Performance-Based Service Differentiation in Clouds

Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez and Erik Elmroth

Department of Computing Science
Umeå University, Sweden
{*ewnetu, cristian.klein, francisco, elmroth*}*@cs.umu.se*

***Abstract*—Due to fierce competition, cloud providers need to run their data-centers efficiently. One of the issues is to increase data-center utilization while maintaining applications' performance targets. Achieving high data-center utilization while meeting applications' performance is difficult, as data-center overload may lead to poor performance of hosted services. Service differentiation has been proposed to control which services get degraded. However, current approaches are capacity-based, which are oblivious to the observed performance of each service and cannot divide the available capacity among hosted services so as to minimize overall performance degradation.**

In this paper we propose performance-based service differentiation. In case enough capacity is available, each service is automatically allocated the right amount of capacity that meets its target performance, expressed either as response time or throughput. In case of overload, we propose two service differentiation schemes that dynamically decide which services to degrade and to what extent. We carried out an extensive set of experiments using different services – interactive as well as non-interactive – by varying the workload mixes of each service over time. The results demonstrate that our solution precisely provides guaranteed performance or service differentiation depending on available capacity.

I. Introduction

Cloud computing has radically changed the way services – such as e-commerce websites or online video conversion tools – are provisioned with computing resources – such as CPU, memory and disk. It allows service providers to avoid upfront costs. Indeed, horizontal elasticity allows Virtual Machines (VMs) to be acquired and release on-demand, while vertical elasticity allows to add or remove resources to a VM on-demand. This simplicity for the service provider becomes a complexity for the infrastructure provider, who needs to run its data-centers efficiently.

A problem increasingly faced by data-centers is energy efficiency. Already this sector consumes more than 1.3% of the world's electricity and is among the sectors with the fastest power increase. Even worse, most of this energy is being wasted due to two reasons: lack of energy proportionality – e.g., a Physical Machine (PM) at 10% utilization may consume as much as half of its full power – and low utilization [1].

To reduce costs associated with energy, data-center owners aim to increase utilization by consolidating services. This problem is challenging as increasing utilization too much, combined with the varying workload typically encountered by services, may lead to infrastructure overload. This may lead to performance degradation of services and disgruntled end-users. A popular technique to address this problem is **service differentiation**, which marks some services as more important (e.g., gold) while others as less important (e.g., silver and bronze) and tries to shield the former from degradation. For example, Amazon EC2 [2] offers users micro instances and spot instances to express that the services contained therein are less important. Of course, service providers are given economic incentives to use such instances. Outside industry, this topic has received considerable attention in academia (see Section II).

However, a problem with such approaches is that differentiation is done based on requested (or used) capacity – i.e., the amount of resources allocated (or consumed) to a service – and not on the actual observed performance. The capacity-based approaches may not be able to minimize the total performance degradation since the infrastructure provider has no knowledge about how its decisions affect the performance of the hosted services. For example, let us assume an infrastructure hosts two bronze services with a target response time of 1 second. To prevent overload, the infrastructure decides to reduce both services' capacity allocation by 10%. However, due to differences in service behavior, this may increase one service's response time to 2 seconds, while the other one only suffers an increase to 1.1 seconds. This behavior is undesirable and, instead, what is expected is that capacity should be reduced in a way that the performance degradation for both services is alike. In this case, the capacity of the first service should be reduced less while the capacity of the second one more, so as to increase their response time to 1.4 and 1.3 seconds, hence minimizing the total performance degradation. Such unexpected behaviors are due to capacity-based provisioning and disappear if, instead, provisioning is based on performance.

In this paper we propose an autonomic capacity controller for performance-based service differentiation, that decides the amount of capacity each service gets depending on the operating environment and the workload dynamics. For example, when enough capacity is available, each service is allocated the right amount of capacity to meet its target Service

Level Objective (SLO), the Key Performance Indicator (KPI) which is expressed either as response time or throughput. On the other hand, during overload, differentiation is employed to decided which services to degrade and to what extent.

The challenges for providing performance-based differentiation are: 1) non-linear relationship between service KPIs and capacity [3], [4], [5]; 2) complex and unpredictable workload dynamics of the services [6]; 3) differences in KPIs used by services; and 4) capacity shortage due to changes in services – flash crowds – as well as changes in operating environment – for example hardware failures [7], [8].

Our contributions are three-fold:

1) *Performance-based provisioning.* Services are provided with the exact amount of capacity that meets their performance targets. As a result, capacity over-provisioning or under-provisioning is prevented under non-overload conditions (Section IV-A). Service performance targets can be specified in terms of either *throughput* or *response time*.
2) *Service differentiation schemes.* Two service differentiation schemes are formulated to resolve capacity contention among different services when aggregated capacity demand is higher than the available capacity. The schemes are independent of KPIs and ensure performance-based differentiation by translating the problem into a penalty minimization problem (Section IV-B).
3) *Autonomic capacity controller.* A self-adaptive controller that dynamically adjusts the capacity that should be allocated to each service depending on its workload dynamics and its relative importance (Section IV-C).

We evaluate two performance models (throughput and response time) and service differentiation schemes using different services and workload mixes (Section V). Results show that the performance models and differentiation schemes work as expected allowing the infrastructure provider to improve utilization and reduce total SLO violations.

II. Related Work

Service differentiation has been studied for decades under different disciplines such as processor sharing [9], packet switched networks [10], storage systems [11], [12], and web servers [3], [4]. These approaches can be broadly categorized as either *intra-application* or *inter-application* service differentiation.

Intra-application service differentiation: The focus is grouping customers into different classes and providing differentiated service for each class within an application [10], [3], [4], [11], [12], [13], [14], [15], [16]. This, allows for instance, to maintain SLOs of preferred (higher classes) customers in the presence of overload. These works assume the use of a single KPI and the load carried by each request is similar or can be normalized into relative values easily. Specifically, incoming requests are classified into multiple priority classes before they are mapped to a specific node that would potentially meet their performance targets. Requests can either be dropped or delayed depending on their priority and resource availability. The solutions are specifically designed for a single service with multiple user classes. As a result, these solutions are not directly applicable to allocating resources to multiple services running in separate VMs. Furthermore, they also fall short of dealing, together, with services having different SLOs, such as one service targeting response time and another one throughput.

