Introducing Service-level Awareness in the Cloud

Cristian Klein[1], Martina Maggio[2], Karl-Erik Årzén[2], Francisco Hernández-Rodriguez[1]
[1] Umeå University, Sweden, [2] Lund University, Sweden
cristian.klein@cs.umu.se, martina.maggio@control.lth.se,
karlerik@control.lth.se, francisco@cs.umu.se

ABSTRACT

Resource allocation in clouds is mostly done assuming hard requirements, applications either receive the requested resources or fail. Given the dynamic nature of workloads, guaranteeing on-demand allocations requires large spare capacity. Hence, one cannot have a system that is both reliable and efficient.

To solve this issue, we introduce Service Level (SL) awareness in clouds, assuming applications contain some optional code that can be dynamically deactivated as needed. First, we design a model for such applications and synthesize a controller to decide when to execute the optional code and when to skip it. Then, we propose a Resource Manager (RM) that allocates resources to multiple SL aware applications in a fair manner. We theoretically prove properties of the overall system using control and game theory.

To show the practical applicability, we implemented SL aware versions of RUBiS and RUBBoS with less than 170 lines of code. Experiments show that SL awareness may enable a factor 8 improvement in withstanding flash-crowds or failures. SL awareness opens up more flexibility in cloud resource management, which is why we encourage further research by publishing all source code.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements; Modeling and prediction*

General Terms

Design, Management, Service Level, Performance

Keywords

Resource allocation, Service Level

1. INTRODUCTION

Cloud computing radically changed the management of data-centers [7]. In the past, machines used to have one specific purpose. The need for a new functionality, such as a new web application, implied the purchase of a new Physical Machine (PM). This tendency resulted in poor resource utilization and energy waste. However, thanks to the advances in cloud computing technologies, applications are now wrapped inside Virtual Machines (VMs) and consolidated onto fewer PMs [33].

As a result, resource management becomes a key issue. Specifically, it is crucial to decide how the available capacity is distributed among applications to ensure that on-demand resource requests are satisfied with the minimum amount of hardware. In this area, there has been a tremendous amount of work, mostly assuming that application resource requirements are "hard", i.e., the application is either given the needed amount of resources or fails. Combined with the fact that most cloud applications have dynamic resource requirements [39], this imposes a fundamental limitation to cloud computing, which decrease its flexibility: To guarantee on-demand resource allocations, the data-center needs large spare capacity, leading to inefficient resource utilization.

In order to increase the flexibility of resource management, we propose introducing *Service Level (SL) awareness* in clouds. SL aware applications are characterized by a dynamic parameter, the *service level*, that monotonically affects both the end-user experience, as well as the computing capacity required by the application. For example, online shops usually offer end-users recommendations of similar products they might be interested in. No doubt, recommender systems greatly increase the user experience. However, due to their sophistication, they are highly demanding on computing resources [28]. By selectively activating or deactivating the corresponding code, resource consumption can be controlled at the expense of end-user experience.

SL awareness opens up the possibility to deal predictably and efficiently with unexpected events. Unexpected peaks — also called flash crowds — may increase the volume of requests by up to 5 times [5]. Similarly, unexpected failures reduce the capacity of the data center until they are repaired. Also, unexpected performance degradations may arise due to interference among co-located applications [33]. These phenomena are well-known and software is readily written to cope with them, using techniques such as replication and dynamic load balancing, as long as resource provisioning is sufficient [3, 24]. However, given the short duration of

such unexpected events, it is often economically unfeasible to provision enough capacity for them. On the contrary, using SL awareness, the infrastructure can simply ask applications to temporarily reduce their requirements. Consequently, end-user experience is reduced, since the optional code is not executed. However, providing the user with partial content is better than having an unresponsive web application.

SL awareness can be an alternative or a complement to other techniques. For example, *out-scaling* is often proposed as a solution to temporary lack of capacity [18] — requesting VMs from a public cloud provider, such as Amazon EC2 or Rackspace, effectively creating a *hybrid cloud*. SL awareness can be an initial, temporary solution, during the time interval when out-scaling is set up, or an alternative, whenever out-scaling is not an option such as budget constraints or privacy concerns. In fact, with out-scaling, besides the cost for renting the VMs, the owner would also have to pay the cost of transferring her data onto the public cloud and back into the data-center after the unexpected condition expired. Also, the owner may deal with sensitive data, such as company know-how, credit card transactions, user profiles, that are not transferable outside the private data-center. Finally, cloud providers themselves have limited capacity and even Amazon EC2 — one of the largest computing inventories — can run out of capacity [13].

Contributions.

In this article we build the necessary infrastructure and software to support SL aware cloud applications. We discuss a model that captures the behavior of a typical Internet-facing SL aware application. In this model, we define the application behavior and the information that it should sense to perform informed choices on how to change its SL. Subsequently, we propose a Resource Manager (RM) that coordinates the resource allocation among applications competing for the same resources. The highlight of our contribution is that the design is backed up by theoretical results both from game and control-theory. Thus, our system provides specific guarantees on desirable properties such as convergence and fairness among the applications, which translates to withstanding unexpected capacity shortages predictably.

We focus on the resources of a single PM, leaving multiple-PM extensions for future work. Our paper offers the following contributions:

- it proposes a model for SL aware cloud applications which is applicable to any application with optional computations;
- it synthesizes a controller for such applications, which adapts the SL to the load and available capacity;
- it implements a game-theoretical resource manager that balances resource allocations fairly among independent, competing applications;
- it extends two well-known cloud benchmark applications, RUBiS [40] and RUBBoS [6], with an SL aware recommender systems;
- it evaluates the resulting framework showing peak load handling and resource distribution among the SL aware applications.

The results show that SL awareness can allow applications to support 8 times more users or run on 8 times less resources than their SL unaware counterpart. Hence, our proposition enables cloud infrastructures to predictably deal with unexpected peaks or unexpected failures, without requiring spare capacity. Moreover, to foster further research on SL awareness in cloud computing, but also to make our results reproducible, we have made all source code publicly available[1].

The rest of this article is structured as follows. Section 2 positions our work with respect to the state of the art. Section 3 describes the theoretical foundations, which serve as a basis for the implementation presented in Section 4. We evaluate the resulting system in Section 5. Section 6 concludes the paper.

2. RELATED WORK

Managing resources in clouds is a challenging task. The state of the art can be categorized along two different dimensions.

Analytic vs. Experimental models.

Resource management systems rely either on analytical models [4, 11, 45, 53, 54, 56] or on running actual experiments and building empirical traffic profiles and signatures [21, 38, 46, 49, 50]. Our system uses an analytical model to infer performance from measurements taken from the actual system. Zheng et al. [55] argue that running actual experiments is cheaper than building accurate models to validate resource allocation strategies. We build on this assumption, validating our technique on a small scale experimental testbed with a similar attitude.

Application vs. Infrastructure centric.

Resource management schemes are either application or infrastructure-centric. Performing *application-centric* resource allocation [8, 10, 23, 38, 42, 51] means deciding the right amount of resources to allocate avoiding under- or over-provisioning. However, applications are not cooperative and cannot reduce their requirements if resources are congested. In this way, the limitations of the underlying infrastructure are neglected, taking only the application's point-of-view. For example, the resulting resource allocation for a web application only depends on the incoming end-user requests.

[1] `https://github.com/cristiklein/cloudish`

Application-centric allocation can be combined with game theory. For example, Ardagna et al. [1] studies resource allocation in which users bid for resources and the provider sets the price to maximize his revenue. A solution which converges to a Nash equilibrium is proposed. Sharma et al. [41] proposes Kingfisher, a system that tries to minimize the cloud tenant's deployment cost while reacting to workload changes. Kingfisher takes into account the cost of each VM instance, the possibility of horizontal and vertical scaling and the transition time between configurations. However, none of these works take into account the capacity limitations of the cloud provider and do not deal with overload.

