among a set of virtual machines (VMs) which are co-hosted on heterogeneous physical machines (PMs) introduces major benefits for improving resource utilization and total cost of ownership, but it can create technical challenges on the running performance. In practice, orchestrators are required to allocate sufficient physical resources to each VM to meet a set of predefined performance goals. To ensure a specific service level objective, the orchestrator needs to be equipped with a dynamic tool for assigning computing resources to each VM, based on the run-time state of the target environment. To this end, we present *LOOPS*, a multi-loop control approach, to allocate resources to VMs based on the service level agreement (SLA) requirements and the run-time conditions. *LOOPS* is mainly composed of one essential unit to monitor VMs, and three control levels to allocate resources to VMs based on requests from the essential node. A tailor-made controller is proposed with each level to regulate contention among collocated VMs, to reallocate resources if required, and to migrate VMs from one host to another. The three levels work together to meet the required SLA. The experimental results have shown that the proposed approach can meet applications' performance goals by assigning the resources that the applications require.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

cloud computing, resource management, auto-scaling, vertical scaling, horizontal scaling, VM migration.

ACM Reference Format:
Auday Al-Dulaimy, Javid Taheri, Alessandro V. Papadopoulos, and Thomas Nolte. 2021. LOOPS: A Holistic Control Approach for Resource Management in Cloud Computing. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPE '21, April 19–23, 2021, Virtual Event, France
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8194-9/21/04…$15.00
https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Cloud computing has gained significant attention in the last decade due to the wide range of services it offers. Cloud computing services are offered by providing an access to a wide range of infrastructures hosted on cloud data centers. The resources are available on-demand in a pay-as-you-go manner. The resources' demand of many applications is not static and varies over time. Cloud users select pre-configured VMs from a set of VM types provided by cloud providers to serve the applications. Thus, from one side, cloud users do not know the performance that they get when they pick a given resource configuration. They can either select the least possible configuration for their applications, which results in performance degradation, or they can acquire VMs based on the application peak demand to achieve high application performance. However, peak load resource allocation leads to resource wastage, and in this case, users pay for resources that would not fully utilize. On the other side, cloud providers try to consolidate VMs on a minimal number of physical servers based on the virtualization concept, so as to use the available physical resources efficiently and minimize the running cost. Despite its benefits, performance degradation and Service Level Agreement (SLA) violations can also occur for the provided services due to contention among collocated VMs. Requirements of cloud-based applications from the physical resources are different, and they may vary for the same application over time. Meeting specific metrics can be achieved by monitoring the system status, understanding the contention among co-hosted VMs, and accordingly reallocating the physical resources among the VMs.

This work presents an auto-scaling approach for controlling contention among co-hosted VMs, and reallocating resources dynamically based on the resources' demand of the cloud services. The proposed approach, called *LOOPS*, consists of two main components: (i) the Monitoring and Measurement Unit (MMU) to monitor the VMs, and (ii) the adaptation level. The adaptation level is composed of three controllers, each operating under the orchestration of the MMU. The three levels are in charge of regulating contention among co-hosted VMs, VM autoscaling, and VM migration. *LOOPS* enforces the different loops coordination via the MMU.

The key contributions of this work are: (i) Design and implementation of a multi-loop control approach to allocate the required resources for cloud-based applications. (ii) Introduction of an component to coordinate and control the work of the proposed loops.

2 RELATED WORK

Many auto-scaling techniques already tackled the problem of dynamic resource allocation in two directions Vertical Scaling (VS) and/or Horizontal Scaling (HS) using different approaches. Some works used upper and lower thresholds as conditions to do the scaling actions [4, 10, 12]. However, identifying the thresholds is a challenging task by itself. Other works adopted learning approach learn from the system and react accordingly [11, 16, 24, 29]. However, learning process is time consuming. Different works, as in [15, 17, 25, 28], employed fuzzy control to manage resources. However, Fuzzy controllers, in general, can guarantee stability and effectively of the system only if their underlying rules are properly designed. Queuing theory inspired many works to apply it for resource (re)allocation, such as [2, 5, 8, 9, 27]. In general, queuing theory technique works well with the applications of stationary characteristics. But this is not the case in cloud computing. Others works, such as [6, 7, 13, 14, 23], used time series to predict the required resources to be allocated to the application in the future. However, time series has a drawback which is the prediction accuracy. Some works adopted control theory to define the conditions needed to maintain a controlled output in the face of input variation by providing automation mechanisms for system management. In [21], the authors presented an adaptive resource control system to dynamically adjusts the resource shares to individual tiers. The classical control theory was used in the system aiming to meet the QoS goals and achieving high resource utilization in the data centres. The authors of [20] proposed a resource allocation system, called *AutoControl*, to automatically allocate resources based on the dynamic workload changes aiming to achieve application SLOs. *AutoControl* consists of two main parts: an online model estimator to estimate the required resources, and a resource controller to do the scaling actions. However, control theory approaches often need some parameters that require to be tuned offline for different applications or workloads. The feedback control systems also need a feedback signal that is stable and well correlated with the design goal measurement. Specifying appropriate feedback signals for different applications adds complexity to the systems. Also, these works only utilize VS and HS, while our work add another level of scaling. Few works depend on designing levels for resource management. In [22], the authors presented an approach for cooperative multilayered scaling. Two layers were investigated: the virtual infrastructure layer and the containers layer. The proposed approach tried to synchronized the scaling actions in a way that two layers are consider the scaling decisions of each other. The most closely related to this work is our previous work presented in [1]. It also adopted a multi-loop control approach, called *MultiScaler*, to allocate resources dynamically to ensure the SLA for cloud applications. However, the loops in *MultiScaler* work independently from each other, without coordinating their actions. It is worth mentioning that auto-scaling in cloud computing is mostly related to the Analyze and Plan parts of the well-known Monitor, Analyze, Plan, and Execute (MAPE) loop [19]. Thus, the design of *LOOPS* utilizes these two phases within a master unit to perform the resource management process in a data center environment. By coordinating with the master node, *LOOPS* 's controllers maintain the desired level of service in multiple closed loops. To the best of our knowledge, *LOOPS* is the first attempt to design an auto-scaling system with three different levels working under an umbrella of a master management unit.