Inter-application service differentiation: Applications are divided into priority classes and capacities are distributed among applications based on their priority during capacity shortage [9], [6]. Kleinrock [9] studied time-sharing of a PM among processes with different priorities in non-virtualized environments. Padala et al. [6] present a control-theoretic approach that dynamically allocates capacity in order to meet performance targets of different services running on different VMs. These works allocate capacity among services based on weights assigned to each class. However, the weights are assigned statically without considering the performance during runtime. Thus, observed and expected performance may diverge. In contrast, our solution makes decisions based on performance along with weights.

A number of service differentiation schemes were proposed in [17], [18], which consist of re-distribution of resources from lesser priority services to higher priority ones in the context of data streams executed over clouds using throughput as KPI. The mechanism proposed makes use of token bucket for data admission and control and petri nets for modeling data flow.

Numerous techniques have been proposed to circumvent overload in clouds. For example, Beloglazov and Buyya [19] try to detect overloaded PMs and mitigate the problem by migrating some VMs to less loaded PMs. Another approach is to address capacity shortage by not executing optional parts of a service [8]. However, not all services may have such optional features. While these solutions address overloaded for specific PMs, they do not specifically consider the case when the whole data-center is overloaded. Antonescu et al. have studied the case of data-center wide overload [7]. They propose a Service Level Agreement (SLA) framework that supports dynamic capacity allocation and employs forecasting of resource utilization and migration to prevent SLO violations. In the event of a data-center-wide overload, a compensation plan is used to repay for the observed SLO violations. Nonetheless, contrary to our contribution, the case when some services are more important than others is not addressed by any of these works.

Several models have been proposed for interactive services to provide guaranteed service [20], [21] using either response time, throughput or both as performance metrics.

We based our response time model on the work by Chandra et al. [21]. In general, the focus of those works is on how to guarantee performance under normal circumstances, when the data-center has enough capacity, not under overload scenarios. The use of performance-based SLAs instead of capacity-based SLAs has also been explored [22]. However, they do not deal with service differentiation, in case of overload, services are offloaded to external infrastructure providers so as to maximize profit by reducing penalties.

III. Problem Definition

We consider a cloud infrastructure that hosts multiple services, each with unpredictable and variable workload dynamics. Each service has a SLA that stipulates a SLO and a class – e.g., gold, silver, bronze. Specifically, the SLO is a target value for a KPI, expressed either as *average response time* or *average throughput*. Also, to allow each service to maintain some functionality at all times – e.g., allow it to return a "service unavailable" message – SLAs may optionally request a minimum capacity, below which the service should never be provisioned. Finally, to shield the user from unexpectedly high costs due to service malfunctioning or an attack, SLAs may optionally request a maximum capacity, above which the service is never provisioned.

The goal is then to continuously adjust capacity allocations to services, so as to drive performance towards the given targets, as closely as enabled by available capacity and minimum capacity constraints. On one hand, if capacity is sufficient, each service should be allocated just the right amount of capacity to reach its performance target, avoiding both under- and over-provisioning. The capacity potentially left unallocated could be used to admit a new service. On the other hand, during capacity shortage, a generic service differentiation scheme should under-provision some services based on the relative performance deviation and the importance of each service so as to reduce total SLO violation.

IV. Performance differentiation

We first present the performance models for throughout and response time, and the performance-based service differentiation schemes. We also present the software architecture of the system.

A. Performance Models

The purpose of the performance models is to map service KPI to capacity requirements. These models are either used to estimate the capacity required by a service to meet its target KPI, or, if capacity is insufficient, choose which services to degrade and to what extent.

Performance models need to fulfill several constraints. First, due to the heterogeneity of hosted services, the performance models need to be as generic as possible and should require no knowledge of service internals. Second, they should predict average behavior and ignore sporadic noise observed in the past. Such noise may be caused, for example, by variations in retrieving data, some being cached in memory, other needing to be fetched from disk. Third, these performance models should quickly adjust to variations in workload and capacity requirements. For example, an increase in the number of users, or a change in request distribution may need refitting the model parameters.

Ideally, given a KPI value as input, a performance model should return the exact capacity that needs to be allocated to a service to reach that KPI value. However, as this is difficult to achieve, a performance model should at least *drive* capacity allocations to the correct value, i.e., by periodically refitting the model parameters when allocating estimated capacity, the application should eventually reach the desired performance. In what follows, we present two such performance models, one for throughput and one for response time. As highlighted below, these two KPIs behave largely differently, hence unifying them into a single model would not be useful.

1) Throughput model: For non-interactive services a commonly used KPI is throughput, i.e., the amount of work completed per time interval or amount of work that the service should complete per time interval. Examples of throughput include frames-per-second for video encoding, transactions-per-second for databases and jobs-per-second for problem-solving applications. In such cases, the average throughput T_i of each service i and the capacity c_i allocated to it can be modeled using a linear relation:

$$T_i = \alpha_i c_i, \tag{1}$$

where α_i is a model parameter, representing the fractions of work completed per second per core that service i can serve. The parameter α_i can be estimated online from past measurements of average throughput and capacity, thus compensating dynamically for small non-linearities in the real system. However, to reduce the impact of measurement noise, we decided to use a Recursive Least Square (RLS) filter [23]. In essence, such a filter takes past estimation of α_i and the current ratio T_i/c_i to output a new value that minimizes the least-squares error. A *forgetting factor* allows to trade the influence of old values for up-to-date measurements. In our experiments, we use a forgetting factor of 0.2.

2) Response time model: End-users of interactive services are sensitive to response time. Indeed, several studies show that increased response time reduces revenue. In particular, end-users abandon the service if response time is above 4 seconds [24] and are likely to join the competition, thus incurring long-term revenue loss. Therefore, it is desirable to maintain target response time for a service. However, in contrast to throughput, response time is monotonically decreasing with capacity in a non-linear fashion [21].