Although some works deal with performance differentiation for multiple classes of clients [36], to our knowledge, the only cloud application that comes close to being SL aware is Harmony [12]. Harmony adjust the consistency-level of a distributed database as a function of the incoming end-user requests, so as to minimize resource consumption. This is a specific example of SL awareness in cloud applications, and the adaptation strategy is not reflected in the resource allocation. Also, in this case, the motivation to introduce the adaptation mechanism lies in limiting the amount of money that the user is charged for. Our work is motivated by helping the infrastructure deal with overload conditions.

Infrastructure-centric resource allocation strategies like [16, 22, 35, 43] mostly regard applications as non-cooperative "black-boxes", with hard resource requirements. Among the different contributions to the area, we most closely relate to those dealing with over-subscription (also called over-booking). In [26], application requirements are modeled as random variables and statistical analysis is applied to avoid data-center overload. In [20] the approach is extended with correlation coefficients between the requirements and portfolio theory is used to increase over-subscription, while controlling the overload risk. However, in both of these works no remedy is given to overload conditions, besides having to pay a penalty to the user. A possible solution is presented in [52] by allowing the provider to suspend the least "important" VMs. However, this solution may be unacceptable when the VMs are hosting interactive, Internet-facing applications.

To guarantee the satisfaction of on-demand requests, large spare capacity is necessary. As a solution to this waste, high-priority and low-priority workloads can be combined [34]. However, such an approach only works if the latter have no quality of service requirements. Otherwise, the infrastructure would again have to have spare capacity.

Adapting batch workloads to resource availability has been studied in HPC infrastructures, such as supercomputers. *Moldable* or *malleable* applications [17] can be directed to trade resources for execution time, for example, an application may run on 4 cores for 1 hour or 2 cores for 2 hours. This additional flexibility in resource management has been shown to bring significant gains [25, 27, 44, 47]. However, the amount of computations essentially remains the same. Unfortunately, theses approaches are designed for allocations whose start-time can arbitrarily be delayed. Therefore, clouds, characterized by on-demand resource allocation, cannot readily take advantage of them.

Summary.

To the best of our knowledge, this is the first work that deals with SL aware cloud applications, integrating them with resource allocation. Existing papers either do not study how such applications change their SL and interact with the infrastructure or how the infrastructure coordinates multiple such applications.

3. THEORETICAL FRAMEWORK

This section presents the theoretical framework of our contributions and describes both application behavior and resource allocation among applications. We borrow the general idea from a framework proposed recently in the domain of embedded computing systems [32] extending the framework to handle the complexity of cloud computing infrastructures. We employ terminology and notations used in control theory [30].

Let us start by giving a high-level view of a data-center and where our proposal fits in. A data center is composed of a variable number of Physical Machines (PMs), each PM having a certain capacity which can be allocated to one or more cloud applications. For easier control over allocations and better isolation, each application is sandboxed in its own Virtual Machine (VM), therefore, from now on, we use the terms application and VM interchangibly. A Resource Manager (RM) runs alongside VMs in each PM to decide capacity allocation. Each application runs at a different **Service Level (SL)**, which monotonically affects both the *quality* of the computation and the *capacity* that it requires.

We shall now focus on a single PM and describe the applications' behavior and the RM. Then, we discuss the formal guarantees obtained with our methodology.

3.1 Applications

We assume that every application i is composed of some time sensitive portions of code, called jobs, which have to be executed before a deadline expires. For example, for a web application a job represents a client request, that has to be executed in a timely manner. Each job can contain some *optional computations*. Being able to run optional computations is desirable, as they would improve end-user experience. However, deactivating them is preferred to missing a deadline. The number of times that these optional computations are executed between time k and $k+1$ (and thus the capac-

ity required by the application) is proportional to the SL of the application s_i^k.

Every application is requested to regularly update the RM about their performance. More precisely, a **matching value** should be computed that respects the following properties. First, the matching value should be close to zero when the assigned resources are perfectly matched with the current SL of the application. Second, if the matching value is positive, the resources assigned to the application are abundant and the application can compute at a higher SL, or the amount of assigned resources can be reduced. Third, and dual, if the matching value is negative, either more resources have to be provided or the application should reduce its SL to avoid missing deadlines.

For the application model described above, we chose to compute our matching value as follows. First, a per-job matching value $f_{j,i}^k$ is computed for each job j as

$$f_{j,i}^k = 1 - \frac{t_{j,i}^k}{\bar{t}_{j,i}} \tag{1}$$

where $\bar{t}_{j,i}$ is the desired deadline and $t_{j,i}^k$ is the measured completion time. Next, the per-job matching values $f_{j,i}^k$ are averaged to obtain the application's matching value f_i^k, which is communicated to the RM. This is the only value that the application has to communicate to the RM. It is easy to prove that our choice respects the properties described above.

Our framework can exploit the adaptivity of applications that change their SL to offer an overall better performance. Each adaptive application i may change the SL it runs at, as a function g_i of the current performance, called the **update rule**

$$s_i^{k+1} = g_i(s_i^k, f_i^k) \tag{2}$$

that can be different for each application. Both the SL s_i^k and the update rule g_i are private to the application, i.e., the RM is not informed about them. This assumption allows the RM to run in linear time with respect to the number of applications, resulting in a lower overhead compared to a complex optimization approach where the RM also selects the SLs of the applications. Moreover, this allows applications to customize their definition of the SLs and their update rule. Note that, our framework allows application to be non-cooperative, i.e, SL-unaware, as most existing applications are. For these applications, if no value is communicated, the RM simply assumes the matching value is zero.

To clarify the concepts just introduced, we sketch an e-commerce website as an example of an SL-aware application. In the e-commerce website, we consider the visualization of a page containing one specific product as one job. The optional code of such a job consists in retrieving recommendations of similar products. For each request, besides retrieving the product information, the application runs the recommender system with a probability s_i^k. Increasing s_i^k increases the number of times recommendations are displayed, thus increasing end-user experience, but also the capacity requirements of the application. In the end, the capacity requirements will be roughly proportional to s_i^k.

One of the main differences between this work and similar research in the context of embedded systems [9] is that we do not assume anything about the application's behavior, thus, the RM does not have access to the SL update rules. In fact, our framework is completely general with respect to the choice of the function g_i in Eq. (2). However, for completeness, we propose a couple of examples of how an SL controller can be designed and which guarantees it may offer.

Controller design for average response time.

The first alternative that we present is a controller whose purpose is to maintain the average response time of the requests around a specific setpoint value. Using a very primitive, but useful model, we assume that the average response time for application i, measured at regular time intervals, follows the equation

$$t_i^{k+1} = \alpha_i^k \cdot s_i^k + \delta t_i^k \tag{3}$$

i.e., the average response time t_i^{k+1} of all the jobs that are executed between time instant k and time instant $k+1$ depends on a time varying unknown parameter α_i^k and can have some disturbance δt_i^k that is a priori unmeasurable. α_i^k takes into account how the service level selection affects the response time, while δt_i^k is an additive correction term that models variations that do not depend on our service level choice — for example, variation in retrieval time of data due to cache hit or miss. Our controller design should aim for canceling the disturbance δt_i^k and selecting the value of s_i^k so that our average response time is equal to our setpoint value.

