Symbiotic and Sensitivity-Aware Architecture for Globally-Optimal Benefit in Self-Adaptive Cloud

Tao Chen
School of Computer Science
University of Birmingham
Birmingham, UK. B15 2TT
txc919@cs.bham.ac.uk

Rami Bahsoon
School of Computer Science
University of Birmingham
Birmingham, UK. B15 2TT
r.bahsoon@cs.bham.ac.uk

ABSTRACT

Due to the uncertain and dynamic demand for Quality of Service (QoS) in cloud-based systems, engineering self-adaptivity in cloud architectures require novel approaches to support on-demand elasticity. The architecture should dynamically select an elastic strategy, which optimizes the global benefit for QoS and cost objectives for all cloud-based services. The architecture shall also provide mechanisms for reaching the strategy with minimal overhead. However, the challenge in the cloud is that the nature of objectives (e.g., throughput and the required cost) and QoS interference could cause overlapping sensitivity amongst intra- and inter-services objectives, which leads to objective-dependency (i.e., conflicted or harmonic) during optimization. In this paper, we propose a symbiotic and sensitivity-aware architecture for optimizing global-benefit with reduced overhead in the cloud. The architecture dynamically partitions QoS and cost objectives into *sensitivity independent regions,* where the local optimums are achieved. In addition, the architecture realizes the concept of symbiotic feedback loop, which is a bio-directional self-adaptive action that not only allows to dynamically monitor and adapt the managed services by scaling to their demand, but also to adaptively consolidate the managing system by re-partitioning the regions based on symptoms. We implement the architecture as a prototype extending on decentralized MAPE loop by introducing an *Adaptor* component. We then experimentally analyze and evaluate our architecture using hypothetical scenarios. The results reveal that our symbiotic and sensitivity-aware architecture is able to produce even better global benefit and smaller overhead in contrast to other non sensitivity-aware architectures.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks—*Distributed Systems*

General Terms

Performance, Management.

Keywords

elasticity, cloud computing, optimization, symbiotic architecture.

SEAMS'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2864-7/14/06...$15.00
http://dx.doi.org/10.1145/2593929.2593931

1. INTRODUCTION

In cloud computing paradigm, the cloud-based services are deployed as Software as-a-Service (SaaS) and are typically supported by the software stack in the Platform as-a-Service (PaaS) layer [1]. They are also supported with Virtual Machines (VM) and hardware within the Infrastructure as-a-Service (IaaS) layer [2]. Under changing environmental conditions (e.g., workload, size of incoming job etc.), it is important to manage and control the Quality of Service (QoS) of cloud-based services. By QoS, we refer to the non-functional attributes (e.g., throughput) experienced by the end-users who use these services. In particular, the QoS can be managed by various control knobs, which include software (e.g., threads) and hardware resources (e.g., CPU) in a shared infrastructure. However, inappropriate use of software and hardware resources could result in large rental cost to the service. In this work, we refer to these control knobs and environmental conditions in the cloud as *primitives*.

With the context in mind, the term *elasticity* in cloud refers to the ability to adaptively scale control knobs to match the demand of cloud-based services. Given the uncertainty and dynamics of QoS, there is an increasing demand on cloud where the realization of elasticity can be managed without human intervention. Therefore, an architecture to address this problem is a contribution to the fundamentals of self-adaptive cloud. In particular, for all cloud-based services, this architecture should continuously select an elastic strategy, which is the combinatorial decision of configurations for various control knobs.

Most existing work for elasticity in self-adaptive cloud is either cost-optimized [8] or QoS-optimized [15], we argue that elasticity in the cloud should be global-benefit optimized, with an attempt to optimize both QoS and the required rental cost. The *optimal benefit* refer to the optimum performance of all QoS attributes with minimal costs for a cloud-based service. If each service in a cloud reaches its optimal benefit, then cloud is said to reach *globally-optimal benefit*. Achieve globally-optimal benefit in the cloud leads to a win-win situation: the owners of cloud-based services gain better QoS with less rental cost. On the other hand, the cloud provider could better utilize resources and earns better reputation.

The global benefit objective consists of various QoS and cost objectives. In the rest of the paper, we use objectives to refer to various QoS and cost objectives of a cloud-based service. Objectives in the cloud could be either conflicting or harmonic due to the presence of overlapping sensitivity (e.g., being sensitive to at least one identical primitive) amongst different QoS attributes and costs; this is referred to as *objective-dependency*. By sensitivity, we refer to the correlation between the fluctuation of QoS/cost to the stimuli caused by changing primitives. In particular, QoS sensitivity is generally dynamic (i.e., *which*, *when* and *how* primitives correlate with QoS tends to be dynamic) and we assume the cost is based on a fixed model and sensitivity. The objective-dependency could be either intra- or inter-service. Intra-service dependency refers to objectives, which are dependent in nature. This for example can be rental cost and throughput of a service. The inter-services dependency means the objectives of two services could be dependent on each other because of QoS interference caused by the co-located services on the Virtual Machine (VM) [3] (as resources contention on a VM) and the co-hosted VMs on a Physical Machine (PM) [4] (as resources contention on a PM). By QoS interference, we refer to scenarios where fluctuation of primitives can indirectly interfere with related services and their QoS due to resources contention. In addition, a dynamic service composition in the cloud implies that dependency might exist between QoS/cost objectives of the services on different PMs, as they are in the composition or functionally dependent on the same service.

The problem, which this papers addresses is how can the architecture dynamically and efficiently determine an elastic strategy that produces globally-optimal benefit. Nevertheless, a major challenge to the design of the architecture is that local optimization of objectives (e.g., optimize objectives per-VM) might not optimize the global benefit due to the presence of objective-dependency caused by overlapping sensitivity. On the other hand, a global optimization in the cloud is likely to result in large overhead in selecting an elastic strategy. As a result, there is a trade-offs between global benefit and overhead in the design.

A common lack in existing architectures for self-adaptive cloud is that they are not sensitivity-aware with respect to QoS and cost. Precisely, they partition the cloud into fixed regions; optimize for QoS and cost objectives and aggregate the results in each region. For example, existing architectures aim at either global optimum in one global region (e.g., *cloud-level*) or local optimum in different local regions (e.g., *PM-level*, *VM-level* and *service-level*) asynchronously and independently. Both solutions ignore QoS and cost sensitivity as their optimization assumes fixed region granularity. Given that the cloud tends to be dynamic and its QoS sensitivity changes at runtime, these architectures can result in inappropriate partitioning of regions, which can lead to non-optimal global benefit or large overhead when optimizing for the said regions. Both global benefit and the overhead are sensitive to the number of services and their objectives in the optimization process. Therefore, the trade-offs between global benefit and overhead is influenced by the region granularity. Consider now a complicated scenario, where the region granularity is linear to both global benefit and overhead in a given optimization algorithm: Figure 1 shows the likely trend of different fixed region granularities in relation to the global benefit and overhead. Based on the degree of granularity in optimization, we classify the architecture for self-adaptive cloud into 4 categories, as shown in Figure 1. *PM-level* denotes the architecture that partitions and aggregates the objectives of services per-PM regions; it attempts to reach local optimum in each region independently. The same principle can be applied to *service-level* and *VM-level*. On the other extreme, *cloud-level* simply consider the cloud as one region and employ an architecture that optimizes the highest level aggregation for objectives of all cloud-based services to reach global optimum. In Figure 1, we can see that finer region granularity implies less number of services and objectives within each region. This tends to result in worse global benefit but smaller overhead.