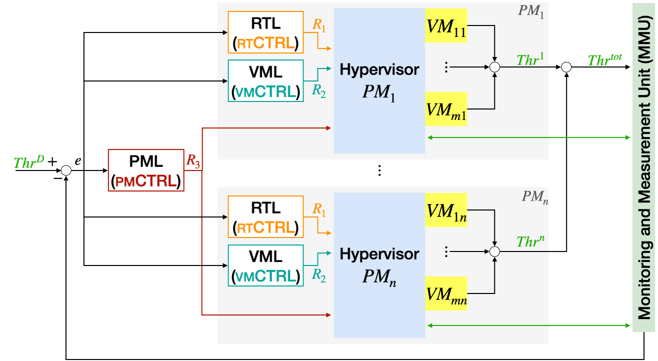

Figure 1: *LOOPS* system model.

3 THE *LOOPS* APPROACH

This section describes the system model of the proposed approach, which is described in Fig. 1, and explains how a multi-loop fashion works under the umbrella of a main monitoring unit to allocate resources to VMs based on their actual needs. The *LOOPS* approach employs a combination of multi-level resource management systems for ensuring an acceptable desired metric rate while allocating the required resources to the VMs that serve the cloud-based applications. More precisely, *LOOPS* comprises an essential unit and three levels to allocate resources efficiently for VMs, as explained in the following sections.

3.1 Monitoring and measurement unit

An essential part in *LOOPS* is the Monitoring and Measurement Unit (MMU). The unit's main functions are: (i) check the status of all running VMs and monitor some metrics, such as the VM and PM throughput, to take some measurements from them, and accordingly, (ii) coordinate the tasks of the three scaling levels by issuing requests to their close-loop controllers to do the necessary resource management actions. Those actions aim to regulate resources' shares (using 4 control strategies), scale resources (using Algorithm 1), and migrate VMs (using Algorithm 2).

3.2 *LOOPS* levels

The *LOOPS* approach works in three levels, each level has its controller. *LOOPS* separates the scopes of the three levels and investigates the levels' compatibility. The controllers are separated, but they complement each other by coordinating with MMU. They receive commands from MMU to trigger the required actions periodically. The three levels are: Resource Tuning Level (RTL), Virtual Machine Level (VML), and Physical Machine Level (PML).

3.2.1 RTL. RTL is the first level in *LOOPS* that is responsible for regulating allocated resources. The controller at this level, called the Resource Tuning Controller (RTCTRL), tunes the resources that are already allocated to the VMs. It adjusts shared resources inside each PM to regulate the contention per VM. RTL handles contention

among co-hosted VMs for obtaining shared resources. Some applications served by the co-hosted VMs need more shared resources than others. This can compromise the performance of other applications. RTCTRL can modify three variables for implementing the amount of resources each VM receives from the PM during contention; they are resource shares, resource limits, and resource reservations. **Resource sharing** is an effective way to prioritize VMs in case of contention, and thus, 'shares' can be provided appropriately and accurately. Moreover, shares must be checked and modified dynamically as they are relative to each other. Resource shares are defined for each VM to enforce the amount of resources each VM would receive in case of contention. **Resource limits** are other control variables that influence the ultimate allocation of resources to VMs. That is, VMs' limits can also be applied in addition to specifying the values of the VMs' shares, and resource limits enforce the final resource allocations regardless of the number of shares provided. **Reserving resources** is another way to ensure the performance of VMs in virtualized environments. PMs will first cut-out the amount of reserved resources to each VM and then split the rest according to 'shares' and 'limits'. VMs co-hosted on the same PM can always receive a minimum compute power of CPU allocation, regardless of other VMs in the system. The MMU unit triggers request to RTCTRL to perform the tuning action based on some measurements from the VMs. The domain of the RTCTRL is the VM, in other words, it can not perform any action outside the VM. In RTL, the four different control strategies, presented in [1], to tune and regulate the share, limit, and reservation values are investigated. Here we generalize the variables used in the formalization of the strategies. They strategies are: *Steps-Slow*, *Steps-Adaptive*, *Steps-Ratio*, and *Free-Ratio*.

The ***Steps-Slow*** control strategy defines a specific step value that will be added/subtracted to/from the current limit value. The limit of the next cycle can then be increased/decreased by RTCTRL based on the MMU decision. Thus, the limit parameter is tuned based on the following equation:

$$CPU_{c+1}^{Limit} = CPU_{c}^{Limit} \pm Steps_{Limit} \tag{1}$$

where: CPU_{c+1}^{Limit} is the CPU limit in the next cycle, CPU_{c}^{Limit} is the current CPU limit. $Steps_{Limit}$ is a hardware-dependent parameter. It can be calculated as:

$$Steps_{Limit} = round\left(\frac{CPU_{VM}^{Reservation} - (vCPUs \times CPU_{Speed})}{\eta}\right) \tag{2}$$

where: $CPU_{VM}^{Reservation}$ is the amount of reserved CPU to the VM, $vCPUs$ is the number of VM cores, CPU_{Speed} is the CPU speed, η is the number of cycles required to change the limit from the minimum to maximum value, or vise versa.