As a first step towards the design, we assume that we know α_i^k and its value is constant and equal to α. We would later be able to substitute an estimation of its current value in the controller equation, to make sure that the behavior of the closed loop system is the desired one. The Z-transform of the plant, i.e., the system that we want to control, before the feedback loop is closed, described by Eq. (3) is

$$z \cdot T(z) = \alpha \cdot S(z) + \Delta T(z) \tag{4}$$

where z is the unit delay operator, $T(z)$ is the Z-transform of the time series t_i^k, $S(z)$ relates to s_i^k and $\Delta T(z)$ transforms δt_i^k. We cannot control the disturbance $\Delta T(z)$, therefore, we are only interested in the transfer function from the input (the SL) to the output (the measured average response time), which is

$$P(z) = \frac{T(z)}{S(z)} = \frac{\alpha}{z}. \tag{5}$$

Every closed loop system composed by one controller and a plant has the generic transfer function

$$G(z) = \frac{C(z) \cdot P(z)}{1 + C(z) \cdot P(z)} = \frac{Y(z)}{R(z)} \quad (6)$$

where $C(z)$ is the transfer function from the error, i.e., the difference between the setpoint and the measured value, to the control signal, in this case, the SL to be applied in the next time interval. $P(z)$ is the plant transfer function [30] — in our case Eq. (5). $Y(z)$ and $R(z)$ are respectively the output and the input of the closed loop system, in our case the measured and the desired average response time. The function $G(z)$ represents the response of the controlled system with the feedback loop closed.

The next step in controller design consists in deriving an equation for $C(z)$. One possible strategy is to choose $C(z)$ so that some properties on $G(z)$, the response of the controlled system, are satisfied — in control terms, to select the "shape" of the response. For example, we want the steady state gain of $G(z)$ in Eq. (6) to be one, since we want the setpoint to be equal to the output. Also, we want to introduce a stable pole in the closed loop system, to control the speed of the response — in order for the system to be stable the pole should lay within the unit circle, in order to also avoid oscillations its value should be between zero and one. Assuming that we want to introduce the stable pole in p_1, our desired closed loop transfer function looks like

$$G(z) = \frac{C(z) \cdot P(z)}{1 + C(z) \cdot P(z)} = \frac{1 - p_1}{z - p_1} \quad (7)$$

and substituting the plant transfer function of Eq. (5) into Eq. (7) we can derive the expression $C(z) = \frac{1-p_1}{\alpha(z-1)}$ for the controller. By applying the inverse Z transform on the expression of $C(z)$, we obtain

$$s_i^{k+1} = s_i^k + \frac{1 - p_1}{\alpha} \cdot e_i^{k+1} \quad (8)$$

where e_i^{k+1} is the difference measured at time $k+1$ between the setpoint for the response time and its measured value. This equation can be used to implement a controller that selects the SL.

The pole p_1 can be used to choose between stability and reactivity. The closer p_1 is to one, the slower the system responds, but the better it rejects measurement noise or other disturbances. Effectively, the controller will only make small SL corrections at every iteration. In contrast, values of p_1 close to zero will make the system respond quickly, but also be more sensitive to disturbances, making large SL corrections, that risk being based on transient disturbances instead of long-term trends. Some values for p_1 can be suggested, depending on the reliability of the measurements and the variability of the incoming requests. However, selecting a value for p_1 should either be done by the controller developer or the application operator, based on empirical testing, so as to fulfill her specific needs.

Now, the only open issue is to provide an estimation of α_i^k. We can use many methods to estimate it online, while executing. The most simple is to take past measurements and compute the average response time t_i^{k+1}, pretend the disturbance δt_i^k is negligible and compute α_i^k based on Eq. (3). Once a first estimation is available, it is also possible to assign a weight to new data points and choose

$$\alpha_i^{k+1} = (1 - \mu) \cdot \alpha_i^k + \mu \cdot \frac{t_i^{k+1}}{s_i^k} \quad (9)$$

where μ is a discount factor that defines how trustworthy the new observations are.

Controller design for maximum response time.

Controlling maximum response time, instead of average, has become the focus of recent research [15]. Our controller can easily be adapted to this purpose, by using $e_i^{k+1} = \bar{l}_{i,\max} - l_{i,\max}^k$ in Eq. (8), where $\bar{l}_{i,\max}$ is the desired maximum response time and $l_{i,\max}^k$ is the maximum response time measured from time instant k to $k+1$. In what follows, we shall use this controller.

3.2 Resource manager

The role of the resource manager is to select the capacity of the PM that each application is allowed to use. In many works cited in Section 2, cloud resource allocation is done based on *monitored* resource usage. However, this approach cannot be used to support SL-aware applications. For example, when an application's CPU usage is low, without additional information, the RM cannot distinguish whether the application is abundantly provisioned and runs at maximum SL, or insufficiently provisioned but runs at low SL to compensate. Therefore, in our case, the RM does not directly monitor the resource usage by the applications but uses information on the applications' performance that are conveyed through the matching value defined in Eq. (1)[2] without needing to know the SLs of the applications.

Let us now describe the RM's behavior. We denote with $c_i^k \in [0,1]$ the capacity assigned at time k to the i-th application relative to the total capacity C of the PM[3]. The RM enforces that the total allocated capacity does not exceed the available one:

$$\sum_i c_i^k \leq 1 \quad (10)$$

[2]As long as the used matching value respects the three properties defined when we introduced Eq. (1), its formulation can be changed.

[3]We use the same time index k to express the applications' and the RM's update rules. However, the update periods of each application and of the RM are completely independent.

At initialization, the RM sets the capacities to $c_i^0 = 1/n$ where n is the number of applications. Subsequently, at each time step k, it first retrieves measurements for all the matching values f_i^k — as the averaged values of Eq. (1) — then updates each capacity according to

$$c_i^{k+1} = c_i^k - \varepsilon_{rm}(f_i^k - c_i^k \cdot \sum_i f_i^k) \quad (11)$$

where ε_{rm} is a constant that determines the step-size of the RM. The sum of the capacities remains equal to one, therefore continuing to enforce the total capacity constraint. Since the matching values of the applications are closer to zero when the resources they receive match their SLs, the new allocation favors the applications that are more distant from their target performance values — whose matching values are more negative. The new resource allocation reflects the relative distance between the applications' performance. Finally, the computed relative capacities c_i^k are multiplied by the total capacity C, to obtain the absolute values C_i^k. The RM itself needs to make sure that it gets enough resources to function correctly, either by reserving some capacity for itself, or by running with a higher priority than the applications.

As can be observed in Eq. (11), the RM's complexity is **linear** with respect to the number of applications, which allows its implementation to have low overhead.

3.3 Convergence analysis

In this section we briefly summarize the relevant contributions in terms of convergence analysis. It can be shown that the RM allocations converge to a **stationary point**, that is characterized by the following property: Applications are either performing sufficiently well, which means that their matching values are close to zero, or are poorly performing but already operate at minimum service level. This means that if a stationary point where all the matching values of the running applications are driven to zero exists, this point is reached.

Moreover, the RM ensures **fairness** among applications. Whenever the applications have similar definition for their matching values, the framework theoretically guarantees that, in case of overload, the resources assigned to the applications converge to equal values. In other words, applications contribute equally to dealing with the overload.

For the complete formal analysis of the framework, we refer the reader to the proofs in [9, Section IV].

4. IMPLEMENTATION

In this section, we show how we implemented our framework building a working prototype. We first describe the general architecture and then move to the introduction of SL awareness in two popular cloud applications: RUBiS [40] and RUBBoS [6].

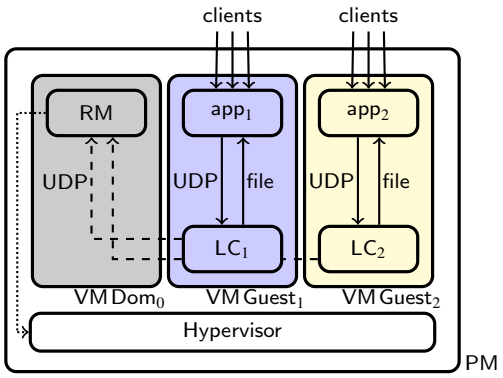


Figure 1: Architecture of a single Physical Machine (PM).