In this paper, we propose symbiotic and sensitivity-aware architecture, which leverages on a decentralized MAPE loop. This architecture can efficiently produce globally-optimal benefit with reduced overhead. The novelty is that QoS and cost objectives of cloud-based services are dynamically partitioned into *sensitivity independent regions* where any objectives of a region are independent to those of the other regions, as a result each region can be optimized locally. Particularly, we dynamically determine the level of region granularity on the fly. In addition, we apply symbiotic feedback loops to the architecture. The differences between such symbiotic feedback and the conventional feedback is that the adaptation in the former one is *bio-directional*: the architecture is not only able to monitor and select better elastic strategy to the managed services, but also able to adaptively consolidate itself by re-partitioning the regions. Henceforth, such *bio-directional* adaptation is referred to as symbiotic feedback.

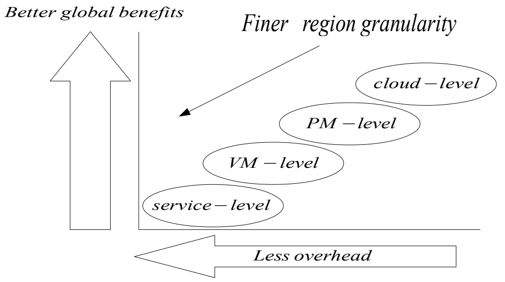

Figure 1. Approximated relationship of fixed region granularities to global benefit and overhead in cloud.

The architecture leverages on our previous work[3], which reports on a sensitivity-aware QoS modeling approach that adaptively learn the correlation between QoS and the useful primitives in cloud. In this paper, we apply our modeling approach to learn dynamic QoS sensitivity so we can partition the regions. Specifically, we make the following novel contributions:

Firstly, unlike existing work, which model the QoS/cost per-application or per-VM. we look at the QoS and cost for each individual services.

Secondly, we consider intra- and inter-services objective-dependency. In addition, we do not rely on fixed instance-type to form an elastic strategy, but we assume arbitrary combinations of control knobs. This would provide more flexible elasticity and follows the current trend in cloud [5].

Thirdly, to enable sensitivity-awareness in the proposed architecture, we develop a 2-phases region partition strategy that partitions the QoS/cost objectives into sensitivity independent super-regions. These super-regions can further partition the objectives into sensitivity independent regions, where the local objectives are optimized independently. The partitioning of super-regions and regions rely on deployment (e.g., VM to PM mapping) and their QoS/cost sensitivity respectively. The basic principle behind the notions of sensitivity independent regions is that, we can reach globally-optimal benefit by asynchronously finding locally-optimal benefit within each sensitivity independent region. This can eventually reduce the search space.

Fourthly, we propose the concept of symbiotic feedback and adaptation in the architecture, which realizes bio-directional adaptation between the managed services and the architecture.

Fifthly, we implement our architecture prototype based on MAPE loop with an extended *Adaptor* component, which realized a hierarchical stack to manage the deployment and QoS sensitivity changes separately. We experimentally evaluate the architecture via hypothetical scenarios, which contain different numbers of services. The results reveal that our symbiotic and sensitivity-aware architecture is able to produce similar global benefit to the PM-level architecture, and better than cloud-level, VM-level and service-level architectures. On the other hand, it produces smaller overhead than the cloud-level and the PM-level architecture; and could be similar to that of the service-level and the VM-level ones. In particular, the achieved global benefit and overhead in our architecture tends to be better when it is possible to have more sensitivity independent regions.

In the following, Section 2 decomposes the problem of globally optimizing benefit in elastic cloud and presents the system model. Section 3 specifies the logical notions of super-region and region. Section 4 describes the physical deployment of our architecture and the components. Section 5 reports on the evaluation and analysis of experimental results. Section 6 and 7 discuss related work and present the conclusion respectively.

2. MODEL AND PROBLEM ANALYSIS

In this section, we present our assumptions and the models used for analyzing the problem.

2.1 Cloud System Model

We assume that cloud-based applications are composed of one or more services, each with its QoS requirements and can experience different environmental changes (e.g., changes in workload). These services are deployed on a cloud software stack, which can be setup using various configurations and tactics. In addition, they are hosted on the cloud infrastructure, where resources are shared via VMs. As a result, the control knobs and environmental conditions could significantly influences their QoSs. In distributed environment like cloud, each tier in a multi-tiers application, composed of concrete services $\{S_1, S_2, \ldots S_i\}$ may have multiple replicas deployed on different VMs. The replica of a tier running on a VM is assumed to have the replicas of its services running on the same VM. In this work, we refer to the replicas of concrete services as service-instances: the *jth* service-instance of the *ith* concrete service is denoted by S_{ij}. Unlike existing work [e.g., 4, 14, 15], which focus on realizing elasticity at the application and VM level, we aim to adaptively optimize the QoS attributes and rental cost of utilizing control knobs for each individual service-instance, considering the QoS interferences caused by the co-located service-instances on a VM and the co-hosted VMs on a PM.

In addition, we do not consider global resources contention caused by shortage in cloud capacity; our architecture works for cases where software and hardware resources tend to be available, which is normal in a cloud environment. Henceforth, we assume that the maximum demand of software and hardware resources for all cloud service-instances (e.g., according to their budget) should be satisfied by the capability of the cloud provider. Under such assumption, we eliminate extreme cases where the capacity of cloud provider reaches its limits causing likely global resources contention. This is because the increasing demand of each service-instance would eventually be satisfied by scale up/out as long as the cost does not exceed the budget. We believe this is a reasonable assumption as in realistic scenarios, proper admission control can be applied to restrict the number of cloud-based service-instances. Moreover, in case where the cloud provider actually encounters capacity shortage, the unsatisfied services can be switched to an alternative provider via a cloud selection mechanism, which presumably hold our assumption. However, the design of admission control and selection mechanism is outside the scope of this paper.

2.2 Cloud Primitives

We advocate a fine-grained approach to the modeling of QoS. To achieve this, we decompose the notion of primitives into two major categories: these are ***Environmental Primitives (EP)*** and ***Control Primitives (CP)***. We posit that CP can be either software or hardware, which could be managed by cloud providers to support QoS provisioning. In particular, software CPs are software tactics and configurations; such as the number of threads in thread pool and its life time, the number of connections in database connection pool, security and load balancing policies etc. Whereas, hardware CPs are computational resources provisioning, such as CPU, memory and bandwidth. As shown in Figure 2, software and hardware control primitives rely on the PaaS and IaaS layers respectively. In particular, it is a non-trivial task to consider software CPs when QoS modeling in the cloud as they tend to influence QoSs significantly [6]. On the other hand, we look at EPs in the context of highly dynamic scenarios, which reflect the cloud setting. The EPs can significantly influence the QoS. The cloud providers often can not predict and fully control their behavior. Examples include unbounded workload and unpredictable bound received data etc. If the cloud provider would be able to predict and control the presence of these scenarios, these can be then considered as CPs.

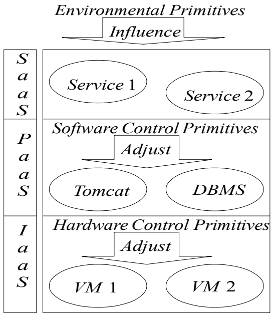

Figure 2. The cloud primitives.