Inspired by [21], we model the relation between average response time R_i and capacity c_i for service i as follows. The relation between average response time R_i and capacity c_i for a service i, from Little's Law is given as:

$$q_i = \lambda_i \cdot R_i, \tag{2}$$

where q_i is the average queue length, i.e., the number of requests that entered the service but have yet to exit, and λ_i is the arrival rate. Next, we use the formula for average response time given by the M/M/1 queuing model:

$$R_i = \frac{1}{\mu_i - \lambda_i}, \tag{3}$$

where μ_i is the average service rate of service i. To model the inverse relationship between capacity and response time, we model $\mu_i = c_i/\gamma_i$, where γ_i is a model parameter. By replacing λ_i from Eq. (2) in Eq (3) and resolving R_i, one obtains the formula for the average response time:

$$R_i = \frac{\gamma_i(q_i + 1)}{c_i} \tag{4}$$

As with throughput, the parameter model γ_i can be estimated using past measurements of capacity, average response time and average queue length using Equation (4). As before, to reduce the influence of measurement noise, we use an RLS filter with forgetting factor 0.2.

B. Service Differentiation Schemes

As long as there is sufficient capacity, the performance models presented in Section IV-A can readily be used to compute the capacity required for each service. However, in case of overload, when the sum of capacities required by each service exceeds infrastructure capacity, there needs to be a mechanism to decide which services to degrade and by how much. Differentiation is achieved by assigning each service a class (e.g., gold, silver, bronze) as stipulated in the SLA.

We propose two schemes for service differentiation: strict and non-strict. With strict differentiation, services in lower classes are degraded (down to a minimum capacity) before degrading services in higher classes. With non-strict differentiation, all services are degraded simultaneously, however, the lower the class, the higher the degradation. It is up to the infrastructure provider to choose among the two differentiation schemes, a public provider based on the adopted business model, while a private provider based on internal policies.

For both schemes, the differentiation decisions are based on the *penalty* p_i, which measures the relative deviation of the actual performance of service i from its target SLA and is defined as follows:

$$p_i = \begin{cases} T_i/\hat{T}_i - 1 & \text{for SLA throughput} \\ \hat{R}_i/R_i - 1 & \text{for SLA response time} \end{cases}, \tag{5}$$

where R_i and $\hat{R}_i$ are measured and target response time, respectively, while T_i and $\hat{T}_i$ are measured and target throughput, respectively.

The penalties formulated this way have several advantages. First, they abstract all KPIs into a single quantity that is positive if the service is over-provisioned, negative if the service is under-provisioned and zero if the service has just the right amount of capacity to reach its target performance. Second, the penalty is linearly dependent on the capacity allocated to the service, as can be easily proven by replacing Eqs. (1) and (4) into Eq. (5). This means that the differentiation schemes have the potential to be more stable due to the linearity of the system.

To provide services a mean to achieve a minimum performance, the SLA of each service i may optionally stipulate a minimum capacity c_i^{min} below which the service should never be degraded. If accepting a new service would make the sum of minimum capacities larger than the total capacity C of the cloud infrastructure, then the new service is simply rejected. Analogously, to protect the service from overconsumption due to a malfunction or denial-of-service attack, a maximum capacity c_i^{max} may optionally be stipulated.

Let us now describe the two schemes in more detail.

1) Non-Strict Scheme: In this scheme, each class is assigned a weight representing the importance of the class, e.g., how high is the compensation paid by the infrastructure provider in case of violation. This scheme can be formulated as a minimization problem as follows:

$$\begin{aligned} \text{minimize} \ & \textstyle\sum_i w_i|p_i| && (6) \\ \text{subject to} \ & \textstyle\sum_i c_i \le C && (7) \\ & c_i \ge c_i^{min} && (8) \\ & c_i \le c_i^{max}, && (9) \end{aligned}$$

where w_i is the weight assigned to the class of service i. Eq. (6) is the weighted sum of penalties that needs to be minimized, Eq. (7) ensures that the total allocated capacity does not exceed available capacity, while Eq. (8) ensures that each service gets the requested minimum capacity. Moreover, Eq. (9) ensures that each service does not go beyond budget.

Note that, even services in the highest class are not guaranteed to reach the target performance in case of overload. For guaranteeing that higher-class services are never degraded before lower-class services are at minimum capacity, we provide a second differentiation scheme.

2) Strict Scheme: The aim of the strict scheme is to give priority to higher-class services and to ensure that they get all capacity needed, as long as lower-class services are at least allocated the minimum requested capacity.

The scheme works as follows. First, each service i is allocated its minimum capacity c_i^{min}. Next, the remaining capacity $C - \sum c_i^{min}$ is distributed from highest to lowest class. If the remaining capacity is not sufficient to satisfy

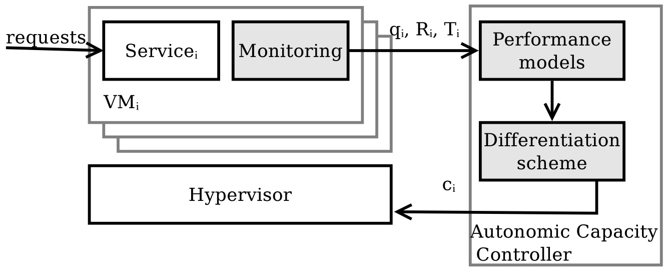


Figure 1. Architecture of the autonomic capacity controller on a single physical machine. Modules in gray are part of our contribution.

the requirements of all the services in a given class, then capacity is allocated so as to minimize the sum of penalties within that class, similar to the optimization problem in the non-strict scheme.

C. *Architecture*

Fig. 1 shows the general architecture of the *autonomic capacity controller* when implemented on a single PM. It loosely follows a Monitor, Analysis, Planning and Execution (MAPE) loop. Monitoring retrieves information from the hosted services about their observed response time, queue length and throughput. During analysis, the capacity required by each service to fulfill its performance target is computed using the performance models in Section IV-A. Past measurements are used to fit model parameters, which are similar to the knowledge component of autonomic controllers.

If total required capacity is insufficient, one of the differentiation schemes in Section IV-B is used to compute the capacities to allocate to each service. As said before, the choice of which scheme to employ (strict or non-strict) is chosen by the administrator. In case enough capacity is available, the differentiation scheme acts as a no-operation, leaving capacities computed by the performance models unchanged. Finally, during the planning and execution phase, the hypervisor is configured to enforce the computed capacities.