4.1 Architecture

Figure 1 presents an overview of the architecture of our framework. As anticipated, we focus on managing a single PM and its applications. This PM is equipped with a *hypervisor* that is capable of setting hard limits on the resource consumption of all the VMs hosted inside the PM. A privileged VM, called Dom_0 runs with high-priority and executes the RM. Alongside this VM, n unprivileged ones run simultaneously and are identified by Guest_i where i is a number between 1 and n. Dom_0 collects performance data from the applications, i.e., the matching value described in Eq. (1), computes the new resource distribution according to Eq. (11) and sets the hard limits for the running applications through a hypervisor-specific interface, such as the one provided by `libvirt` [31].

In our current implementation, we focus on sharing processing resources. Therefore, we use the Xen hypervisor [2] as it offers a versatile interface for controlling such allocations. At a course level, Xen supports dynamically changing the number of virtual cores of a VM and their mapping on physical cores. At a fine level, the credit scheduler can be employed [29] to set a limit for the percentage CPU time, through the `cap` attribute. For example, to allocate 4 and a "half" cores to a VM, i.e., let the VM receive computing power somewhere between having exclusive access to 4 physical cores and 5 physical cores of the PM, the RM instructs Xen to change the number of virtual cores to 5, the rounded up value, and set `cap` to 450. Initial testing with a web application and a compute intensive application showed that this method offers good performance proportionality, i.e., the number of requests per second or the number of floating-point operations per second, respectively, are roughly proportional to the allocated CPU resources of the VM. In what follows, we shall only report the `cap` implying that the number of virtual cores has been adjusted accordingly. Although, we focus on processing resources, our implementation can easily be extended to control other virtual resources, such as network and disk bandwidth, being limited only by the capabilities

of the hypervisor.

Each Guest_i VM hosts a complete Internet application. We do this simplification to make sure that the performance of an application only depends on the resources allocated to its VM. For example, all tiers of an n-tier application have to be contained inside a single VM. Allocating resources to multiple, performance-interdependent VMs is left as future work. Besides the application itself, each VM hosts an application-specific **Local Controller (LC)**, whose purpose is to monitor the performance of the application, control its SL and communicate to the RM the matching value of the application. We wanted our implementation to be minimally intrusive, therefore, we use UDP sockets to communicate between the LC and the RM. Alternatively, an efficient shared-memory mechanism can be set up, requiring "punching" a hole through the virtualization layer. The communication between the LC and the hosted application is not constrained in any way by our RM implementation.

Regarding the RM's algorithm, it follows the specification in Section 3.2, with two minor adjustments that improve stability: matching values are saturated between $[-1,+1]$ and allocations are unchanged if all matching values are positive. These modifications preserve the theoretical results related to convergence and fairness.

4.2 Applications

We now move to the description of how individual applications can take advantage of our framework. We start by showing the ease of complying with the framework rules by taking two widely used examples of cloud benchmark applications — RUBiS [40] and RUBBoS [6] — and making them SL aware.

RUBiS [40] is an extensively used benchmark that implements an auction website, similar to eBay. It has been widely used to evaluate resource allocation schemes, see for example [11, 23, 43, 45, 46, 49, 55].

We built a service-level aware RUBiS version, extending the PHP implementation and in particular the `ViewItem.php` page. Our extension features a simple recommender system: When the application receives a request for an item j, if the recommender system is active, it retrieves the set of users U_j that bid on the same item in the past. Then, the recommender system composes the set of recommended items R_j, retrieving other items that the users in U_j have bid on. The items in R_j are ordered by popularity, i.e., the number of bids on them, and the top 5 are returned. It is outside the scope of this article to propose a sophisticated recommender system. The described recommender simply serves as a reasonable example of an optional code that a cloud application may enable or disable at runtime. Clearly, such a recommender system adds a great value to the user experience. However, it is also resource hungry. Initial testing revealed that 20 times fewer concurrent users can be served when the recommender is enabled.

The presence of the potentially invoked recommender systems allows us to make the auction website SL aware. We define s^k_{RUBiS} as the SL of the RUBiS application at time k. This value is comprised in the interval $[0,1]$ and represents the per-request probability that the recommender system is executed between time instant k and $k+1$. As a consequence, the higher the SL, the higher the resource consumption, but also the better the user experience, which is a necessary feature to support SL aware applications using our framework. Having an SL of 0 means that no user will see recommendations, while a value of 1 means that all the users will receive recommendations. When serving a request arrived between time k and time $k+1$, the PHP script reads the current service-level s^k_{RUBiS} from a file and generates a random number in $r \in [0,1]$: If $r < s^k_{RUBiS}$, recommendations are displayed, otherwise the recommender system is not executed.

To make this application useful to the user, we need to control the user-perceived latency [48]. To this end, the beginning and the end time of each "view item" request are recorded and, by subtracting the two, the response time can be measured. While this quantity is slightly different than the user-perceived latency — due to network latencies, queuing at the web server, context switches and other external factors — it should be reasonably close, assuming the application is not overloaded. Ensuring the absence of overload conditions depends on how the feedback loop between the measured latency and the SL is closed, which is the role of the Local Controller (LC).

The aim of the LC is to set the SL so as to keep the maximum user-perceived latency $l_{\max}$ smaller than a given amount $\bar{l}_{\max}$. A study on web-user **tolerable waiting-time** suggest using $\bar{l}_{\max}$ between 2 and 4 seconds [37]. The LC controller receives all the measured latencies from the PHP script through a local UDP socket and selects the new SL based on Eq. (8), with $e_i^{k+1} = \bar{l}_{\max} - l_{\max}$. It then atomically writes the new SL in a file using the `rename` system call. The communication protocol is sufficiently efficient for our needs (the small file is served from cache) and has been chosen to avoid intrusive modifications to the RUBiS code. Indeed, our modifications to the original application, which include the recommender system and the controller, are limited to 170 logical SLOC.

It is important to note that the recommender system introduces a large variability in the perceived latency. If a product is unpopular, our recommender returns results in approximately 1ms on a testing system. On the contrary, producing a recommendation for a popular item might take 80ms. We decided to keep this

behavior, instead of limiting the problem, to increase the difficulties (i.e., the disturbances) that the LC has to face and to stress test our system. As a consequence, given constant incoming load and resources, there is no value for s^k_{RUBiS} which maximizes the number of displayed recommendations, while guaranteeing that all latencies are below the set limit. Therefore, a website administrator must choose between a more conservative controller, that prefers keeping latencies low, or a more aggressive one, that prefers serving more users with recommendations, but risks a few high latencies. Choosing an optimal trade-off would require user-behavior studies that model revenue as a function of latency and recommendations and is out-of-scope. Instead, we show in the evaluation the behavior of two different controllers, a more conservative and a more aggressive one.

RUBBoS [6] is a bulleting-board prototype website modeled after Slashdot and has been used as a benchmark in cloud computing research [14].

Introducing SL awareness in RUBBoS can be done with a similar strategy as used for RUBiS, focusing on the "view story" page. However, RUBBoS offers even more flexibility. As a first step, we added a recommender system, that suggests other stories that might be related or interesting for the reader, based on common commentators. As a second step, comments can be disabled. While the comments section is an essential part of such a website, users are better served without it than not at all in case of data-center overload.

In our SL enabled RUBBoS application, a story can be served in three different modes. The lightest version serves neither comments nor recommendations. On the contrary, whenever comments are enabled, the website can retrieve a story with or without recommendations.