2.3 Problem Models and Objective

We formulate an "online" QoS model, which captures both dynamic sensitivity and interference with respect to the selected primitives over time. The model at given sampling interval *t* is formally expressed as:

$$QoS_k^{ij}(t) = f(SP_k^{ij}(t), \delta) \tag{1}$$

where $QoS_k^{ij}(t)$ is the average value of *kth* QoS of S_{ij} at interval *t*. *f* is the QoS function, which dynamically changes at runtime. δ refers to any other inputs that are required by the algorithm to train *f* apart from the primitives. Examples of other inputs may include historical QoS values and tuning variables. To handle QoS interferences, we denote the input $SP_k^{ij}(t)$ of Eq. 1 as the *selected primitives matrix* of $QoS_k^{ij}(t)$ at interval *t*. This matrix contains the selected primitive inputs of $QoS_k^{ij}(t)$ and it is updated online. In this work, we dynamically update $SP_k^{ij}(t)$ and function *f* using the QoS modeling approach described in our previous work [3].

In the context of cloud, utilizing CPs may be subject to certain monetary cost to the service owners, therefore the total costs model for S_{ij} can be represented as:

$$Cost^{ij} = \sum_{a=1}^{n} g(CP_a^{ij}(t), P_a) \tag{2}$$

where *g* is the fixed and unified cost function for each type of CP, and *n* is the total number of CP type that used by service-instance S_{ij} to supports its QoS attributes. $CP_a^{ij}(t)$ is the amount of the *ath* CP provision for S_{ij} at interval *t*. P_a denotes the corresponding price per unit of the *ath* CP. In this work, we assume that the price of each CP type is fixed for all service providers and their service-instances. It is worth noting that the hardware CPs (e.g., CPU and memory) can be only provisioned for each VM whereas the cost model is per-service based, thus the price of a hardware CP should be equally proportioned to each of the service-instances deployed on the provisioned VM.

To achieve globally-optimal benefit in elastic cloud, our architecture aims at adaptively and dynamically determine and scale to the CP configurations, which supports the best of all QoS attributes (Eq. 1) with minimal costs (Eq. 2) for all service-instances in the cloud. In this work, we apply a linear weighted-sum aggregation to express the global benefit for QoS attributes and costs of different service-instances in the cloud. Formally, at any given interval *t*, we aim to optimize the global objective by maximizing the function in Eq. 3.

$$\sum_{i=1}^{n}\sum_{j=1}^{m} w'_{ij}\cdot(\sum_{a}^{l} w_a\cdot QoS_a^{ij}(t)-\sum_{b}^{r} w_b\cdot QoS_b^{ij}(t)-w_{(l+r+1)}\cdot Cost^{ij} \quad (3)$$

where *n* and *m* are the total number of services and their instances in the cloud; w'_{ij} is the weight for each service-instance. Because the global objective is to maximize Eq. 3, we need to carefully place the maximized QoS (e.g., throughput) and the minimized ones (e.g., response time); thus *l* and *r* are the total number of the maximized and minimized QoS for S_{ij} respectively; w_a, w_b and $w_{(l+r+1)}$ are refer to the corresponding weight of the QoS and cost for S_{ij}. In addition, the optimization of Eq.3 should be subject to the constraint of budget and SLA.

It is worth noting that the purpose of this work is not to find out the best formalization of the global benefit and its optimization algorithms; but to evaluate the effectiveness of our symbiotic and sensitivity-aware architecture towards reaching globally-optimal benefit. In future work, we will look at more sophisticated formalization (e.g., removal of the weights) of the global benefit.

3. LOGICAL VIEW OF THE ARCHITECTURE

In the section, we explain the notions and principles behind our architecture from a logical perspective. The practical architecture deployment in cloud environment is demonstrated thereafter.

3.1 An Overview

Recall that our objective is to optimize the global benefit for QoS attributes and costs of all service-instances, therefore from a logical point of view, our basic problem entity in the cloud are different QoS (max/min Eq. 1) and cost (min Eq. 2) objectives of different service-instances. The objective functions of these objectives are their corresponding QoS/cost models, the *kth* objective of S_{ij} is denoted by O_k^{ij}. In particular, we argue that any two objectives are either dependent (i.e., conflicted or harmonic) or independent (i.e., an objective is neither directly/transitively conflicted nor harmonic with another). With this in mind, we propose a 2-phased region partitioning, where the first phase partitions the objectives into different sensitivity independent *super-region*, which defines the boundary of likely independent objectives for the entire cloud under current deployment. The purpose of super-region is to classify those objectives, which might be independent for now but could become dependent to the others as the QoS sensitivity changes. In other words, the objectives should be partitioned into the same super-region as long as they are *likely* to have objectives-dependency. In the second phase, the objectives within each super-region are further partitioned into smaller sensitivity independent *regions* where the local optimization takes place. By *sensitivity independent regions*, we refer to the case where any objective from a region is *currently* independent to any objective from another region at given time. By doing so, the search space of the global objective function in Eq.3 is partitioned into *n* subspaces based on sensitivity, where *n* is equivalent to the number of regions. The objectives within each subspace (still can be expressed by Eq. 3, but with smaller search space) is optimized independently and asynchronously.

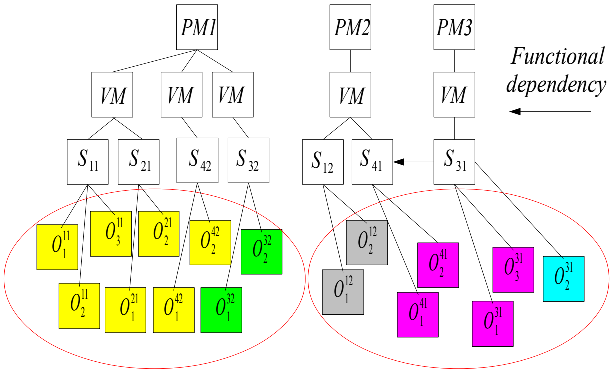

Figure 3. Overview of the notion of super-region and region.

The basic principle behind our symbiotic and sensitivity-aware cloud architecture is that, we can reach a globally-optimal benefit by asynchronously doing local optimization for locally-optimal benefit within different sensitivity independent regions, which have smaller search space. The partitioning of super-region and their regions is a dynamic online process based on the deployment and sensitivity respectively, which are expressed by rules (we will describe in Section 3.2 and 3.3).

In the following, we use SR_i to denote the *ith* super-region and R_k^i to denote the *kth* region of the *ith* super-region. The partitioning should follow the constraints below:

Constraint 1: $\forall(R_a^i \cap R_b^j)=\emptyset$

Constraint 2: $if\ (\exists O_a^{ij} \in SR_k)\ and\ (\exists O_b^{ij} \in SR_l),\ then\ SR_k = SR_l$

Constraint 1 means that each objective can at most belongs to one region within a super-region. *Constraint 2* indicates that all objectives of a service-instance should belong to an identical super-region. However, these objectives might belong to different regions within such super-region. The logical view of our 2-phased region principle in the cloud is shown in Figure 3 where we assume a simple scenario consists of 3 PMs, 4 VMs and 6 service-instances with various QoS/cost objectives. The two red cycles represent two super-regions. Different colors on the objective entities express different regions within those two super-regions. In addition, there is a functional dependency between S_{41} and S_{31}, which means that S_{41} requires the invocation of S_{31} to complete its service.