The MAPE loop is invoked periodically, in our experiments every 10 seconds. We chose this value as it is short enough to make the system reactive and long enough to observe the effects of the new capacity allocation on the performance of the services [6].

V. Evaluation

In this section we first describe the experimental setup and then present the results of evaluation of the performance models and service differentiation schemes using throughput, response time and mixed targets.

A. *Setup*

Experiments were conducted on a single PM equipped with a total of 32 cores[1] and 56 GB of memory. To emulate a typical cloud environment and allow to easily enforce capacity allocations, we used the Xen hypervisor [25]. Each service was deployed with all its components – e.g., video conversion tool, web server, database server – inside its own VM, as is commonly done in practice [26], e.g., using a LAMP stack [27]. Since we are primarily interested in evaluating CPU allocation strategies, we configured each VM with 6 GB of memory, enough to avoid disk activity. To allow experimenting with a diverse load-mix, as can be found in a cloud data-center [28], we used two types of services: interactive and non-interactive.

1) Interactive services: This type of services only perform computations as a result of a user request. We used the following 4 services: RUBiS [29], RUBBoS [30], Olio [31] and a math service. The first three are widely-used cloud benchmarks (see [32], [33], [34], [35], [36], [37], [38]) and represent an eBay-like e-commerce service, a Slashdot-like bulleting board and an Amazon-like book store, respectively. With the math service we stress the floating-point unit of the CPU, by computing the factorial of a number requested by the user.

To emulate the users accessing interactive services, we used our custom `httpmon` workload generator[2], which supports both closed and open client models [39]. The closed system model is characterized by two parameters: number of users and think-time. Each user sends a request, waits for a reply, then waits an exponentially-random think-time and repeats. The average request inter-arrival time is the sum of the average think-time and average response time of the service (multiplied by the number of users). As a result, the users are "slowed down" by a service close to overload. In contrast, in an open system model, the generation of new requests does not depend on the completion of previous ones. We used this model when measuring response time, as the tested services are more sensitive to under-provisioning [39], hence, bad decisions taken by our system are better highlighted.

2) Non-interactive services: Cloud data-centers are also used to run services in the background. For non-interactive services we used two applications: Sudoku Solver[3] and video encoding. The Sudoku Solver continuously solves random sudokus and **throughput** is measured in terms of the number of sudokus solved per second (sps). `Mencoder` was used to convert several video files, using a diverse set of video encoding parameters. In this case, **throughput** is defined as the number of frames per second (fps).

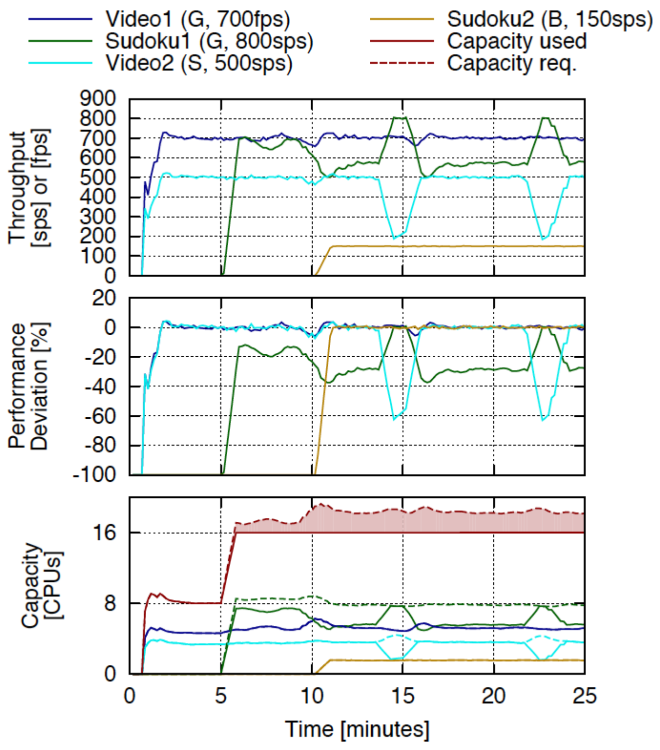
B. *Results*

To incrementally test the performance models and differentiation schemes, we focus on three cases: When throughput is the only KPI, when response time is the only KPI and when both KPIs are employed at the same time. For

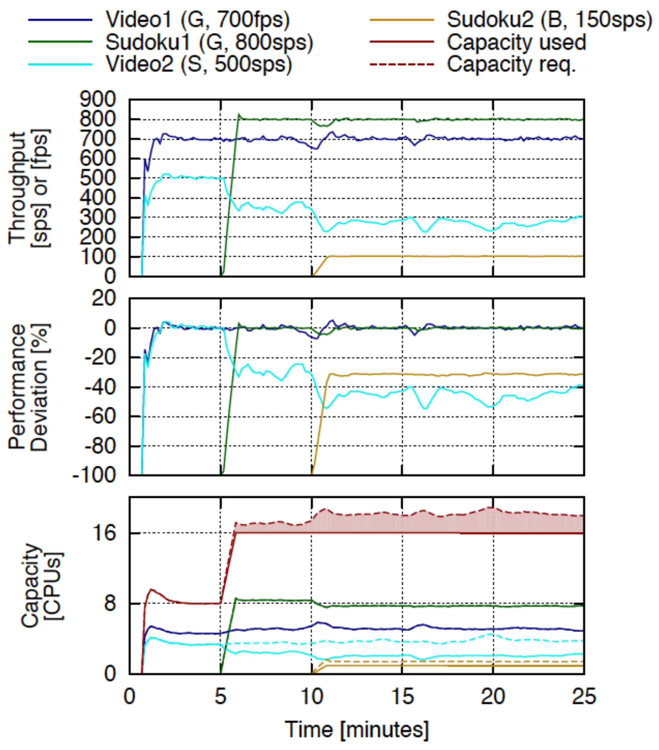
[1]Two AMD Opteron^{TM} 6272 processors, 2100 MHz, 16 cores each, no hyper-threading.