Again, we want enabling SL awareness to be as non-intrusive as possible. Therefore, we use a similar approach to the one adopted for RUBiS and define $s^k_{RUBBoS} \in [0,1]$ as the SL of the RUBBoS application at time k. This value represents two different probabilities. First, it stands for the probability that the comments are added between time instant k and time instant $k+1$. Second, if the story is presented with comments, the same value represent the probability that the recommender system is executed, i.e., the unconditional probability is $\left(s^k_{RUBBoS}\right)^2$. Similarly to RUBiS, we can readily use the same LC which maintains maximum latency below a set value.

To sum up, based on the experience acquired with two widely used prototype applications, we believe that enabling SL-awareness in production software should not be too difficult, once the optional parts have been identified. Indeed, for both RUBiS and RUBBoS, this can be achieved with 170 logical SLOC. The benefits of doing these changes are highlighted in the next section.

5. EVALUATION

In this section, we test two different aspects of our implementation, backing up the theoretical results of Section 3. Here, we describe our experimental setup. We continue discussing the behavior of the LC in Section 5.1 and of the RM in Section 5.2.

Experimental Setup.

Our testbed is a single PM equipped with two AMD Opteron™ 6272 processors[4] and 56 GB of memory, which hosts several VMs. We used Xen 4.1.2 as a hypervisor and Ubuntu 12.04.2 LTS 64-bits with Linux kernel version 3.2.0, both for the privileged $\mathsf{Dom_0}$ and the unprivileged $\mathsf{Dom_U}$ VMs. Every unprivileged VM is configured with 4 GB of memory and a variable number of virtual CPUs. The number of virtual CPUs is determined as a function of the **cap** parameter, as described in Section 4.1 — a cap of 400 means that the VM has exclusive access to 4 cores of the PM, while with $cap = 50$ the VM has access to a single core of the PM, but only for half of the time. We deployed our SL-aware versions of RUBiS and RUBBoS, each inside a single VM, and the RM inside $\mathsf{Dom_0}$. Each application's VM contains the LC for that application and all tiers belonging to it — Apache web server, PHP interpreter, MySQL server.

To simulate the users' behavior, we have found the clients provided by RUBiS and RUBBoS insufficient for our needs. Specifically, they do not allow to change the number of concurrent users and their behavior at runtime. Moreover, they report statistics for the whole experiment and not as time series, therefore, we could not observe transient phases. Last, these tools cannot measure the number of requests that have been served with recommendations or comments, which represents the service levels as perceived by the users.

For these reasons, we developed a custom tool, `httpmon`, to simulate web users. Its behavior is similar both to the tools provided with RUBiS and RUBBoS, and to the TPC-W benchmark specification [19]. Specifically, it allows to dynamically select a **think-time** and a **number of users** and maintains a number of client threads equal to the number of users. Each client thread runs an infinite loop, which waits for a random time and then issues a request for an item or a story. The random waiting time is chosen from an exponential distribution, whose rate is given by the think-time parameter. A master thread collects information from the client threads and prints statistics for the last second of execution. Specifically, it records the maximum **perceived latency** — which is the time elapsed from sending the first byte of the HTTP request to receiving the last byte of the HTTP response — and the **perceived SL** — the ratio of requests that have been served executing the optional code for recommendations

[4] 2100 MHz, 16 cores per processor, no hyper-threading

or comments. Note that, this may be different from the **target SL**, which is the output of the local controller.

Since we are interested in studying how well the framework controls CPU resources, we made sure that network or disk did not influence our results. Therefore, we ran our workload generator inside Dom_0 on a dedicated core. Furthermore, we disabled logging and made sure that each VM had enough memory to keep the whole database in-memory. Indeed, disk activity measured during the experiments was negligible.

5.1 Local Controller

This section is dedicated to the test of the LC. We report three different experiments, in each of them we focus on a different time-varying aspect. First, we vary the resources that the application can use — its cap. Second, we vary the application load — the number of connected users. As a third experiment, we vary both these quantities together to simulate the real execution environment. Since previous research suggests that unexpected peaks vary considerably in nature [5], we manually chose values for load and resources that exposed the application to extreme conditions.

The following figures are structured as follows. The x-axis represents the time elapsed since the beginning of the experiment, its numerical value being shown on the bottom. Every experiment consists of 5 intervals, each of them lasting 100 seconds. The experimental parameter that is changed for every interval and its value are reported on the top x-axis. Each figure shows three different metrics. First, the maximum perceived *latency* $t_{j,app}$ is shown in continuous blue lines and its y-axis is depicted on the left side of the plot. Second, the right y-axis hosts two different curves for the SL. The first one, $s_{j,app}$, is the *target SL* set by the controller and is shown in dotted red lines. The second one, $s^*_{j,app}$, depicted in dashed black lines, is the *perceived SL* or the average ratio of pages served with optional content. To improve the readability of the graphs, the values are aggregated over a 10 seconds interval, using the maximum for the latency and the average for the SLs.

Each figure represents a single experiment. We refrained from applying statistics over several experiments with the same parameters, as they would hide some interesting features, such as oscillations and transient behaviors. Ideally, the controller should maximize the target SL while keeping the latency below the tolerable waiting time of 2 seconds as recommended in [37]. Also, the user SL should closely follow the target one.

Constant Load and Variable Resources.

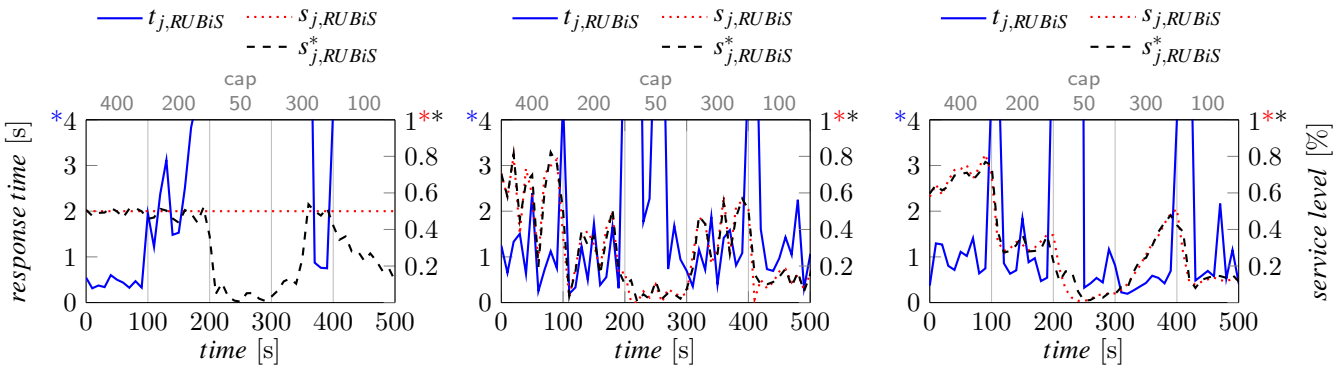
In this set of experiments, we keep the load constant — 100 concurrent users with a think-time of 3 seconds — and vary the cap of our SL-aware version of RUBiS. In a real environment, this situation arises when the capacity of a PM is saturated, before load is redistributed. In fact, when multiple VMs share the same machine, performance interference may occur [38]. Also, CPU throttling due to cooling failures or VM migration due to failures of a different PM may cause similar VM capacity reductions.

The LC was configured with a control period of 1 second to allow a quick reaction, a target latency of 1 second to allow a safety distance to the tolerable waiting time, a discount factor $\mu = 1$ and a sliding measure window of 5 seconds. This means that the controller's input is the error between the desired value of 1 second and the maximum latency measured within the application itself over the last 5 seconds. First, we test the behavior of the system when no LC is present, with a constant service level of 0.5. This means that the application is SL-unaware and serves a request with recommendations only half of the time. We chose to have a fixed value of 0.5 for the SL because that value was optimal in the initial conditions, allowing us to demonstrate the importance of adaptation. We compare these results to the case of an SL-aware application, when the controller's pole p_1 is set to 0.5 and 0.9. We intend to show the advantages that SL-awareness brings in comparison to lack thereof, and highlight trade-offs in the controller's parameter choice.