3.2 Super-Regions

Objectives in the entire cloud can be partitioned to different super-regions. Each of the super-region contains the objectives of service-instances that are likely to be directly or transitively dependent. The partitioning rule of super-regions is specified as:

Rule 1: $Given\ S_{ab}, S_{cd}\ \ and\ \ \forall O_i^{cd} \in SR_k,\ then (\forall O_j^{ab}\ of\ S_{ab})$ *belongs to* $SR_k\ if$:

1) S_{ab} and S_{cd} are deployed on the same VM/PM, or
2) S_{ab} has direct functional dependency on S_{cd}, or
3) S_{cd} has direct functional dependency on S_{ab}.

Rule 1 assumes that given arbitrary service-instances S_{ab} and S_{cd}. It also assume that the objectives of S_{cd} are in the super-region SR_k. Under these assumptions, objectives of S_{ab} are said to belong to SR_k if and only if it follows any of the above three conditions (either directly or transitively).

Consider the scenario in Figure 3 as an example. The objectives on PM1 are assigned to the same super-region because they satisfy *condition 1* in *Rule 1*. On the other hand, the objectives on PM2 and PM3 form another super-region as they satisfy all the conditions. In particular, the objectives of S_{12} and S_{31} are within the same super-region even they do not directly satisfy any of the conditions. This is because S_{41} functionally depends on S_{31} , thus they satisfy *condition 2*. In addition, S_{41} and S_{12} satisfy *condition 1*. As a result, the S_{12} and S_{31} transitively satisfy the conditions in *Rule 1* via S_{41}.

Given the assumption that shortage in cloud capacity is beyond our concerns, the partition rule of super-regions are designed based on the fact that the objectives of a service-instance and its functionally dependent service-instances are very likely be dependent under some scenarios (e.g., sequential interaction). In addition, the likely QoS interference can only be caused by the co-located service-instances on a VM and the co-hosted VMs on a PM. Therefore, the objectives from any service-instances that do not directly or transitively satisfy *Rule 1* can be optimized independently as they would have no way to influence each others.

The partitioning of super-region could change at runtime due to the dynamic cloud environment. The super-regions would be re-partitioned according to *Rule 1* upon *deployment changes*, for example: VM migration/replication, PM boots-up/shutdown and changes in service compositions etc.

3.3 Regions

Within each super-region, we further partition the objectives into different sensitivity independent regions, where a local optimization algorithm is running. The partitioning of regions could be triggered upon symptoms described in Section 4. The aim is to further narrow down the number of dependent objectives according to their *current* sensitivity at a given time. Therefore, the partition *Rule 2* of regions are designed based on the sensitivity of QoS and cost models presented in Section 2:

Rule 2: *Within a super-region SR_l, given O_i^{cd} and $\exists\, O_j^{ab} \in R_k^l$,*
then $O_i^{cd} \in R_k^l$ if O_i^{cd} has inputs in common to O_j^{ab}.

Concretely, *Rule 2* expresses that an objective should belongs to a region R_k^l if and only if it has at least one identical primitive input to one or more objectives from R_k^l (meaning that they are dependent and have overlapping sensitivity). If two objectives have neither common inputs themselves nor common inputs to the same intermediate objectives, they are said to be independent during optimization.

Using the scenario in Figure 3 as an example. There are two regions within the left super-region; this is because the objectives of S_{11}, S_{21} and S_{42} use certain identical primitives inputs. On the other hand, the objectives of S_{32} is in an alternative region because it is insensitive to and has no identical inputs to any of those objectives from S_{11} and S_{21} as it suffers limited QoS interference on the co-located services. In particular, suppose that O_2^{11} has identical inputs to O_1^{21} and O_1^{11} ; O_1^{11} and O_1^{21} do not directly satisfy *Rule 2*. However, all of these 3 objectives are put in the same region because O_1^{11} and O_1^{21} are transitively satisfy *Rule 2* via O_2^{11} . Similar scenario occurs in the right super-region. In addition, we can see that even O_1^{31} , O_2^{31} and O_3^{31} are objectives of the same service-instance, O_2^{31} is put in an alternative region to that of O_1^{31} and O_3^{31} . This is a possible scenario: suppose that O_3^{31} is cost objective, O_1^{31} and O_2^{31} are throughput and consistency QoS objective respectively; it is likely that O_2^{31} is only sensitive to an unique CP (e.g., ordering error), which is free of charge and henceforth, it is independent on O_1^{31} and O_3^{31} .

Similar to the super-regions, the partitions of regions are also subject to dynamic changes. However, region portioning is likely to change more frequently than that of the super-region. This is because it requires updates when changes in QoS sensitivity tend to be significant. Examples of *significant QoS sensitivity changes* could include scenarios, where QoS is becoming sensitive to a new primitive or insensitive to an existing primitive. Insignificant changes on *how* the primitives correlate with QoS can not trigger re-partitioning of the regions.

4. ARCHITECTURE AND COMPONENTS

In this section, we specify the deployment of architecture and we detail its components.

4.1 The Bio-directional Adaptation in Symbiotic Loop

The physical deployment of architecture is shown in Figure 4. As we can see that the architecture is deployed as distributed instances, each of which running on a separate VM (e.g., *Dom0* on Xen [7]) on every PM in the cloud. In particular, for each instance, we adopt a decentralized MAPE style. Unlike the traditional realization of MAPE, our architecture adapts symbiotic loop to enable bio-directional adaptation in the sense that: 1) it does not only allow the MAPE to monitor and manage the service-instances by dynamically search an elastic strategy for globally-optimal benefit upon symptoms. 2) It also adaptively consolidates itself with the up-to-dated context information by dynamically re-partitioning the super-regions and regions. Such consolidation is realized in the extended *Adaptor* component. In addition, the architecture can trigger adaptations in both proactive and reactive manners. In our prototype, the communication amongst instances from different PMs is realized by Group Communication Service (GMS) [16], which supports fast and reliable multicast protocols.

The workflow of bio-directional adaptation has been shown in Figure 4. More precisely, the *sensor* on each PM collects the data (e.g., QoS values, CP usages and EP values) from the underlying VMs and service-instances; and possibly from other PMs due to functional dependency (step 1). In addition, the sensor could sense deployment changes and QoS sensitivity changes from other PMs. Next in step 2, the sensor passes raw information it received to the *Monitor* (denoted by M) for normalizing the data. At step 3, the *Analyzer* (denoted by A) receives both current and historical data after normalization, this data is used by QoS modeler to build QoS models (step 3.1). The QoS models, cost models and the related detected changes are transiting to the *Adaptor* via step 4. The partitioning of super-region/region and/or adaptation can be triggered if one or more of the following symptoms is detected:

- *Symptoms 1:* Proactively detect if the QoS of a service-instance is likely to violate SLA constraint by using the QoS models.
- *Symptoms 2:* Reactively detect if the QoS of a service-instance has violated its SLA constraint and/or if the utilization of a CP has violated the constraint.

- *Symptoms 3:* Significant changes in the QoS sensitivity of the objectives in a managed region.
- *Symptoms 4:* Deployment changes occur in a managed super-region.