In ***Steps-Adaptive***, the value of the limit parameter in the next cycle can be adapted by RTCTRL based on the MMU decision. The value is changed as same as in the Steps-Slow control strategy, but the difference here is that it is possible to change more than one defined $Steps_{Limit}$ to the CPU_{c}^{Limit} in order to get CPU_{c+1}^{Limit}. However, we restrict the number of possible change times to be from 1 to 4. In general, this strategy exhibits more aggressive behavior compared to Steps-Slow strategy, but such behavior may be important to make sure that the SLA violations can be maintained. Thus, the limit parameter in Steps-Adaptive is tuned based on the following equation:

$$CPU_{c+1}^{Limit} = CPU_{c}^{Limit} \pm \left(Steps_{Control} \times Steps_{Limit}\right) \tag{3}$$

where: CPU_{c+1}^{Limit} is the CPU limit in the next cycle, CPU_{c}^{Limit} is the current CPU limit, $Steps_{Limit}$ is a hardware-dependent parameter calculated as in (2), $Steps_{Control}$ is the the number of possible adding/subtracting times, it is calculated as follows:

$$Steps_{Control} = \frac{\left(QoS_{Target} - QoS_{Current}\right)}{\delta} \tag{4}$$

and δ is a metric-dependent parameter, in this work it is calculated as in (5)

$$\delta = QoS_{Target} \times \vartheta \tag{5}$$

where ϑ is a percentage from the desired metric, which is empirically set to 0.1 to get an acceptable number of $Steps_{Limit}$ to be added/subtracted from the current limit value.

In ***Steps-Ratio***, the limit of the next cycle can be adapted based on ratio and proportion. A proportion is written as an equation with a ratio on each side, one side represents the ratio of the current actual metric over the current limit parameter, and the other side represents the ratio of the desired metric over the unknown limit parameter value. The cross products is used to find the unknown limit parameter value. Then, after determining the specific step value ($Steps_{Limit}$) that will be added/subtracted to/from the current limit value, the unknown limit parameter value is rounded to serve as the limit value for the next cycle. In this case, the limit parameter is tuned based on the following equation:

$$CPU_{c+1}^{Limit} = \text{round}\left(\frac{QoS_{Target} \times CPU_{c}^{Limit}}{QoS_{Current}}\right) \tag{6}$$

After getting the new value to the limit, we round that value to the nearest $Steps_{Limit}$.

In ***Free-Ratio***, the limit of the next cycle is measured the same way as in the Steps-Ratio strategy, but with only one difference: the next CPU limit is not rounded because it results from ratio and proportion, while as explained before, the next CPU limit in the Steps-Ratio strategy is rounded to the nearest $Steps_{Limit}$. Thus, the parameters are tuned based on the following equation:

$$CPU_{c+1}^{Limit} = \frac{QoS_{Target} \times CPU_{c}^{Limit}}{QoS_{Current}} \tag{7}$$

Fig. 1 shows that RTL is responsible for tuning element k_1 in vector $R1 = (i, j, k_1)$, where i represents VM_i, j represents PM_j, and $k_1 = (\overrightarrow{C}, \overrightarrow{M}, \overrightarrow{D}, \overrightarrow{B})$ is a vector itself which encompass all parameters that RTCTRL can tune in the system, where:

$$\overrightarrow{C} = (CPU^{share}, CPU^{limit}, CPU^{reservation})$$

$$\overrightarrow{M} = (Memory^{share}, Memory^{limit}, Memory^{reservation})$$

$$\overrightarrow{D} = (Disk^{share}, Disk^{limit}, Disk^{reservation})$$

$$\overrightarrow{B} = (BW^{share}, BW^{limit}, BW^{reservation})$$

3.2.2 VML. This level includes the Virtual Machine Controller (VMCTRL) that performs vertical scaling by adding/removing resources or performs horizontal scaling by adding/removing VMs

Algorithm 1 The VML algorithm.

```
1: function VMSwitchingOnOff(Set LIST_PM , Set LIST_VM)
2:   Thr^D ← Desired threshold
3:   Thr^A ← Actual threshold
4:   Thr^Diff ← | Thr^D − Thr^A |
5:   VM^active ← The number of active VMs
6:   ThrPerVM ← floor (Thr^A/VM^active)
7:   if Thr^D >= 1.1 × Thr^A then
8:     VM^number ← ceil (Thr^Diff/ThrPerVM)
9:     SwitchOn (VM^number, VM)
10:  end if
11:  if Thr^D <= 0.9 × Thr^A then
12:    VM^number ← 1
13:    SwitchOff (VM^number, VM)
14:  end if
15: return Updated Set LIST_VM
16: end function
```

from the system. As shown in Fig. 1, VML is responsible for modifying element k_2 in vector $R2 = (i, j, k_2)$, where i represents VM_i, j represents PM_j, and k_2 is:

- Either a vector which encompass all parameters that vmCTRL can scale vertically, and it is represented in the system as $k_{2VS} = (Core^{number}, Memory^{size}, Disk^{size}, BW^{capacity})$. In this case, *LOOPS* can vertically scale the parameters of k_{2VS} (For example: add/remove cores to VMs).
- Or an integer value representing the variation of number of VMs that vmCTRL can scale horizontally, and it is represented in the system as $k_{2HS} \in \mathbb{Z}$, $\mathbb{Z} = \{0, \pm1, \pm2, \ldots\}$.

In this case, *LOOPS* can horizontally scale the number of VMs (For example: add/remove VMs to/from the system). However, this work adopts HS as most hypervisors do not support online vertical scaling, i.e., k_2 is always an integer value and it represented as k_{2HS}. The MMU unit triggers request to vmCTRL to perform the HS action based on some measurements from the VMs. The domain of the rtCTRL is the PM, in other words, it can add/remove VM(s) to/from the hosted VM. Algorithm 1 describes how the VM placement problem is solved in this work.