Figure 2 plots the results of our test. Figure 2a shows the situation when no control is present. The system performs quite well in the first interval, up to 100 seconds, when resources are abundant as the cap is set to 400. Indeed, the latency is below 2 seconds and the user perceived SL is closely following the target one of 0.5. However, during the next interval, when resources are slightly insufficient as the cap is halved, the latency starts increasing, because the application is unable to serve requests fast enough and queues them. In the next time interval, when even fewer resources are available, the system becomes unresponsive and some users experience huge latencies, up to 10 seconds[5]. The perceived SL is very low, as requests that would potentially receive recommendations are abandoned by the client due to timeouts. When, in the next interval, more resources are assigned to the web application as its cap is increased to 300, the latency decreases and the queue is drained. However, this process takes 60 seconds, during which the application seems unresponsive. In the last interval, resources are insufficient and the application becomes again unresponsive.

Figure 2b plots the results with the controller configured with the pole p_1 at 0.5. In contrast to the system without control, the application is perceived as more predictable from the user's perspective. Effectively, it managed to maintain the latency below 2 seconds for most of the time, despite a factor 8 reduction of re-

[5]We limit the plot to 4 seconds to ease comparison with the other scenarios.



(a) Without the LC. (b) $p_1 = 0.5$. (c) $p_1 = 0.9$.

Figure 2: Response time and probability of executing the recommender system varying the VM cap.

sources, from a cap of 400 to one of 50. This is due to the target SL adjustment that follows the available capacity. Furthermore, the perceived SL closely follows the target one.

We discuss here the few deviations from the desired behavior, where the latency increases above the tolerable waiting time. The highest deviations occur as a result of an overload condition, when the cap is reduced, around time instant 100, 200 and 400. This is in accordance with theory, since the controller needs some time to measure the new latencies and correspondingly select the new SL. Nevertheless, the system quickly recovers from such conditions, in less than 20 seconds. Around time instant 50, 240 and 480, the controller seems to be too aggressive. It tends to increase the SL quickly, violating therefore the 2 seconds tolerable latency. Such an aggressive controller is preferred when users are more tolerant to latencies and maximizing SL is a priority.

Whenever users are less tolerant to sudden increases in the latency, the proposed controller can be configured to be more conservative. Figure 2c plots the results with the same controller configured with $p_1 = 0.9$. As predicted by theory, it reacts slower, with small adjustments at every iteration. Its output oscillates less and it generally does a better job at keeping the latency around the setpoint of 1 second. By using this controller, the likelihood of having latencies above the tolerable waiting time is decreased. However, this configuration also recovers slower from overload conditions. Compared to the previous configuration, it required twice as much time to react to the cap reduction at time instant 200. Also, during recovery, the perceived SL differs from the target one. This happens because the responses arriving at the client have a high latency and were actually started at a time when the target SL was higher. Considering that the resources were reduced instantaneously by a factor of 4, the slower recovery is unlikely to be a problem in a production environment.

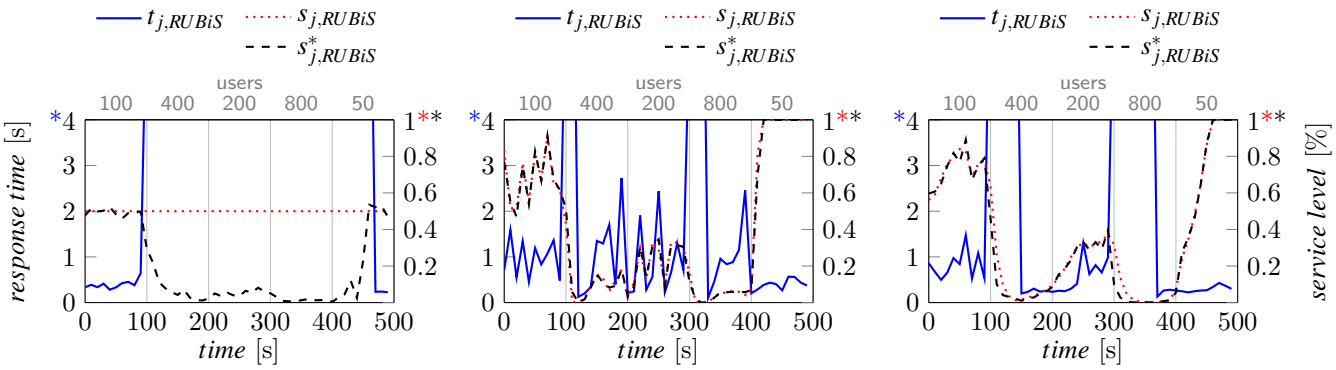
Summarizing, making an Internet-facing application service-level aware may considerably improve its flexibility with respect to resource allocation. Effectively, the application behaves a lot more predictably and can withstand reduction in resource allocation by a factor of 8, compared to the minimum amount of resources that its SL-unaware counterpart would require.

Constant Resources and Variable Load.

In this second set of experiments, we keep the resources constant, setting the cap to 400, and vary the number of users accessing our SL-aware RUBiS page[6]. In a real data-center, this situation may happen due to flash crowds — sudden increase in popularity when the page is linked from another high-profile website. However, it can also be the result of load redistribution due to a failing replica or denial-of-service attacks. The controller is configured identically to the previous set of experiments.

Let us now discuss the results. Figure 3a shows the results without any control, when the system cannot keep up with the load increase. Even after the number of users is significantly decreased, such as at 400 seconds, the application requires a significant time to recover, up to 62 seconds. In contrast, Figure 3b and Figure 3c shows the results with the enabled LC. Despite an 8-fold increase in the number of users from 100 to 800, the application is more responsive, adjusting the service level to adapt to the load increase. Regarding the adaptation time, in the worst interval, when the number of users was spontaneously increased by a factor of 4 at time 300, the controllers required respectively 22 seconds and 66 seconds when $p_1 = 0.5$ and 0.9. As in the previous experiment, the controller with a pole of 0.5 is more aggressive, quickly increasing the target SL and risking latencies above the tolerable level. In contrast, setting the pole to 0.9 produces a more conservative controller, which does smaller SL adjustments. The latency presents smaller oscillations and is less likely to increase above the tolerable waiting time. In the

[6]To simplify the discussion we change the number of users as a single parameter, keeping the think-time constantly equal to 3 seconds. Equivalently, we could have kept the number of users constant and varied the think-time.



(a) Without the LC. (b) $p_1 = 0.5$. (c) $p_1 = 0.9$.

Figure 3: Response time and probability of executing the recommender system varying the number of users.

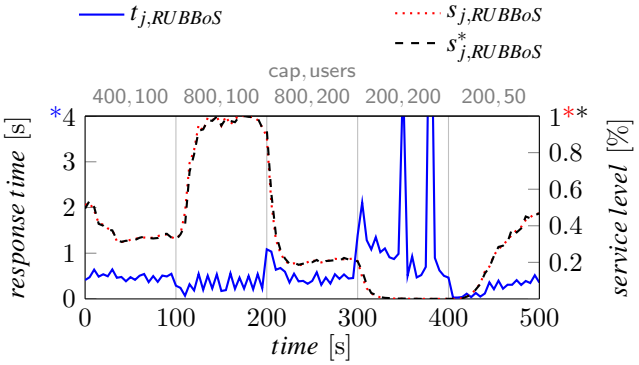


Figure 4: Controlling RUBBoS when both the cap and the number of users vary, $p_1 = 0.9$, setpoint 0.5 seconds.

end, the proposed controller design enables a trade-off between minimizing latency spikes and maximizing SL through the choice of the pole, chosen depending on how tolerant users are to latencies.