Symptoms 1 and 2 would trigger the elastic adaptation of the managed service-instance(s); whereas, symptoms 3 and 4 require the architecture to adapt itself by re-partitioning the super-regions and/or regions. In particular, to prevent the problem of triggering elastic adaptation too frequently, symptoms 1 and 2 are valid only if the leap time after the previous adaptation for the affected service-instances is more than a threshold *t*. Once we reach the *Adaptor* component, the changes in symptoms 3 and 4 would be addressed separately in a hierarchical stack. Concretely, *Super-Region Control* component manages symptom 4 and maintains the super-region on to its PM (step 4.1) as only one super-region exists on a PM according to *Rule 1*. In the lower stack, *Region Control* component manages the regions within the aforementioned super-region (step 4.2 and 4.3) according to *Rule 2*; it aims to cope with symptom 3. Additionally, it could be triggered by symptom 4 as the partition of a super-region might change. Once both symptoms 3 and/or 4 are resolved, the propagation goes to the *Planner* component (denoted by P) where the *Autoscaler* component within each region is designed to address symptoms 1 and 2. This can be done through dynamically searching the best adaptation strategies toward the locally-optimal benefit of region, using the QoS and cost models (step 4.4). In particular, the autoscaler of each regions is triggered independently and asynchronously. There are cases where a region might be associated with multiple PMs (we will explain this in Section 4.5). Therefore in order to ensure that each region is optimized on one PM; the autoscaler can be activated only if the leader of those PMs confirm that the region is not currently being optimized on any other PMs. These processes are expressed as step 4.5 and 4.6.

Once the elastic strategy is determined, the process proceeds to the *Executor(s)* (denoted by *E*) via step 5. In particular, *E* is responsible for determining which concrete actions (e.g., scale up/down, in/out and/or VM migration and replication etc) need to be taken in order to fulfill the elastic strategy. In this work, we consider both vertical and horizontal scaling and apply a simple solution to determine the actions, this is: we always try vertical scaling (i.e., scale up/down) first before horizontal scaling (i.e., scale out/in). This is because horizontal scaling is usually more expensive than vertical scaling. As for the VM migration/ replication decision, we always choose the one that result in smaller overhead based on a predefined VM profiling pattern. Finally, the actions are taken by the actuator via step 6 and 7.

4.2 QoS Modeler

The QoS models are dynamically constructed within *QoS modeler* component using our previous work [3]. In particular, the modeling is an online and continuous process, which captures the dynamic QoS sensitivity of the underlying service-instance. Each PM has a dedicated *QoS modeler* to update the QoS models for all service-instances deployed on the VMs, which hosted on the said PM. In this work, we simply make use of the resulted models to search the best elastic strategy and to determine the partitions of regions within each super-region in the *Planer* component.

4.3 Super-Region Control

Super-Region Control is designed to manage the partitioning of super-regions in the cloud and it aims to cope with symptom 4. In particular, the dedicated *super-region control* on each PM only maintains one super-region. As a result, each PM only shares partial view of the entire super-region partitions in cloud. To determine re-partitioning following deployment changes, each PM share a global knowledge of the deployment dependency (i.e., service-instance to VM mapping, VM to PM mapping and the functional dependency of service-instances). The initial partitions can be easily setup by the cloud administrator using the rules mentioned in Section 3. When deployment changes occur (symptom 4), each *super-region control* updates the global knowledge of deployment dependency; and its super-region according to *Rule 1*. As mentioned, the communication amongst PMs is realized by GMS, thus the PM where the deployment changes are taken place would notify the other PMs..

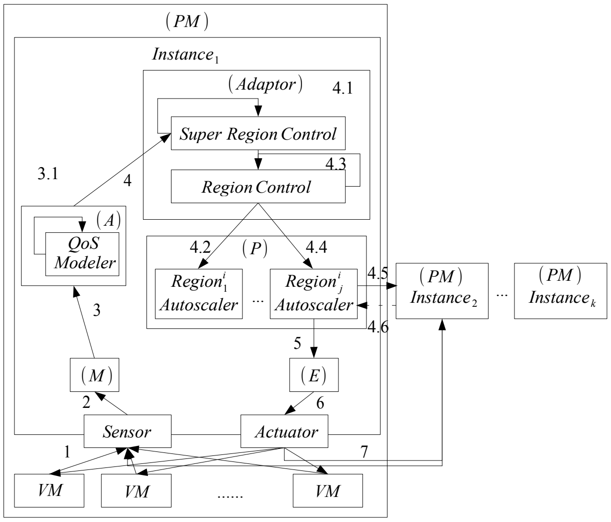

Figure 4. Overview of the architecture deployment.

It is possible that two or more *super-region controls* maintain the same super-region. Recall the scenario in Figure 3, the right super-region would be maintained by both PM2 and PM3 as both PM owns objectives, which belong to the same super-region. However, the *super-region controls* on both PMs only need to asynchronously maintain an identical view of super-region, and make sure their view is consistent when deployment changes occur. Henceforth there is no extra overhead caused by constantly reaching consensus.

4.4 Region Control

When the necessary re-partition of super-regions is completed, *Region Control* is responsible for managing the partitions of multiple regions within a super-region; it aims to cope with symptom 3. However, it should be triggered if symptom 4 is detected since the partition of super-region might change. As there is only one super-region on each PM, the regions within super-regions are managed by a dedicated *region control.* In particular, the *region control* keeps track on the models of the managed objectives. Specifically, at the end of each model update intervals, the *region control* would examine whether any deployment and/or significant QoS sensitivity changes occur in the managed super-region and regions. If this is the case, it then re-partitions the objectives to regions within the corresponding super-region, according to *Rule 2*.

It is possible that two or more *region controls* maintain the same regions. As shown in Figure 3, the region of O_1^{41} , O_2^{41} , O_1^{31} and O_3^{31} cover both PM2 and PM3, thus this region is maintained on both PMs. In this case, the *region control* only required to keep an identical view of the said region and ensure the related QoS models are updated.

Table 1. Initial deployments and the examined objectives/primitives

PM	VM	Service-instance	Objectives	Software CP	Hardware CP	EP
PM1	VM	S_{11}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{21}	Throughput and cost	The max threads		workload
	VM	S_{31}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{41}	Throughput and cost	The max threads		workload
PM2	VM	S_{12}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{51}	Throughput and cost	The max threads		workload
PM3	VM	S_{32}	Throughput and cost	The max threads	CPU and Memory	workload
		S_{61}	Throughput and cost	The max threads		workload

Table 2. Scaling options and price of control primitives

CP	Optional Values	Unit	Price
Max Threads	5,10,15,20,25,30,35,40,45,50	Thread count	\$0.8 for each 5 unit per hr
CPU	1, 2,3, 4,5,6, 7, 8	Compute Unit	\$2.5 for each 1 unit per hr
Memory	0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2	GB	\$1.5 for each 0.1 unit per hr

Table 3. Number of regions for each architecture under different setups of service-instances

Setup	Number of regions				
	symbiotic and sensitivity-aware	cloud-level	PM-level	VM-level	service-level
2 service-instances	maximum of 1	1	1	1	2
4 service-instances	maximum of 1	1	1	2	4
6 service-instances	maximum of 3	1	2	3	6
8 service-instances	maximum of 4	1	3	4	8

4.5 Autoscaler of Region

Once the partitions of both super-region and region are completed, the autoscaler of each region would be in a functional stage to trigger local optimization for searching an elastic strategy. In particular, the autoscaler aims to cope with symptoms 1 and 2. It is possible that, the *region control* on different PMs manage the same region as mentioned in Section 4.4. To prevent the same region from being optimized on more than one PM, we elect a leader for each group of PMs that manage the same region. More precisely, upon the occurrence of symptoms 1 or 2, the autoscaler of the corresponding region firstly queries the leader regarding whether the region, which the affected objective belongs to, is currently under optimization. If it is not the case, it will then trigger local optimizations; otherwise, the process should be aborted. Leader would be notified once the local optimization and the corresponding actuations are completed. In our preliminary prototype, we elect the oldest PM as the leader of each group.