3.2.3 *PML.* To avoid any possible performance degradation, the system is required to make decisions at runtime, such as resources scaling decisions, as in RTL and VML. VM migration also involves making important decision to avoid such a degradation. PML includes the Physical Machine Controller (pmCTRL) that performs the dynamic VM placement from one host to another based on metrics received from the PM itself rather than the VMs co-hosted on the PM. Fig. 1 shows that PML is responsible for specifying a value for element k_3 in vector $R3 = (i, j, k_3)$, where i represents VM_i, j represents PM_j, and k_3 is an index for the destination $PM_j \in PM$ where VM_i is selected by pmCTRL to migrate to.

The PMs are represented as a set of m PMs in the cloud data center: $PM = \{PM_1, PM_2, \ldots, PM_m\}$. Each PM has a limited capacity (PM_j^{Cap}) of the following resources: processing core(s), memory, disk, and bandwidth. The VMs are represented as a set of n VMs hosted on PM in the data center: $VM = \{VM_1, VM_2, \ldots, VM_n\}$. The requirements of an instance VM_i from PM_j can be represented as ($VM_{i,j}^{Req}$). The VM placement can be considered as a bin packing problem, which is NP-hard. Thus, it is reasonable to apply a heuristic. This is solved in many works in the literature, such as in [3] and [18]. In this work, there is a slight difference in solving the VM problem. We are aiming to minimize a specific value Thr^{Diff}, which is the absolute difference between two values: Thr^A and Thr^D which are representing the actual and desired throughput, respectively, *i.e.*, $Thr^{Diff} = | Thr^A - Thr^D |$. The VM placement process considers the following constraints:

$$\sum_{i=1}^{n} VM_{i,j}^{Req} \times x_{i,j} \leq PM_j^{Cap} \tag{8}$$

$$\sum_{\forall i,j} x_{i,j} = 1 \tag{9}$$

The constraint in (8) ensures that the capacity condition is met for all VMs co-hosted on PM_j. The VM placement process in this work requires more than a single capacity constraint in selecting the destination node. The considered resources are: cpu/core, memory, disk, and bandwidth. The constraint in (9) ensures that each VM is hosted only on one PM. The decision variable x is equal to 1 if the VM is assigned to the node, otherwise it is 0.

The VM placement process adopted in this work follows a distributed model, in which the problem is divided into three phases:

Source host detection: This phase determines the need to migrate one or several VM from a selected host when it is not optimally utilized. In this work, the host with the highest CPU usage is considered the source host for VMs migration.

VM selection: This phase selects the VM(s) that would be migrated from the source host selected in the detection phase. This work selects the VM with the highest CPU usage for migration.

Destination host detection: Placing VM(s) selected form the VM selection phase on other active, or reactivated hosts. This phase is responsible for placing the migrated VMs on the host with the lowest CPU usage. The phase finds a PM_j to be the destination host to migrate a VM to.

The MMU triggers request to pmCTRL to perform the VM migration processes based on some measurements from the PMs, within the datacenter. Algorithm 2 describes how the VM placement problem is solved in this work.

3.3 Coordinating scaling actions using MMU

MMU identifies a decision vector $D = (L1, L2, L3)$, to demonstrate the relationship between intervals in the activation time of the three levels. These intervals are represented as cycles. L1, L2, and L3 are decision variables to be set to ‘0’ or ‘1’ by MMU at each cycle. The variable is set to ‘1’ if its corresponding level is activated, otherwise it is ‘0’. On each cycle, L1, L2, and L3 represent activating RTL, VML, and PML in the system respectively. The cycle interval is variable and can be changed and tuned. In general, short intervals would result in repeated scaling actions and would ultimately compromise the system’s reliability and increase system overheads. At the same time, longer intervals also stalled reaction to the system requirements and prevent it from taking prompt scaling actions. Therefore, it should be selected carefully. MMU is responsible for assigning the values of D at the beginning of each cycle:

$$D = (L_1, L_2, L_3) \tag{10}$$

where $\forall i \in \{1, 2, 3\} : \ L_i \in \{0, 1\}$.

Algorithm 2 The PML algorithm.

```
1: function VMPlacement(Set LIST_PM , Set LIST_VM)
2:   Sort LIST_PM by PMs CPU_%
3:   PM_SRC ← The PM with the highest CPU_%
4:   if | Thr^D − Thr^A |≥ 0.1 × Thr^D then
5:     PM_DST ← PM with lowest CPU_%
6:     SRC_vmLIST ← all VMs ∈ LIST_VM on PM_SRC
7:     Sort SRC_vmLIST by VMs CPU_%
8:     VM_SELECTED ← The VM with the highest CPU_%
9:     if the configuration of VM_SELECTED fits in PM_DST then
10:      Migrate (VM_SELECTED, PM_SRC, PM_DST )
11:    else
12:      Switch on new PM_new
13:      PM_DST ← PM_new
14:      Migrate (VM_SELECTED, PM_SRC, PM_DST )
15:    end if
16:  end if
17: return VM placement
18: end function
```

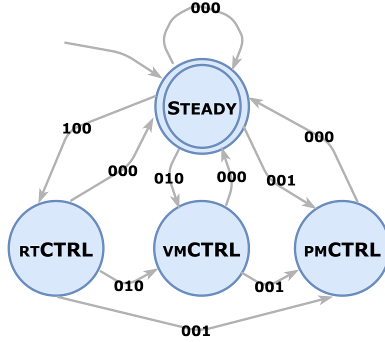

Figure 2: The state diagram of the MMU and the controllers.

Eq. (10) illustrates that MMU can issue one out of eight possible requests at each cycle, as there are three levels and two values for L_i. For example, when *D = (1,0,0)* at specific cycle number, say cycle 10, then at cycle 10 only RTL will be activated. And when *D = (1,1,0)* at specific cycle number, say cycle 20, then at cycle 20 RTL and VML will be activated. And this is the case for all other eight possible options. *LOOPS* is designed in a way that, at the beginning of each cycle, it checks and decides to activate or ignore activating the levels in the following sequence: (1) RTL, (2) VML, and finally (3) PML. In other words, MMU consecutively decides if there is a need to tune resources allocated to VMs (activate/skip RTL), then, it decides if it is required to switch on/off VMs or not (activate/skip VML), and finally, it decides if migrating VMs between PMs can enhance the system performance or not (activate/skip PML).