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

[3]http://norvig.com/sudoku.html

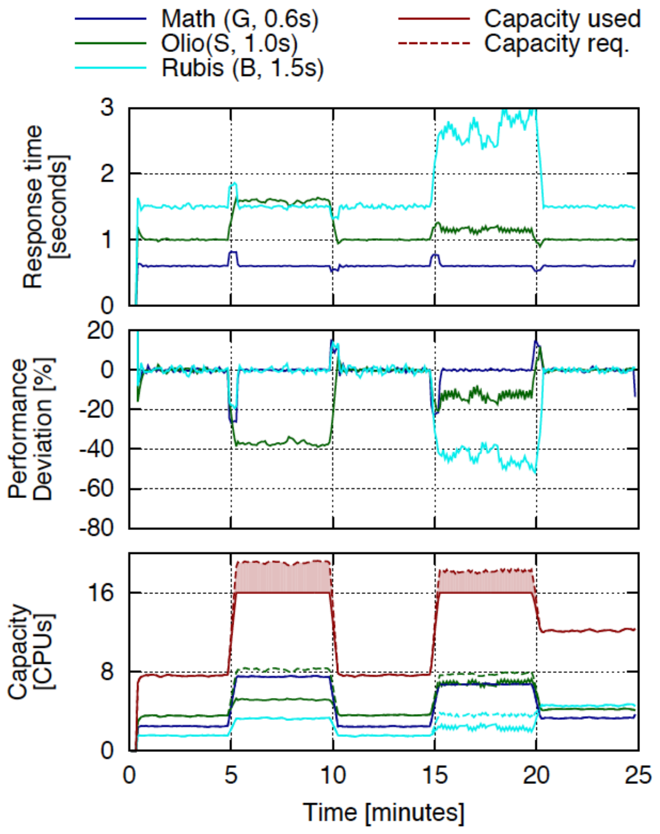


(a) Non-strict.

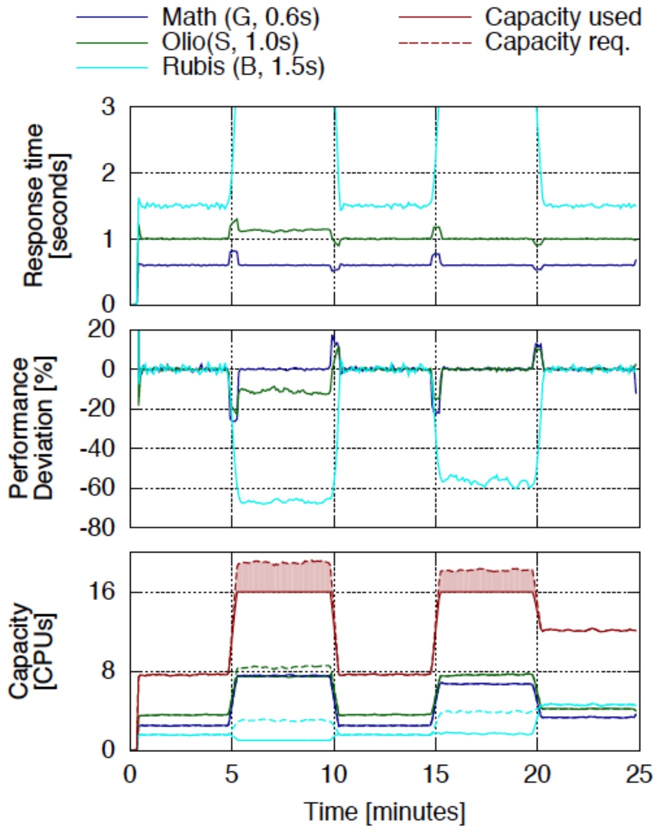


(b) Strict.

Figure 2. Throughput for four services: two gold, one silver and one bronze. Infrastructure capacity is 16 CPUs.



(a) Non-strict.



(b) Strict.

Figure 3. Response time for one service per class. Infrastructure capacity is 16 CPUs.

interactive services, we injected a variable load in order to vary capacity requirements, so as to test the system both when capacity is sufficient and when it is insufficient. We configured the system with three classes—gold, silver and bronze—with weights 4, 2 and 1 as well as a minimum capacity (CPUs) of 2, 1.5 and 1, respectively. We chose not to set maximum capacity for any service to better highlight the behavior of the differentiation schemes in case of overload.

The plots in this sections are structured as follows. Each figure shows the results of a single experiment. The x-axis represents the time elapsed since the start of the experiment. The time is divided into 5 equal intervals marked with vertical dotted lines. At the start of each interval, an environmental factor is changed, either a new service enters the system or the number of end-users changes, as described for each experiment. The top graph of each figure plots the measured KPI value for each service. The middle graph plots the relative performance deviation from the target, as computed using (5). The bottom graph plots the required capacity in dashed, as computed by the performance module, and the allocated capacity in solid, as computed by the differentiation module. To show overload situations, we also plot the total allocated and requested capacity, and highlight the missing capacity by filling the area between the two. The legend displays the name of the service and its SLA in parenthesis: the class (**G**old, **S**ilver or **B**ronze) and the SLO (i.e., the target KPI).

1) Throughput targets: Fig. 2 shows the two service differentiation schemes when the KPI is throughput. The two video services were started at the beginning while Sudoku1 and Sudoku2 services were started later at minutes 5 and 10, respectively. The target throughputs were met for both video services for the first 5 minutes, since the aggregate demand in capacity was less than the capacity available in the infrastructure. This shows that the throughput performance model provides accurate capacity estimations. However, the targets were not met for some of the services after 5 minutes due to capacity shortage.

Under the non-strict scheme (Fig. 2(a)), Video1 (gold) and Sudoku2 (bronze) services were not affected while Video2 (silver) service was little affected. However, Sudoku1 (gold) service was performing below its target in almost all intervals. This is because for a small performance improvement, Sudoku1 service required a very large capacity increment, whereas for a small capacity increment, the bronze (Sudoku2) and the silver (Video2) services showed a large performance improvement. Since the non-strict scheme is based on total penalty minimization, the controller allocated capacity to the silver and bronze services instead of satisfying the demand for the sudoku gold service, i.e., Sudoku1.