5. EXPERIMENTAL EVALUATION

To evaluate global benefit of the elastic strategies produced by our architecture and the overhead for reaching these strategies, we have conducted an experimental evaluation. In particular, we have implemented the architecture prototype using Java JDK1.6, and we assessed the elastic scaling of 8 hypothetical cloud-based service-instances under the control of our architecture prototype. In the experiment setup, each service-instance was deployed on software stack including Apache, Tomcat and MySQL. We simulate a synthetical workload to each service-instance. The workload has been designed in a way that the intensity was sufficient for causing QoS interference on the co-located services and co-hosted VMs. The testbed is a private cloud, where PMs are connected by Gigabit Ethernet and a switch. Xen [7] is used as the underlying hypervisor. The initial deployment and the considered CP/EP of our experiments are shown on Table 1. The scale of each CP and their corresponding prices are specified in Table 2.

We compare our symbiotic and sensitivity-aware architecture (we simply refer to as sensitivity-aware architecture in the following sections) to other 4 architectural styles that do not cater for sensitivity. Each of the 4 architectures assumes different fixed region granularities: service-level, VM-level, PM-level and cloud-level architectures. Because these 4 styles do not consider symptoms 3 and 4; they trigger elastic adaptation only when symptoms 1 and/or 2 are detected.

For simplicity, we assume that each service-instance has only one QoS requirement, which is throughput and one predefined cost model. To optimize the global objective function in Eq.3 , we apply random optimization algorithm with the same number of iterations for each architecture. This is because exhaustive algorithms might not be able to produce a decision efficiently due to the large number of possible elastic strategies. In addition, we assume that these service-instances and their QoS/cost are equivalently important and thus all weights in the global objective function are set to 1.

5.1 The Global Benefit

To examine the global benefit of the elastic strategies produced by our symbiotic and sensitivity-aware architecture, we run 4, 6 and 8 service-instances setups separately for 100 sampling intervals. For each of the setup, we collect the quality of global benefit for each elasticity strategy made during the period. The purpose of the different setups is to examine the sensitivity of our architecture to the total number of objectives in cloud. Under each setup, we have performed independent runs for each of the five architectures. The global benefit is measured by score, which is the average result calculated by Eq.3 for the interval after a previous elasticity decision point and before the next one. Each of these intervals is referred to as *effect point*. Table 3 illustrates the number of regions for each architecture, which was observed during the experiments. It is worth noting that unlike the other architecture, the number of regions in our symbiotic and sensitivity-aware architecture is subject to dynamic change. Therefore, the number for our architecture shown in Table 3 is the maximum observed partitions of regions.

Figure 5-7 illustrate the results of the global benefit score (y-axis) in relation to each effect point (x-axis). Precisely, Figure 5 shows the global benefit of our architecture in contrast to the other 4 styles using setup for service-instances S_{11}, S_{21} , S_{31} and S_{41}. As we can see that the differences in global benefit for the sensitivity-aware, the PM-level and the cloud-level architecture are marginal. This is because they partition all the objectives of these 4 service-instances within the same region. Therefore, they perform the same under such case. In contrast, the service-level and the VM-level architecture achieve much worse global benefit following

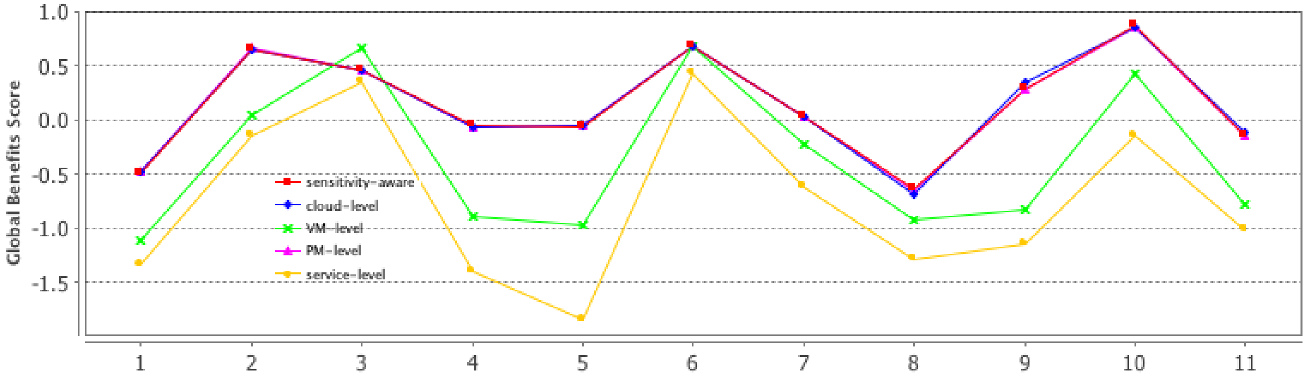

Figure 5. Global-benefit in case of 4 service-instances.

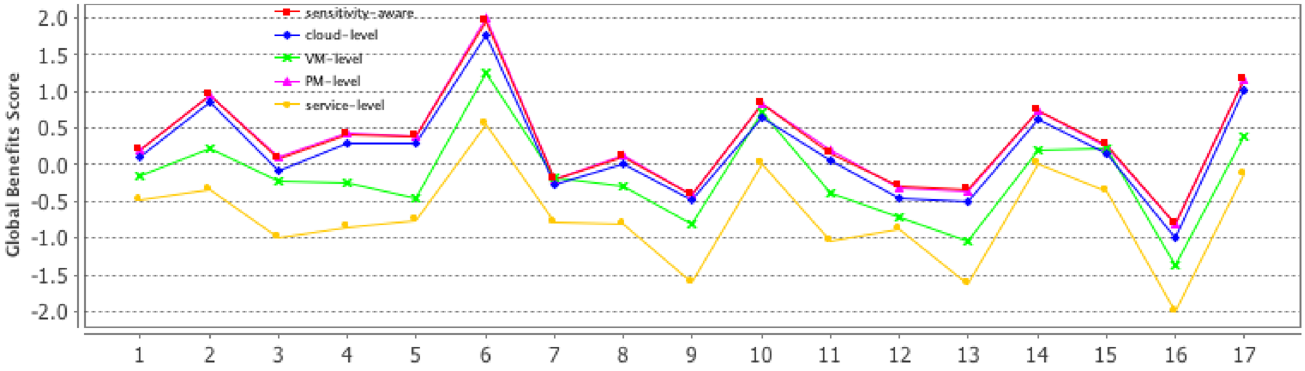

Figure 6. Global-benefit in case of 6 service-instances.