3.4 Model formulation

As explained in Fig. 2, the activation of the three controllers is triggered by a finite state automaton, that is described by the 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$.

Q = {Steady, rtCTRL, vmCTRL, pmCTRL} is a finite set of states. The elements of this set represent the controllers in the system, in addition to another state called Steady, which represents the MMU. Steady is considered as the state where there is no need to take any action (no tune/scale/migrate) by the controllers.

$\Sigma = \{000, 001, 010, 011, 100, 101, 110, 111\}$ is a finite set of alphabet symbols. The elements of this set describe the requests issued by the MMU to the controllers to take actions, that is, to tune, scale, and/or migrate VMs aiming to meet a predefined metric. In *LOOPS*, the metric is throughput. At a specific predefined cycle and according to the system status, MMU decides the required actions to be taken by the controller in order to meet the desired metric value. The decisions made after comparing the actual throughput with the desired one. Each alphabet $\in \Sigma$ is not a string of three elements, it is only one element composed of three characters. Another note to be illustrated is that, as *LOOPS* activates or ignores activating the levels in a specific order, not all the eight alphabets are used in the system. As discussed before in the previous section, the sequence applied in the system is RTL, VML, and then, PML. Thus, only four alphabets are needed to formulate the model, and Σ can be expressed as:

$$\Sigma = \{000, 001, 010, 100\}$$

Hence, instead of using:
- 011 as an input when MMU wants to activate VML and PML, the system reads two inputs in the following sequence: 010 , 001
- 101 as an input when MMU wants to activate RTL and PML, the system reads two inputs in the following sequence: 100 , 001
- 110 as an input when MMU wants to activate RTL and VML, the system reads two inputs in the following sequence: 100 , 010
- 111 as an input when MMU wants to activate RTL, VML, and PML, the system reads these three inputs 100 , 010 , 001 sequentially.

$\delta : Q \times \Sigma \rightarrow Q$ is the set of transition functions. The transaction function defines the rules for moving from one state to another. If the system has an arrow from state 1 to state 2 labeled with condition x, that means that if the automaton is in state 1 when condition x happens, it then moves to state 2. The system processes the running VMs until the end of execution. The processing begins with the start state. The automaton receives the symbols from the MMU represented as the Steady state. After reading each symbol, the system moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, the system ends the execution of VMs.

$q_0 \in Q$ is the initial state, which is set as q_0 = Steady.

$F \subseteq Q$ is the set of final states, defined as F = {Steady}.

4 PERFORMANCE EVALUATION

4.1 Experimental setup

The proposed approach is implemented in PowerShell running on a Windows 10 (64bit) system. A cloud-based application, which is Shamir's Secret Share schema, is selected to examine its performance evaluation. More information about the application can be found in [26]. The experiments were run on a VMware vSphere cloud with four PMs. Each VMware-ESXi PM had a quad-core Intel 3.4GHz CPU, 8 GB memory, 200 GB storage, and 10 Gbps network. A HTCondor platform, which is a distributed computing system, is installed to emulate a workload for our cloud environment. This platform consists of one master node (HTCondor master-node) and 10 worker nodes (HTCondor execute-node). The task of HTCondor master-node is to distribute jobs amongst the other ten HTCondor

execute-nodes which, in turn, execute the jobs allocated to them. Those nodes were VMs, each with 2 vCPUs, 2GB of RAM, 20GB of storage, and 1Gbps vNICs. In this work, once HTCondor master-node begins submitting the jobs to the HTCondor execute-nodes, they started executing the jobs till they finish the execution. During the execution, *LOOPS* monitors the system status periodically on cycles basis, as the total execution time of the submitted jobs is divided into step cycles. The cycle step time (cycle interval) is designed to be variable and can be changed and tuned. It empirically sets to 30 seconds in all experiments because any shorter period would result in repeated scaling actions and would ultimately compromise the system's reliability and increase system overheads. Longer periods also stalled reaction to the system requirements and prevent it from taking prompt scaling actions. After each cycle step, the MMU decides which level in *LOOPS* should be activated. Thus, the action was determined by the MMU to activate a specific level among the three, regardless of previous or future actions of other controllers. Different experiments were tested. In the experiments, RTL adopts the four different control strategies (*Steps-Slow*, *Steps-Adaptive*, *Steps-Ratio*, and *Free-Ratio*), which are discussed in Section 3, to tune and regulate the share, limit, and reservation values of the resources that are already allocated to VMs. In this work, we only focused on changing the "Limit" value, but the proposed control strategies at RTL can be applied to other parameters ("Share" and "Reservation") as well. Concerning the value of $Steps_{Limit}$ which is defined as a hardware-dependent parameter to be used in *Steps-Slow* and *Steps-Adaptive* strategies, it is set to 500 in the experiments. To clarify why it is set to 500, refer to (2), and to VMs configuration used in our experiments. As described earlier, each VM is of a 2-core with 3.4 GHz speed. The minimum amount of CPU we reserved to this VM is 1000, and the maximum amount of CPU we can allocate to this VM is 6800 (2 x 3.4 = 6.8). The allowed change is from 1000 to 6800, which means that the difference is 5800. In this work, we are aiming go from the minimum to the maximum amount of CPU (and vise versa) in around 10 steps. Thus the value of η is set to 10. Thus, by dividing the difference (5800) by η, the result is equal to 500 after flooring (580), which can (almost) approach the maximum amount of CPU which could be allocated to a VM ($1000 + (10 \times 500) = 6000$). Thus, in about 10 cycles, we can go from the minimum to the maximum amount of CPU limit for any VM. However, *Steps-Slow* value is variable which can be changed/adjusted in the system. We could make it higher value to approach the maximum amount of CPU we can allocate to this VM, but we noticed an aggressive system behavior when we did so in the experiments.