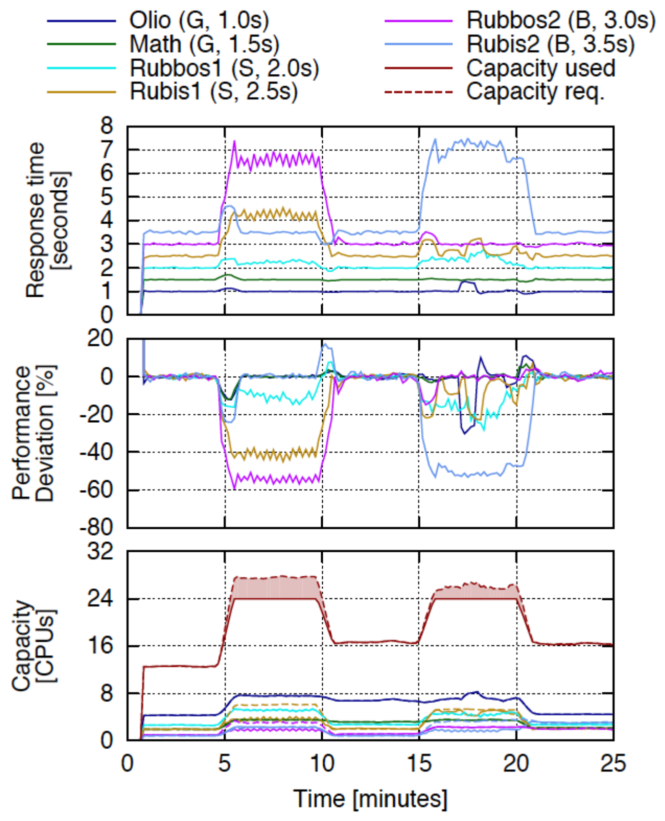
On the other hand, under the strict scheme (Fig. 2(b)), both the silver and the bronze services were affected while the targets of the two gold services were met. The reason is that, in this scheme, the demands of higher priority services are met first before lower priority services. As a result, after having allocated the required capacity to the two gold services, the remaining capacity was insufficient to meet the requirements of both the silver and bronze services. Therefore, the bronze service was degraded to minimum capacity, while silver was given the rest. While this was insufficient to meet silver's target, it was operating above its minimum capacity level.

Note that, a small spikes in performance which are above or below the target can be observed for both the Video as well as Sudoku service, e.g., around minute 10 in Fig. 2(b). This was due to model errors caused by sudden changes in the workload behavior: However, in all cases, the performance model quickly adapted the performance to the target values. Besides, we observe a reverse in performance for Sudoku1 and Video2 around minutes 14 and 22 in Fig. 2(a). This was due to the workload behavior at these particular moments. By coincidence the videos encoded at these two different moments were very compute intensive compared to the other videos while the sudokus solved were relatively easy. Thus, the optimizer favored the service which gave a higher performance improvement for a smaller capacity increment, which was Sudoku1 during that time interval.

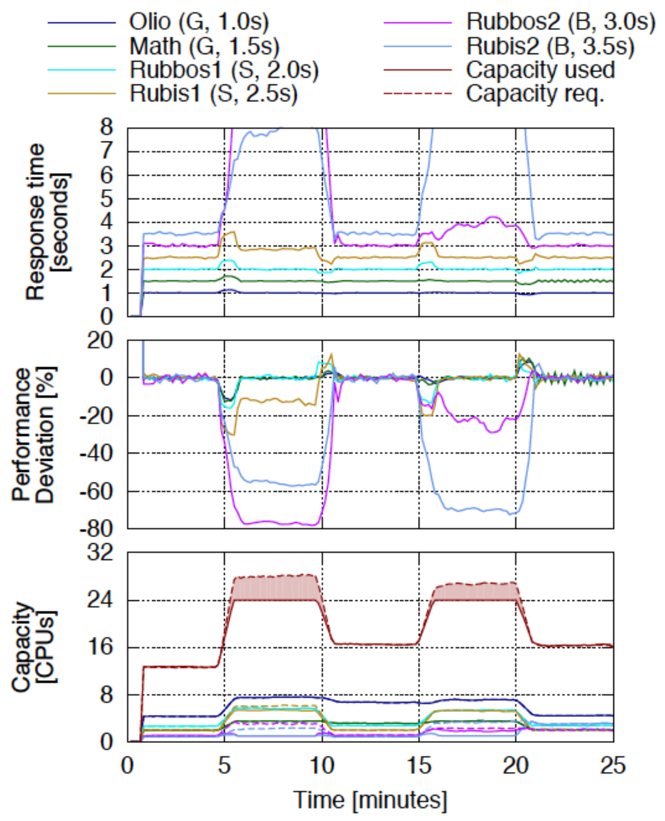
2) Response time targets: Fig. 3 shows the result for response time when a single service was deployed per class. The target response time was met for the three services during minutes 0–5, 10–15 and 20–25, in both the strict and non-strict schemes. This confirms that, as long as aggregated capacity demand is below the infrastructure capacity, the response time model correctly estimates the right capacity to be allocated to each service.

During the other time intervals, infrastructure capacity is insufficient. With the non-strict scheme, the response times for both silver and bronze service were above their targets during minutes 5–10 and 15–20 due to insufficient capacity. In particular, during minutes 5–10, the silver service was highly affected while the bronze service was not affected at all. The reason is that the non-strict differentiation scheme favors services with higher performance improvement per capacity increment, which in this case happens to be the bronze service. With the strict scheme, the gold service was not affected at all while the silver service was slightly affected only during minutes 5–10. In contrast, the bronze service was highly degraded, receiving only minimum capacity during minutes 5–10 and minutes 15–20.

Fig. 4 shows the response time when two services were deployed per class. The performance targets were met for gold services, both using the strict and non-strict schemes. Capacity shortage occurred during intervals 5–10 and 15–20. Using the non-strict scheme, the performance target of one of the silver service, Rubbos1, was slightly degraded during the two intervals. A high performance degradation



(a) Non-strict.



(b) Strict.