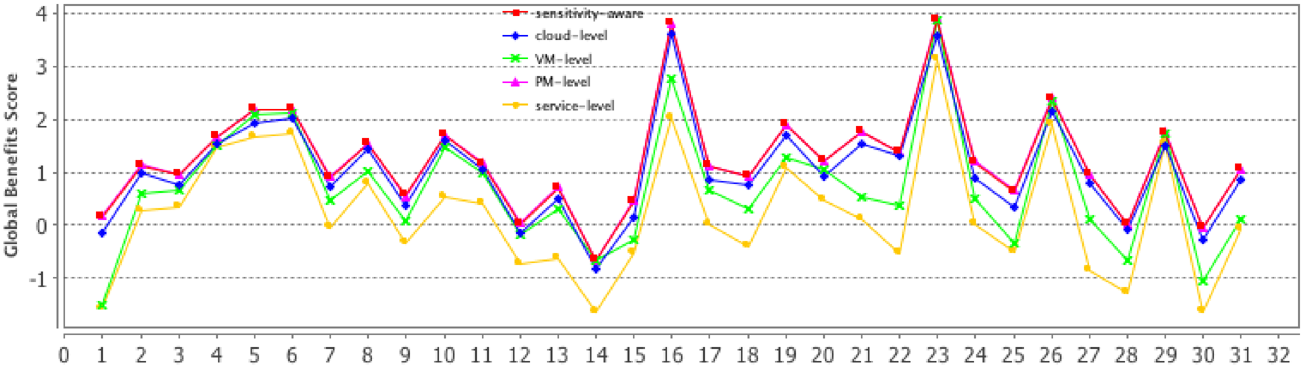

Figure 7. Global-benefit in case of 8 service-instances.

the elastic adaptation. This is due to incorrect partitioning of the regions as they ignore the sensitivity caused by QoS interferences on co-located services and co-hosted VMs, which are significant in our experiments. Figure 6 considers two more service-instances (S_{12} and S_{51}) in addition to the ones of Figure 5. We can see that the service-level and the VM-level architecture performs worse than the other three due to the same reason as the previous case. Surprisingly, although our architecture (at most 3 regions) partitions more regions than that of the cloud-level one, its global benefit is better than that of the cloud-level one. We believe that this is because we apply random algorithm in the optimization and our architecture is able to properly partition the objectives into more regions. This implies that optimizing locally and asynchronously on each sensitivity independent region could result in emergent global benefit using probabilistic algorithms. The PM-level (2 regions) architecture, on the other hand also performs better than that of the cloud-level one. We believe that this is because it partitions the objectives per-PM, which similar to the partitions produced by our architecture and thus meets the actual sensitivity in the experiments by chance. The sensitivity-aware architecture performs similarly in contrast to the PM-level architecture. This is because they produce similar partitions of regions. The only difference is that our sensitivity-aware architecture produces one extra region (we observe only 2 objectives within such region), which is not significant enough to produce emergently better results. However, in the next section we will show that our architecture produces much smaller overhead than that of the PM-level one.

Finally, Figure 7 illustrates the global benefit for all 8 service-instances. We can see that the service-level and the VM-level architecture produce the worst results. The gap between their results to the other three is larger than Figure 5 and 6. This is due to the fact that they has incorrectly partitioned the regions when introducing more service-instances, and henceforth affecting the global benefit more seriously. Similar to the case of Figure 6, our sensitivity-aware architecture performs slightly better than that of the cloud-level one. The PM-level architecture performs similar to our architecture for the reasons previously explained.

In summary, the elastic adaptations of our symbiotic and sensitivity-aware architecture produces much better global benefit than the service-level and the VM-level architecture under the presence of QoS interferences. In addition, the global benefit produced by our architecture are slightly better than that of the cloud-level architecture and similar to the PM-level architecture. We observe that the improvement in global benefit tend to be better when having more sensitivity independent regions. In addition, we believe that our architecture could outperform the PM-level one when the number of QoS attributes and/or the number of services on each PM increase.

5.2 The Overhead for Reaching Elastic Strategy

To evaluate the overhead for reaching an elastic strategy, we compare the average time taken in the optimization processes of the symbiotic and sensitivity-aware architecture to the other 4 styles, under the setup of 2, 4, 6 and 8 service-instances. In particular, the average time is calculated based on the time taken for reaching all the elasticity strategies within the entire experiment run. As shown in Figure 8, which reveals the overhead (y-axis) in relation to the number of service-instances (x-axis), we can see that in case of 2 service-instances (S_{11} and S_{21}), the service-level architecture produces the smallest overhead. This is because it performs optimization and reaches a strategy for each service-instance independently. The remaining architectures, on the other hand, produce similar overhead because all the service-instances exist on a single VM.

In the case of 4 service-instances (S_{11}, S_{21} , S_{31} and S_{41}), the differences among the sensitivity-aware architecture, the PM-level and the cloud-level architectures are marginal. They tend to result in bigger overhead than that of the service-level and VM-level ones. This is because the sensitivity-aware architecture and the PM-level one only results in one region; they are actually the same as the cloud-level architecture. In contrast, the service-level and the VM-level style are unaffected by the increasing number of service-instances. In particular, the VM-level architecture produce bigger overhead than that of the service-level one but better than the other three. This is attributed to the fact that it optimizes per-VM, which is coarser-level than the service-level style. As expected, in case of 6 (S_{12} and S_{51} in addition to the case of 4) and 8 service-instances, the overhead of the sensitivity-aware architecture and the PM-level one is becoming better than that of the cloud-level one. This is because our architecture and the PM-level style tend to produce more regions (as shown in Table 3), which implies that it is able to asynchronously search within a smaller search space for each region with less complexity in contrast to the cloud-level one. On the other hand, the service-level and the VM-level styles remain unaffected. However, we can see that our architecture perform similar to the VM-level style and only slightly worse than the service-level style. In contrast to the PM-level style, our sensitivity-aware architecture still performs better. This is attributed to the fact that we further allow partitioning within a PM. Consequently, this result in one more regions and thus the search space is further reduced. We can see that even with only one more region, the achieved overhead of our architecture gains considerable improvement. We believe that such improvement can be amplified when it is possible to partition more regions.

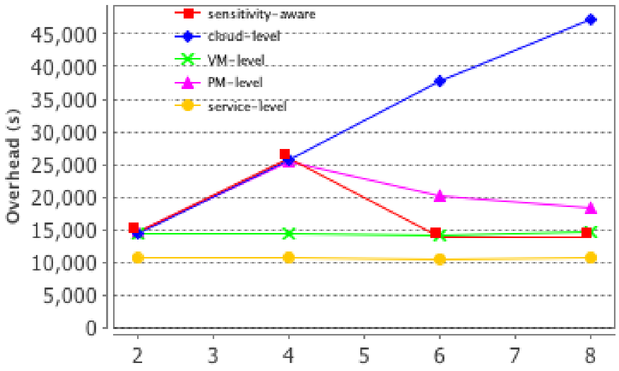

Figure 8. Overhead under different numbers of service-instances.

Interestingly, we can see that unlike the overhead for cloud-level architecture, which increases linearly; the overhead of our sensitivity-aware architecture and the PM-level architecture increase from the cases of 2 to 4 service-instances. They can drop again from the cases of 4 to 6 and remain stable for the case of 8 service-instances. This is because both architectures determine that only one region is allowed for the case of 2 and 4 service-instances. Therefore, it is the same as the cloud-level one and the overhead could also increase in a similar way. When 6 and 8 service-instances exist, both architectural styles result in more than one region. Henceforth, the average result of overhead tends be smaller than previous cases, as there are numbers of elasticity decisions made for a region with smaller search spaces than the single region in case of 2 and 4 service-instances.