4.2 Results and discussion

Resulted throughput. As shown in Fig. 1, the system model in *LOOPS* takes the desired or targeted throughput (expressed as desired metric) as an input, and the output of the model as the actual throughput. To examine the resulted throughput, a hypothetical pattern with clear sudden workload bursts is used in the experiments as desired throughput patterns. The results of applying *LOOPS* indicates that the actual throughput almost matched, or above, the pattern of the desired throughput, as shown in the top part of Fig. 3. The performance of *LOOPS* converges to the

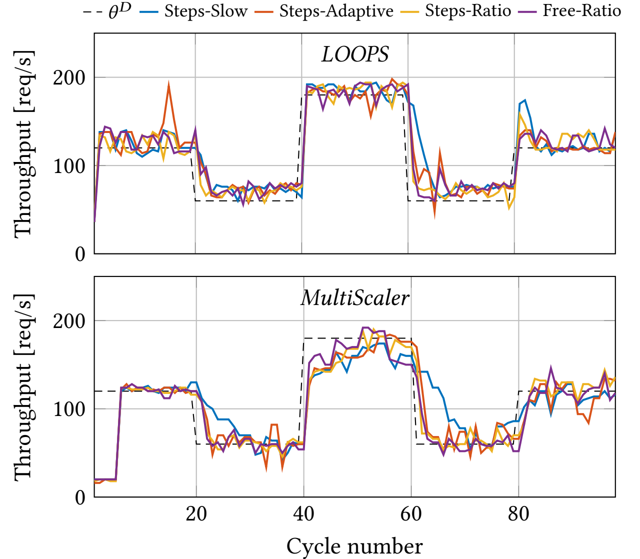

Figure 3: Resulted throughput compared with a hypothetical workload pattern using the *LOOPS* approach (top) and with the *MultiScaler* (1/5/25) approach (bottom).

optimal result (*i.e.*, matches the metric pattern) at each cycle. The convergence takes place as the MMU gives the system the ability to activate RTL, VML, and/or PML at each cycle. For examples, in the top part of Fig. 3, convergence takes place directly at cycle 1. This is because *LOOPS* can provide the necessary resources to meet the desired throughput at the beginning of each cycle.

Comparing LOOPS with MultiScaler. To validate the correctness and effectiveness of *LOOPS*, we compared it with *MultiScaler*, a closely related auto-scaling approach presented in [1]. *MultiScaler* is chosen as it is similar to *LOOPS* when it comes to the number of resource management levels, where both approaches are working within the same three closed-loops. The main difference is that *MultiScaler* has no essential component or unit to coordinate the defines specific intervals, and at each interval, only one level is activated. In a brief, *MultiScaler* proposed a V1/V2/V3 design that defines the relationship between intervals in the activation time of the three levels. These intervals are represented as cycles. V1, V2, and V3 are variables. V1 represents the first activation of RTL in the system as well as the first reactivation within the cycles; V2 represents the first activation of VML in the system and its first reactivation within the cycles; and V3 represents the first activation of PML in the system and its first reactivation within the cycles. For example, when V1/V2/V3 sets to 1/5/25 (as shown in the bottom of Fig. 3), it means that RTL is activated at cycle 1, VML at cycle 5, and PML is at cycle 25. One level is activated at each cycle. This will continue repeatedly until the end of execution. While in *LOOPS*, MMU is added to the system to make it possible to issue requests to controllers to take the required action(s) at any cycle or interval. Comparing the results shown in Fig. 3 (top) with the results shown in Fig. 3 (bottom) demonstrates the importance of

injecting the MMU to the system in order to meet the desired metric. As seen in Figure 3, the actual throughput could approach the desired throughput from the first cycle by applying *LOOPS*, this is due to the ability of MMU to trigger the necessary requests to the controllers at the same time to tune, scale, and/or migrate the running VMs. This is not the case in *MultiScaler*, as for example, what was needed in cycle 1 is switching on a number of VMs to meet the desired throughput, which happened at cycle 5 when VML activated. The convergence continues until the desired throughput value is changed. We can then see the difference again between the desired and actual throughput. Any difference in the intervals before convergence is due to the lack of resources serving the application: an issue that can be resolved when activating the three scaling levels after the desired throughput value is changed.

Applying different control strategies. Adopting different control strategies is essential to enhance the system's performance and to achieve the desired metric, especially when the resources allocated to the submitted jobs relatively suffice. Each control strategy leads to different results for the same application and within the same number of cycles. To further clarify this, let us again examine Fig. 3. When Free-Ratio control strategy is selected in RTL, the system reacts faster than it does when selecting other strategies. This is because the limit can be changed in the Free-Ratio strategy without restrictions, as it is changed proportionally to meet the system's requirements. However, such unrestricted changes require further tuning actions, a matter that makes the selection of the control strategy a crucial and precise process.

Response to workload changes. *LOOPS* features fast response to the changes, and shows stability in approaching the desired throughput, even though at some cycles produces more throughput than the desired one. This can be done by providing the required amount of resources at the beginning of each cycle. *LOOPS* reacts to the changes by taking the required action(s) at each cycle after submitting the jobs to the system. For example, in Figure 4 (top), we can see that when the desired throughput jumped from 60 to 180 at cycle 40, *LOOPS* takes the necessary actions directly at the same cycle. The number of VMs, which were switched on, notably increased at cycle 40 in order to maintain stable results by approaching the desired throughput as fast as possible. This is the case when *LOOPS* utilizes a number of VM migrations to reduce contentions, which results in performance enhancement.