To sum up, SL-awareness increases the number of users that an Internet-facing application can serve by a factor of 8, when compared to the maximum number of users served without SL-awareness.

Variable Load and Resources.

To reproduce a realistic setup, we studied how the LC behaves when both the cap and the number of users are varying. We present the results of an experiment conducted with RUBBoS, incidentally showing also another SL-aware application. The perceived SL represents the ratio of pages with comments activated[7].

We select the pole to be 0.9. To further show the flexibility of the controller design, we chose to reduce the desired latency to 0.5 seconds. The rationale behind this choice is that this application serves a different purpose compared to RUBiS. A RUBiS user often has a clear purpose, such as buying an item through the auction website. In contrast, a RUBBoS user might have no clear purpose, like reading news. It is conjectured that users are less tolerant to latencies if they are browsing the web without a clear objective [19]. Hence, by directing the RUBBoS controller to keep latencies lower, we potentially increase the duration that a user stays on the website and, thus, the income of the website owner.

Figure 4 shows the results of the experiment. As can be observed, except for the fourth interval, the controller successfully manages to keep the maximum latency around 0.5 seconds with very small oscillations. In the fourth interval, the SL is kept as close as possible to zero to serve the maximum number of requests in a reasonable time. In general, the SL is increased when the conditions allow for better performance, for example during the second and fifth interval, and decreased when the capacity is insufficient or the load is too high, during the remaining intervals.

These results show that the LC adapts the SL to the available capacity and number of users as expected, and keeps the perceived latencies close to the target value.

5.2 Resource Manager

In this section, we evaluate the behavior of the RM. The platform is limited to 4 cores of the PM, on which we deploy both the SL-aware RUBiS and RUBBoS. Their caps are selected by the RM according to the matching values they send. The two local controllers are configured with the control period to 1 second, a pole at 0.9 and the setpoints 1 second and 0.5 seconds, respectively. The RM's control period is set to 5 seconds and ε_{rm} is 0.2. During the experiments, we vary the number of users accessing the two services and observe the behavior of the LCs and the RM.

Figure 5, displaying the results, is structured as follows. 4 metrics are plotted as a function of time for each of the two applications: the cap chosen by the RM, the matching value computed by the LC, the perceived SL and the maximum latency. The vertical bars represent time intervals during which the number of users is kept constant, with values listed on top. At time instant 0, the experiment starts in its default configuration: Each application is allocated half of the platform and both SLs are 0.5. Since the load on RUBBoS is low, its LC

[7]For increased readability of the figures, we do not plot the ratio of pages with recommendations of similar stories.

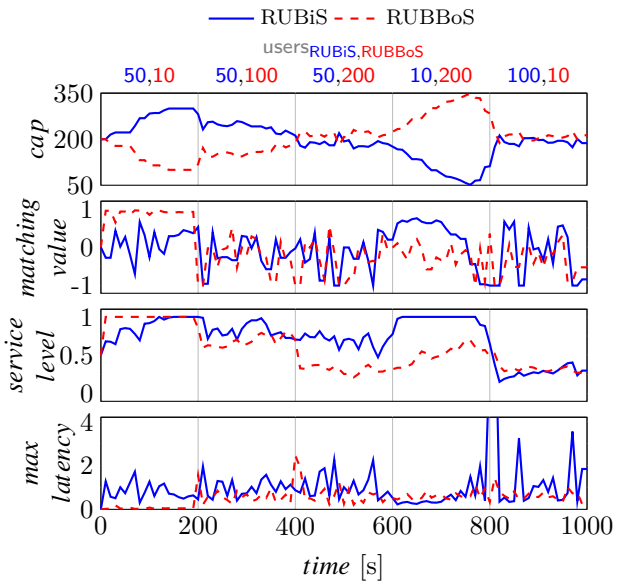


Figure 5: Resource manager and two applications.

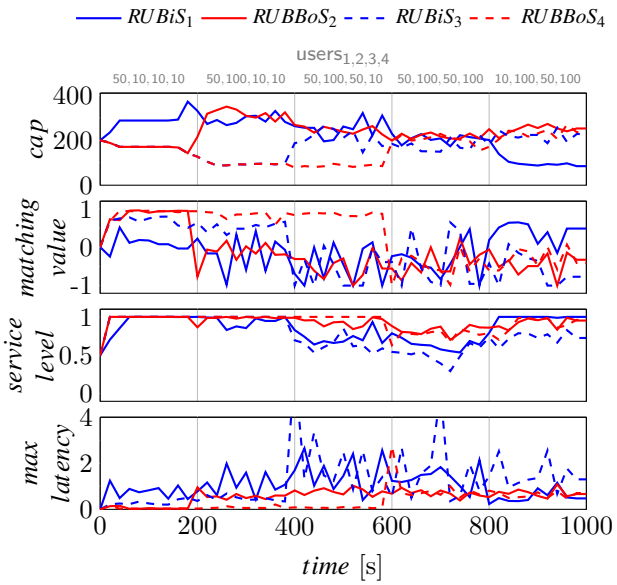


Figure 6: Resource manager and four applications.

increases the SL to maximum. Similarly, the RUBiS LC will try to increase the SL, however, it has insufficient resources to do so immediately. The RM detects this conditions, through the transmitted matching value, and rebalances the platform, so as to reduce RUBBoS's cap and increase RUBiS's cap. Thanks to this, the system reaches a configuration in which both applications may run at maximum SL. At time instant 200, we increase the number of RUBBoS users. Its LC reacts to avoid overload and reduces the SL. Furthermore, the RM increases its cap and decreases RUBiS's cap, whose LC reduces the SL to deal with the new resource allocation. Thus, the system oscillates around a stationary point, in which the performance requirements of both applications are satisfied. Indeed, both RUBiS and RUBBoS users experience maximum latencies around the configured setpoint of each application.

To test the fairness of the system, we conducted an experiment with 4 SL-aware applications, 2 RUBiS and 2 RUBBoS VMs, and a platform consisting of 8 cores. As can be seen in all intervals of Fig. 6, applications that do not run at full SL are assigned equal caps, whose value we call *fair cap*. In other words, despite targeting different setpoints and executing different code, applications that reduce their SL to deal with the infrastructure's overload contribute with an equal amount of resources to overload reduction. This is easily observed for application 1, 2, 3 and 4 in the 4th interval, whose caps oscillate around 200 or applications 2, 3, 4 in the 5th interval, whose caps oscillate around 230.

Some applications may be able to run at full SL with fewer resources than the fair cap. For these applications, their cap is reduced to the minimum value which allows them to run at full SL. Thus, such applications contribute with even more resources to overload reduction, without sacrificing their SL. For example, application 1 in the 5th interval runs at full SL with a cap around 98, which is smaller than the fair cap of 230.

To summarize, the RM behaves as predicted, managing to balance the resources fairly among multiple competing SL-aware applications.

6. CONCLUSION

In this paper, we introduced Service Level (SL) awareness in cloud platforms. On the application-side, we discussed a model for applications with optional code, i.e., computations that can be activated or deactivated per client request. We described our experience with two widely-used, prototype web applications, RUBiS and RUBBoS, showing the applicability of our approach to many Internet-facing applications. We synthesized a controller to select the SL based on incoming load and available capacity and proved its correctness with control-theoretical tools. On the intrastructure-side, we proposed a game-theoretic Resource Manager (RM) to coordinate the demands of multiple applications in a predictable and fair way. We implemented the framework and tested it with real-life experiments. The results show that SL awareness can allow applications to support 8 times more users or run on 8 times fewer resources than their SL unaware counterpart. Hence, our proposition enables cloud infrastructures to predictably deal with unexpected peaks or unexpected failures, without requiring spare capacity.

Future work include extending the contribution to multiple machines and combining SL awareness with other mechanism like migration and horizontal scaling. We believe that SL awareness opens up a new level of flexibility in cloud platforms, whose full potentials need to be further studied.