To conclude, our symbiotic and sensitivity-aware architecture is able to achieve smaller overhead in contrast to the cloud-level and the PM-level architecture as the number of region increases. The overhead of our architecture is close to that of the service-level and the VM-level style. However, we can observe from Section 5.1 that the achieved global benefit are significantly better than these two. In addition, the experiments reveal that the overhead of our architecture is sensitive to the number of partitioned regions. In particular, the more sensitivity independent regions are partitioned, the smaller overhead is realized.

6. RELATED WORK

The increasing complexity of the elastic management of applications in the cloud urges the need for self-adaptive solutions. Most of the recent research on architecture for self-adaptive cloud rely on classic MAPE, and assume fixed region granularity when partitioning and optimizing for QoS/cost objectives. Up to our knowledge, none of the existing solutions pursue partitioning to optimize for QoS and cost in the cloud based on sensitivity. Our previous work [17] is the first attempt to conceptually capture the requirements for sensitivity-aware architecture in the cloud.

Cloud-level architecture could assume either centralized or decentralized deployment. The elastic strategy tends to be selected by taking the objectives of all cloud-based services into account. [8] propose a cost-optimized, hierarchical architecture that select elastic strategy from the lowest layer in the hierarchy. These strategies are then combined and the final decision is made in a cloud level controller. [9] present a framework for optimizing benefit of all cloud-based services. Their approach optimizes for benefit. The limitation in cloud-level architectures is that the overhead heavily relies on the number of service and the possible elastic strategies to select. As a result, the consensus in decision making could easily become bottleneck. Sensitivity-aware solutions with dynamic region granularity has the potentials to efficiently reach the right strategy and consequently improve adaptation.

On the other extreme, service-level architecture is proposed as they optimize the objectives for each cloud-based service independently with ignorable overhead. [10] propose OPTIMIS, a toolkit for managing cloud-based service using fixed elastic rules. Their approach is cost-optimized and they only assume horizontal scaling of VM, however. [11] describe a service-level architecture for automatically controlling each cloud-based service. Unlike our work, they do not attempt to optimize for QoS and cost. They discuss an automatic way to enable self-adaptive cloud. In addition, they assume limited number of elastic strategies; whereas, we assume arbitrary combinations. The lack of these architecture is that they might not achieve better global benefit for all cloud-based services, as they ignore QoS interference.

The VM-level [12, 13] and PM-level [14, 15] architecture result in acceptable balance between globally-optimal benefit and overhead. In particular, although approaches like [13] aim for per-application, they only assume one application per-VM. Thus, it can be categorized as VM-level architecture. The approach proposed by [12] aim for benefit, however. Unlike our work, they only focus on locally-optimal benefit on each VM. In addition, their QoS modeling rely on static and offline approaches.[13] propose a framework for SLA enactment in the cloud, they have also considered the objective dependency. Nevertheless, they only aim for the fundamental objective-dependency (e.g, throughput and cost per-VM) and do not take the objective-dependency caused by QoS interference into account; our architecture covers both, however. [14] report on a *model predictive control* based approach for elasticity in self-adaptive cloud. They do not intend to take QoS interference into account, however. The approach proposed by [4] has particularly catered for the interference caused by co-hosted VMs. They rely on local optimization per VM. The architecture of [15] assume that hybrid and fixed region granularity; they provide a QoS-optimized architecture for optimization in the cloud based on feedback adaptive loops. In addition, they claim that VM-level architecture should be used for SaaS whereas PM-level one for IaaS and a hybrid one for PaaS.

Unlike the aforementioned work, our architecture reaches better balance between global benefit and overhead by dynamically partitioning the objectives into sensitivity independent regions. We adapt symbiotic loop to realize such. In addition, we assume arbitrary combinations of elastic strategies. In particular, the architecture caters for QoS interference caused by co-located service-instances and co-hosted VMs.

7. CONCLUSION AND FUTURE WORK

We have proposed a symbiotic and sensitivity-aware architecture for dynamically guaranteeing globally-optimal benefit in elastic cloud. In particular, we apply symbiotic feedback loops to the architecture: the architecture is not only able to monitor and adapt better elastic adaptation strategies to the managed services, but also able to adaptively consolidate its sensitivity awareness by re-partitioning the regions for more efficient and effective decision making. Experimentally, we have evaluated our architecture with respect to global benefit achieved by the produced elastic adaptation strategies and the overhead to reach these strategies. We compare the results to other 4 non-sensitivity-aware architectural styles. The results reveal that our architecture produces similar global benefit to the PM-level architecture, and better than other non sensitivity-aware architectures. On the other hand, it produces smaller overhead than the cloud-level and the PM-level architecture; and could be similar to that of the service-level and the VM-level ones. The improvement on global benefit and overhead tends to amplify when it is possible to have more regions. In future work, we will focus on the design of more sophisticated formalization of the global objective function. We will also investigate the other optimization algorithms within the proposed architecture.

8. REFERENCES

[1] Google App Engine, http://code.google.com/appengine/

[2] Amazon Elastic Compute Cloud,http://aws.amazon.com/ec2/

[3] T. Chen and R. Bahsoon "Self-adaptive and sensitivity-aware QoS modeling for the cloud," in Proc. of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 43 –52, May 2013.

[4] CZ. Xu, J. Rao, and X. Bu,. "URL: A unified reinforcement learning approach for autonomic cloud management." Journal of Parallel and Distributed Computing, vol. 72, no. 2, pp. 95-105, 2012.

[5] G. Galante, and LCE. Bona. "A survey on cloud computing elasticity." Utility and Cloud Computing (UCC), 2012.

[6] J.Li, et al, "Profit-based experimental analysis of IaaS cloud performance: impact of software resource allocation," *In Proc. of Conference on Service Computing*, 2012.

[7] Xen: a virtual machine monitor, http://xen.xensource.com/.

[8] M. Kesavan, et al. "Practical Compute Capacity Management for Virtualized Datacenters." IEEE Transaction on Cloud Computing, vol. 1, no. 1, 2013.

[9] JZ. Li, et al. "CloudOpt: multi-goal optimization of application deployments across a cloud." Proceedings of the 7th International Conference on Network and Services Management. International Federation for Information Processing, 2011.

[10] AJ. Ferrer, et al. "OPTIMIS: A holistic approach to cloud service provisioning." Future Generation Computer Systems, vol. 28, no. 1 pp. 66-77, 2012.

[11] J. Kirschnick, JM. Alcaraz Calero, and N. Edwards. "Toward an architecture for the automated provisioning of cloud services." Communications Magazine, IEEE, vol.48, no. 12 pp. 124-131, 2010.

[12] H. Wu, et al. "A benefit-aware on-demand provisioning approach for multi-tier applications in cloud computing." Frontiers of Computer Science, 2013.

[13] M. Maurer, I. Brandic, and R. Sakellariou. "Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures." IEEE Cloud Computing (CLOUD), 2012.

[14] H. Ghanbari, et al. "Optimal autoscaling in a IaaS cloud." Proceedings of the 9th international conference on Autonomic computing. ACM, 2012.

[15] M. Litoiu, et al. "A business driven cloud optimization architecture." Proceedings of the 2010 ACM Symposium on Applied Computing, 2010.

[16] JGroup: A Toolkit for Reliable Multicast Communication, http://www.jgroups.org

[17] T. Chen, R. Bahsoon, and G. Theodoropoulos. Dynamic QoS Optimization Architecture for Cloud-based DDDAS. Procedia Computer Science, 2013.