Figure 4. Response time for two services per class. Infrastructure capacity is 24 CPUs.

was observed for the Rubis1 silver service and the Rubbos2 bronze service, while no degradation was observed for the Rubis2 bronze service during the second interval (minutes 5–10). Noteworthy is that the performance deviation for Rubis1 was worse than for Rubis2, despite Rubis1 being of a higher class, since this is what drives the sum of penalties to a lower value. In the fourth interval (15–20), a higher performance degradation was observed only for Rubis2 bronze service.

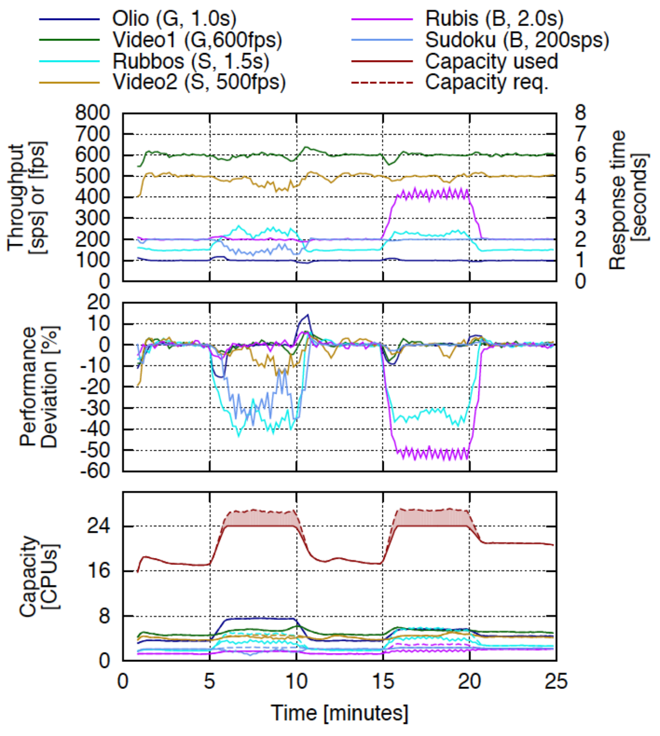
On the other hand, under the strict scheme, the performances of the silver services were improved while the bronze services were degraded. While the capacity was insufficient to meet Rubis1's target during the first interval, its performance was improved under the strict scheme compared to the non-strict scheme. That is because, under strict scheme, capacity is allocated per class from high to low priority and the penalty optimization was performed inside the silver class, as capacity was not sufficient to satisfy the requirements of both silver services. The bronze services were degraded to operate at minimum capacity.

Note that the small spikes observed after introducing changes in workloads are due to the fact that the model requires a small adaptation period (a maximum of 40 seconds) [40] to converge to the target response time. At any rate, the results confirm the system behaves as designed using both differentiation schemes when using response time as a KPI.

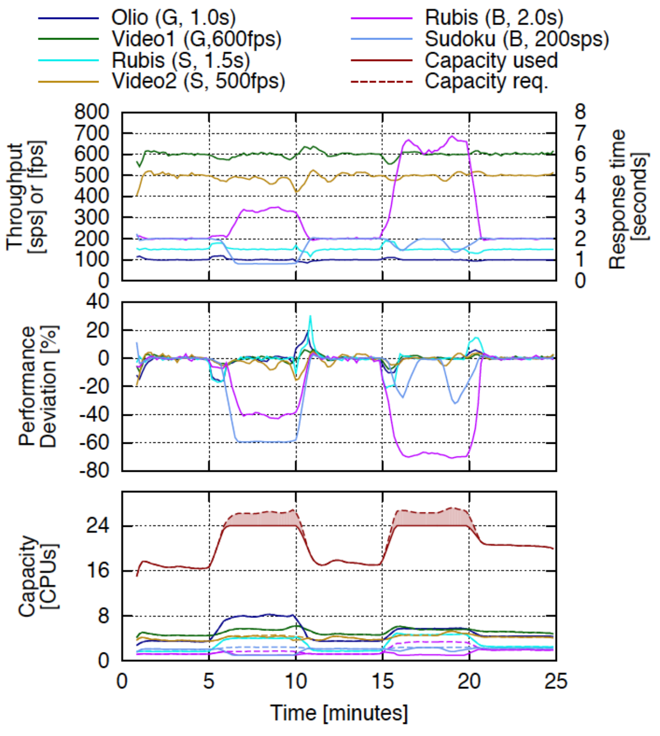
3) Mixed targets: Let us now show how the system behaves when some services use throughput and others use response time as KPI.Fig. 5 shows the results with 6 services. Capacity shortage occurred during the intervals 5–10 and 15–20. The targets for the two gold services were met under both the strict and non-strict schemes. As for the other services, using the non-strict scheme, during the first interval, minutes 5–10, one of the silver services was slightly degraded (Video2) while the other silver service (Rubbos) and one of the bronze service (Sudoku) were highly affected. On the other hand, during the second interval, minutes 15–20, only Rubbos and Rubis were degraded. In contrast, using the strict scheme, only the bronze services were degraded.

C. Discussion

The experiments highlighted that both service differentiation schemes work as designed. When capacity is sufficient, the target performance of all services is reached, no matter the KPI, thanks to the accurate estimations of the performance models. During capacity shortage, service differentiation took place to degrade one or more services. The experiments also highlighted the trade-off between the strict and the non-strict scheme. When the strict scheme was used, the performance of higher classes was given priority over the performance of lower classes. In fact, as long as minimum requested capacity could be allocated to lower



(a) Non-strict.



(b) Strict.

Figure 5. Mixed targets for two services per class. Infrastructure capacity is 24 CPUs.

classes, higher classes were not impacted at all. When the non-strict scheme was employed, the performance of all classes was degraded in case of capacity shortage, however, higher classes were proportionally degraded less than lower classes.

To sum up, our contribution provides service differentiation to services hosted in a cloud infrastructure. It is up to the infrastructure provider, based on its business plan, to choose among the two differentiation schemes. The focus of this work was to show the behavior of the system under both schemes. Comparison of which scheme is better is left for future work as it would require finding a profit model for the services and a pricing model for the infrastructure.