References

[1] D. Ardagna et al. "A game theoretic formulation of the service provisioning problem in cloud systems". In: *WWW*. 2011.

[2] P. Barham et al. "Xen and the art of virtualization". In: *SOSP*. 2003.

[3] L. A. Barroso et al. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[4] M. Bhadauria et al. "An approach to resource-aware co-scheduling for CMPs". In: *ICS*. 2010.

[5] P. Bodik et al. "Characterizing, modeling, and generating workload spikes for stateful services". In: *SOCC*. 2010.

[6] *Bulletin Board Benchmark*. URL: `http://jmob.ow2.org/rubbos.html`.

[7] R. Buyya et al. "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility". In: *Future Generation Computer Systems* 25.6 (2009).

[8] E. Caron et al. "Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients". In: *J. Grid Comput.* 9.1 (2011), pp. 49–64.

[9] G. Chasparis et al. "Distributed Management of CPU Resources for Time-Sensitive Applications". In: *ACC*. 2013.

[10] L. Y. Chen et al. "Achieving application-centric performance targets via consolidation on multicores: myth or reality?" In: *HPDC*. 2012.

[11] Y. Chen et al. "SLA Decomposition: Translating Service Level Objectives to System Level Thresholds". In: *ICAC*. 2007.

[12] H.-E. Chihoub et al. "Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage". In: *CLUSTER*. 2012.

[13] Compute Cycles. *Lessons learned building a 4096-core Cloud HPC Supercomputer*. Mar. 2011. URL: `http://blog.cyclecomputing.com/2011/03/cyclecloud-4096-core-cluster.html`.

[14] W. Dawoud et al. "Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning". In: *Global Trends in Computing and Communication Systems*. Vol. 269. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, pp. 11–25.

[15] G. DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007), pp. 205–220.

[16] A. Fedorova et al. "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler". In: *PACT*. 2007.

[17] D. G. Feitelson et al. "Toward Convergence in Job Schedulers for Parallel Supercomputers". In: *Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1996, pp. 1–26.

[18] A. J. Ferrer et al. "OPTIMIS: A holistic approach to cloud service provisioning". In: *Future Generation Computer Systems* 28.1 (2012), pp. 66–77.

[19] D. F. García et al. "TPC-W E-Commerce Benchmark Evaluation". In: *Computer* 36.2 (Feb. 2003), pp. 42–48. ISSN: 0018-9162. DOI: `10.1109/MC.2003.1178045`.

[20] R. Ghosh et al. "Biting Off Safely More Than You Can Chew: Predictive Analytics for Resource Over-Commit in IaaS Cloud". In: *CLOUD*. 2012.

[21] D. Gmach et al. "Selling T-shirts and Time Shares in the Cloud". In: *CCGrid*. 2012.

[22] I. Goiri et al. "Multifaceted resource management for dealing with heterogeneous workloads in virtualized data centers". In: *GRID*. 2010.

[23] Z. Gong et al. "PRESS: PRedictive Elastic ReSource Scaling for cloud systems". In: *CNSM*. 2010.

[24] J. Hamilton. "On designing and deploying internet-scale services". In: *Proceedings of the 21st conference on Large Installation System Administration Conference*. LISA'07. USENIX Association, 2007, 18:1–18:12. ISBN: 978-1-59327-152-7.

[25] J. Hungershofer. "On the Combined Scheduling of Malleable and Rigid Jobs". In: *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 206–213. ISBN: 0-7695-2240-8. DOI: `10.1109/sbac-pad.2004.27`.

[26] I. Hwang et al. "Portfolio Theory-Based Resource Assignment in a Cloud Computing System". In: *CLOUD*. 2012.

[27] C. Klein et al. "An RMS for Non-predictably Evolving Applications". In: *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. IEEE, 2011. ISBN: 978-1-4577-1355-2. DOI: `10.1109/CLUSTER.2011.56`.

[28] J. A. Konstan et al. "Recommended to you". In: *IEEE Spectum* (Oct. 2012).

[29] M. Lee et al. "Supporting soft real-time tasks in the Xen hypervisor". In: *VEE*. 2010.

[30] W. Levine. *The Control handbook*. CRC Press, 1996.

[31] *libvirt: The virtualization API*. URL: `http://libvirt.org/`.

[32] M. Maggio et al. "A Game-Theoretic Resource Manager for RT Applications". In: *ECRTS*. 2013.

[33] J. Mars et al. "Bubble-Up: increasing utilization in modern warehouse scale computers via sensible colocations". In: *MICRO*. 2011.

[34] P. Marshall et al. "Improving Utilization of Infrastructure Clouds". In: *CCGrid*. 2011.

[35] X. Meng et al. "Efficient resource provisioning in compute clouds via VM multiplexing". In: *ICAC*. 2010.

[36] A. Merchant et al. "Maestro: quality-of-service in large disk arrays". In: *ICAC*. 2011.

[37] F. F.-H. Nah. "A study on tolerable waiting time: how long are Web users willing to wait?" In: *Behaviour and Information Technology* 23.3 (2004), pp. 153–163.

[38] R. Nathuji et al. "Q-clouds: managing performance interference effects for QoS-aware clouds". In: *EuroSys*. 2010.

[39] C. Reiss et al. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *SOCC*. 2012.

[40] *Rice University Bidding System*. URL: `http://rubis.ow2.org`.

[41] U. Sharma et al. "A Cost-Aware Elasticity Provisioning System for the Cloud". In: *ICDCS*. 2011.

[42] U. Sharma et al. "Provisioning multi-tier cloud applications using statistical bounds on sojourn time". In: *ICAC*. 2012.

[43] Z. Shen et al. "CloudScale: elastic resource scaling for multi-tenant cloud systems". In: *SOCC*. 2011.

[44] S. Srinivasan et al. "Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs". In: *Proceedings of the 9th International Conference on High Performance Computing*. HiPC '02. Springer-

Verlag, 2002, pp. 174–183. ISBN: 3-540-00303-7. DOI: 10.1007/3-540-36265-7_17.

[45] C. Stewart et al. “Performance modeling and system management for multi-component online services”. In: *NSDI*. 2005.

[46] C. Stewart et al. “Exploiting nonstationarity for performance prediction”. In: *EuroSys*. 2007.

[47] R. Sudarsan et al. “ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment”. In: *Proceedings of the 2007 International Conference on Parallel Processing*. ICPP ’07. IEEE Computer Society, 2007. ISBN: 0-7695-2933-X. DOI: 10.1109/ICPP.2007.73.

[48] N. Tolia et al. “Quantifying Interactive User Experience on Thin Clients”. In: *Computer* 39.3 (Mar. 2006), pp. 46–52.

[49] N. Vasić et al. “DejaVu: accelerating resource allocation in virtualized environments”. In: *ASPLOS*. 2012.

[50] A. Verma et al. “ARIA: automatic resource inference and allocation for MapReduce environments”. In: *ICAC*. 2011.

[51] S. Vijayakumar et al. “Automated and dynamic application accuracy management and resource provisioning in a cloud environment”. In: *GRID*. 2010.

[52] L. Wang et al. “Remediating Overload in Over-Subscribed Computing Environments”. In: *CLOUD*. 2012.

[53] M. Woodside et al. “Service System Resource Management Based on a Tracked Layered Performance Model”. In: *ICAC*. 2006.

[54] Q. Zhang et al. “A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications”. In: *ICAC*. 2007.

[55] W. Zheng et al. “JustRunIt: experiment-based management of virtualized data centers”. In: *USENIX Annual Technical Conference*. 2009.

[56] X. Zhu et al. “1000 Islands: Integrated Capacity and Workload Management for the Next Generation Data Center”. In: *ICAC*. 2008.