4.3 Performance metrics and evaluation

For the performance evaluation of *LOOPS*, three metrics are considered. Also, we performed a pairwise t-test among the different techniques, to examine if their behavior is statistically similar, or if there is statistical evidence of one method being better than another.

Throughput error. The first metric is the **throughput error** $\Delta\theta$. Let θ_i be the throughput at iteration i produced by the system when a given method is executing, and θ_i^{D} the corresponding desired throughput. $\Delta\theta_i$ is computed as

$$\Delta\theta_i = \theta_i - \theta_i^{\mathrm{D}} \tag{11}$$

This metric quantifies the instantaneous accuracy of the considered method, with respect to the desired throughput. A positive value would reflect an overprovision of resources, while a negative value would reflect an underprovision. The ideal value for $\Delta\theta$ is zero, meaning that the method is able to always provision the right amount of resources to produce the desired throughput. Table 2 reports the p-values of the pairwise t-tests performed on the throughput error. In order to conclude that there is statistical evidence on the difference between the methods, the p-value should be lower than 0.01.Overall, the different methods perform similarly in terms of following the desired throughput. This requires a deeper analysis on the allocated resources. One method may behave statistically similar to another method in terms of Δt, but it may manage the available resources better.

Table 1: Averages and standard deviations of the Throughput Error $\Delta\theta$, and of the number of active virtual machines v, for the different methods.

Control Strategy	Avg. $\Delta\theta$	Std. $\Delta\theta$	Avg. v	Std v
Steps-Slow	11.3	26.5	6.31	2.37
Steps-Adaptive	9.5	24.5	6.1	2.27
Steps-Ratio	7.2	23.0	6.01	2.18
Free-Ratio	9.0	23.2	6.02	2.16

Table 2: t-test results for the throughput error.

Control Strategy	Steps-Slow	Steps-Adaptive	Steps-Ratio	Free-Ratio
Steps-Slow	-	0.220	0.012	0.170
Steps-Adaptive	0.220	-	0.129	0.718
Steps-Ratio	0.012	0.129	-	0.144
Free-Ratio	0.170	0.718	0.144	-

Table 3: t-test results for the number of VMs.

Control Strategy	Steps-Slow	Steps-Adaptive	Steps-Ratio	Free-Ratio
Steps-Slow	-	0.029	**0.0003**	**0.0038**
Steps-Adaptive	0.029	-	0.274	0.342
Steps-Ratio	**0.0003**	0.274	-	0.894
Free-Ratio	**0.0038**	0.342	0.894	-

Number of VMs. The second metric that we consider is associated with the number of VMs v_i. Such quantity can be monitored by considering an initial number of VMs v_0, when the system is initialized, and by adding the number of VMs that are switched on, and removing the number of VMs that are switched off, i.e.

$$v_i = v_{i-1} + \#VM_{\mathrm{on}} - \#VM_{\mathrm{off}} \tag{12}$$

The top of Fig. 4 shows the evolution of v_i for the different methods over the *LOOPS* experiments shown in Fig. 3. The allocated number of VMs highlight that all the methods follow – in slightly different ways – the trend imposed by the desired throughput. Table 1 reports the average and the standard deviation of the number of VMs, for the different control strategies. When it comes to the number of VMs, the Free-Ratio and Step-Ratio control strategies have the lowest averages. Table 3 reports the p-values of the pairwise t-tests performed on the number of used VMs in *LOOPS* part of experiments 3. The table show that Free-Ratio and Step-Ratio are statistically better only with respect to Steps-Slow.

VM migration. Finally, the last metric considers the cumulative number of VM migration performed over the experiments. The cumulative number of migrations cm_i at cycle i, can be computed recursively on the basis of the number of current migrations m_i as

$$cm_i = cm_{i-1} + m_i \tag{13}$$

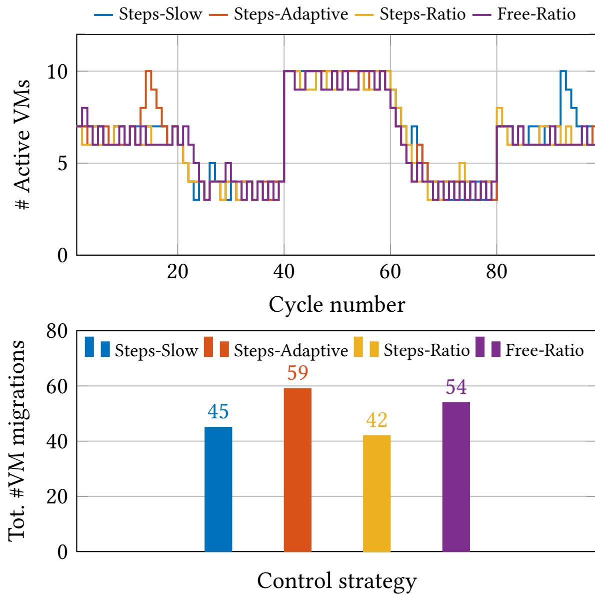

Figure 4: Number of active VMs (top) and total number of VM migrations (bottom).

The bottom of Fig. 4 shows the value of *cm* at the end of the *LOOPS* experiments shown in Fig. 3 tells that Steps-Slow control strategy beat the other strategies in term of resulting the minimum number of VMs migration.

Considering all the different metrics, we can conclude that the strategies which utilize the ratio and proportion method in calculating the "limit" value (*i.e.*, Steps-Ratio and Free-ratio) are better when they are combined with VML and PML. This is due to the following two reasons: (i) They use resources efficiently as they have the lowest averages of used resources (VMs), and (ii) They minimize the cost of VMs migration, as they produce the minimum number of VMs migration.

5 CONCLUSION

This work presents a holistic approach, called *LOOPS*, to ensure matching a desired metric for cloud-based applications. *LOOPS* works in three different levels, under the orchestration of an essential unit, in order to: (i) tune some variables allocated to co-hosted VMs during contention, (ii) scale VMs in/out, and (iii) migrate VMs. The results of evaluating the approach in a real testbed demonstrated a promising results in meeting the desired level of performance. *LOOPS* features fast response to changes which is a critical issue for real-time applications, and achieves the required management and scaling actions efficiently in a dynamic and periodic manner. As a future work, we are working on testing more control strategies to be implemented with different applications.

ACKNOWLEDGMENTS

This research has been performed with the support from the Swedish Knowledge Foundation (KKS) under the SACSys project, and from the Swedish Research Council (VR).

REFERENCES

[1] A. Al-Dulaimy, J. Taheri, A. Kassler, M. Hoseiny Farahabady, S. Dencg, and A. Zomaya. 2020. MULTISCALER: A Multi-Loop Auto-Scaling Approach for Cloud-Based Applications. *IEEE Trans. Cloud Computing* (2020).
[2] A. Ali-Eldin, M. Kihl, J. Tordsson, and E. Elmroth. 2012. Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control. In *W. Scientific Cloud Computing*.
[3] A. Beloglazov and R. Buyya. 2013. Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers under Quality of Service Constraints. *IEEE Trans. Par. and Distr. Syst.* 24, 7 (2013).
[4] E. Casalicchio and L. Silvestri. 2013. Mechanisms for SLA provisioning in cloud-based service providers. *Computer Networks* 57, 3 (2013).
[5] W. Dawoud, I. Takouna, and C. Meinel. 2012. Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning. In *Glob. Trends in Comp. and Comm. Syst.*
[6] S. Dutta, S. Gera, A. Verma, and B. Viswanathan. 2012. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *IEEE Int. Conf. Cloud Computing*.
[7] W. Fang, Z. Lu, J. Wu, and Z. Cao. 2012. RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center. In *IEEE Int. Conf. Services Computing*.
[8] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang. 2014. Adaptive, Model-driven Autoscaling for Cloud Applications. In *Int. Conf. Aut. Comp.*
[9] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond. 2014. Enabling Cost-Aware and Adaptive Elasticity of Multi-Tier Cloud Applications. *Future Gener. Comput. Syst.* 32 (2014).
[10] R. Han, L. Guo, M. M. Ghanem, and Y. Guo. 2012. Lightweight Resource Scaling for Cloud Applications. In *IEEE/ACM Int. Symp. Cluster, Cloud and Grid Comp.*
[11] R. Hu, J. Jiang, G. Liu, and L. Wang. 2013. KSwSVR: A New Load Forecasting Method for Efficient Resources Provisioning in Cloud. In *IEEE Int. Conf. Services Comp.*
[12] W. Iqbal, M. Dailey, and D. Carrera. 2009. SLA-Driven Adaptive Resource Management for Web Applications on a Heterogeneous Compute Cloud. In *Cloud Comp.*
[13] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek. 2011. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gen. Comp. Syst.* 27, 6 (2011).
[14] S. Islam, J. Keung, K. Lee, and A. Liu. 2012. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Gen. Comp. Syst.* 28, 1 (2012).
[15] J. Wei and C.-Z. Xu. 2006. eQoS: Provisioning of Client-Perceived End-to-End QoS Guarantees in Web Servers. *IEEE Trans. Comp.* 55, 12 (2006).
[16] S. Kundu, R. Rangaswami, A. Gulati, M. Zhao, and K. Dutta. 2012. Modeling Virtualized Applications Using Machine Learning Techniques. *SIGPLAN Not.* 47, 7 (2012).
[17] P. Lama and X. Zhou. 2010. Autonomic Provisioning with Self-Adaptive Neural Fuzzy Control for End-to-end Delay Guarantee. In *IEEE Int. Symp. Modeling, Analysis and Sim. of Comp. and Telecomm. Syst.*
[18] Z. Li, X. Yu, L. Yu, S. Guo, and V. Chang. 2020. Energy-efficient and quality-aware VM consolidation method. *Future Gen. Comp. Syst.* 102 (2020).
[19] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing* 12, 4 (2014), 559–592.
[20] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. 2009. Automated Control of Multiple Virtualized Resources. In *ACM European Conf. Computer Systems*.
[21] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. 2007. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *ACM European Conf. Computer Systems*.
[22] V. Podolskiy, A. Jindal, and M. Gerndt. 2019. Multilayered Autoscaling Performance Evaluation: Can Virtual Machines and Containers Co–Scale? *Int. Journal of Applied Mathematics and Comp. Science* 29, 2 (2019).
[23] R. Prodan and V. Nae. 2009. Prediction-based real-time resource provisioning for massively multiplayer online games. *Future Gen. Comp. Syst.* 25, 7 (2009).
[24] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin. 2009. VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration. In *Int. Conf. Aut. Comp.*
[25] J. Rao, Y. Wei, J. Gong, and C. Xu. 2013. QoS Guarantees and Service Differentiation for Dynamic Cloud Applications. *IEEE Trans. Network and Service Management* 10, 1 (2013).
[26] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.
[27] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood. 2008. Agile Dynamic Provisioning of Multi-Tier Internet Applications. *ACM Trans. Auton. Adapt. Syst.* 3, 1 (2008).
[28] L. Wang, J. Xu, M. Zhao, and J. Fortes. 2011. Adaptive Virtual Resource Management with Fuzzy Model Predictive Control. In *Int. Conf. Aut. Comp.*
[29] C.-Z. Xu, J. Rao, and X. Bu. 2012. URL: A unified reinforcement learning approach for autonomic cloud management. *J. Parallel and Distrib. Comput.* 72, 2 (2012).