VI. Conclusions and Future Work

We introduced performance models for throughput and response time that map performance to capacity. The models are used to provision capacity in order to maintain performance around requested targets. We also introduced two service differentiation schemes, strict and non-strict, to manage infrastructure overload events. The former scheme favors higher-class services, whereas the latter one proportionally degrades all services, but lower-classes are degraded more than higher-class services.

We carried out an extensive set of experiments using different services – interactive as well as non-interactive – by varying the workload mixes of each service over time. The results demonstrate that our performance-based controller precisely provides guaranteed performance or service differentiation depending on available capacity.

Our contribution enables infrastructure providers, whether private or public, to run their infrastructure closer to peak capacity, thanks to a better control on how services are degraded in case of overload. This enables them to utilize their data-center efficiently while maintaining performance guarantees for critical services. Our performance-based controller spreads the capacity shortage so as to minimize overall performance degradation, also taking into account the importance of each service.

Future works include investigating tail values instead of average values for response time, extending the performance models for cases when more than one type of resource are combined (e.g. CPU and memory) as well as differentiation schemes for services spanning multiple PMs. To encourage further research and make our results reproducible, we released all source code[4].

Acknowledgement

This work was partially supported by the Swedish Research Council (VR) project *Cloud Control*, the Swedish Government's strategic effort *eSSENCE* and European Union's Seventh Framework Programme under grant agreement 610711 (CACTOS).

[4]https://github.com/ewnetu/cloud-diff

References

[1] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[2] A. W. Services. Amazon ec2 spot instances. Available online: http://aws.amazon.com/ec2/purchasing-options/spot-instances/.

[3] M. Andreolini *et al.*, "A cluster-based web system providing differentiated and guaranteed services," *Cluster Computing*, vol. 7, no. 1, pp. 7–19, Jan. 2004.

[4] Y.-D. Lin *et al.*, "Multiple-resource request scheduling for differentiated QoS at website gateway," *Comput. Commun.*, vol. 31, no. 10, Jun. 2008.

[5] Y. Wei *et al.*, "Session based differentiated quality of service admission control for web servers," in *Computer Networks and Mobile Computing (ICCNMC)*, 2003.

[6] P. Padala *et al.*, "Automated control of multiple virtualized resources," in *European Conference on Computer Systems (EuroSys)*. ACM, 2009.

[7] A.-F. Antonescu *et al.*, "Dynamic SLA management with forecasting using multi-objective optimization." in *Integrated Network Management (IM)*. IEEE, 2013, pp. 457–463.

[8] C. Klein *et al.*, "Brownout: Building more robust cloud applications," in *ICSE*, 2014.

[9] L. Kleinrock, "Time-shared systems: a theoretical treatment," *J. ACM*, vol. 14, no. 2, pp. 242–261, Apr. 1967.

[10] N. Saxena *et al.*, "A new service classification strategy in hybrid scheduling to support differentiated QoS in wireless data networks," in *ICPP*, 2005, pp. 389–396.

[11] V. Sundaram *et al.*, "A practical learning-based approach for dynamic storage bandwidth allocation." in *IWQoS*, ser. LNCS, vol. 2707. Springer, 2003, pp. 479–497.

[12] M. Mesnier *et al.*, "Differentiated storage services," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 57–70.

[13] N. Bhatti and R. Friedrich, "Web server support for tiered services," *Network, IEEE*, vol. 13, no. 5, pp. 64–71, Sep 1999.

[14] T. Voigt, "Kernel mechanisms for service differentiation in overloaded web servers," in *Usenix Annual Technical Conference*, 2001, pp. 189–202.

[15] X. Chen *et al.*, "Aces: An efficient admission control scheme for qos-aware web servers," *Comput. Commun.*, vol. 26, no. 14, pp. 1581–1593, Sep. 2003.

[16] H. Zhu *et al.*, "Demand-driven service differentiation in cluster-based network servers," in *INFOCOM*, vol. 2, 2001, pp. 679–688.

[17] R. Tolosana-Calasanz *et al.*, "Revenue-based resource management on shared clouds for heterogenous bursty data streams," in *Economics of Grids, Clouds, Systems, and Services - 9th International Conference, GECON.*, 2012, pp. 61–75.

[18] J. Á. B. others, "Revenue creation for rate adaptive stream management in multi-tenancy environments," in *Economics of Grids, Clouds, Systems, and Services - 10th International Conference, GECON. Proceedings*, 2013, pp. 122–137.

[19] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 7, Jul. 2013.

[20] M. Bjrkqvist *et al.*, "QoS-aware service VM provisioning in clouds: Experiences, models, and cost analysis." in *ICSOC*, ser. LNCS, vol. 8274. Springer, 2013, pp. 69–83.

[21] A. Chandra *et al.*, "Dynamic resource allocation for shared data centers using online measurements," in *IWQoS*. Springer, 2003, pp. 381–398.

[22] D. Dib *et al.*, "SLA-Based Profit Optimization in Cloud Bursting PaaS," in *CCGrid*, May 2014, pp. 141–150.

[23] W. Liu, J. C. Principe, and S. Haykin, *Kernel Adaptive Filtering: A Comprehensive Introduction*, 1st ed. Wiley Publishing, 2010.

[24] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour and Information Technology*, vol. 23, no. 3, 2004.

[25] P. Barham *et al.*, "Xen and the art of virtualization," in *SOSP*. ACM, 2003.

[26] K. Sripanidkulchai *et al.*, "Are clouds ready for large distributed applications?" *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, 2010.

[27] (2013) Tutorial: Installing a LAMP web server. Available online: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html.

[28] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *SoCC*. ACM, 2012.

[29] (2014) Rice university bidding system. Available online: http://rubis.ow2.org.

[30] (2014) Rubbos. Available online: http://jmob.ow2.org/rubbos.html.

[31] (2014) Olio. Available online: http://incubator.apache.org/projects/olio.html.

[32] Z. Gong *et al.*, "PRESS: Predictive elastic resource scaling for cloud systems," in *CNSM*. IEEE, 2010.

[33] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "CloudScale: elastic resource scaling for multi-tenant cloud systems," in *SoCC*. ACM, 2011.

[34] W. Zheng *et al.*, "JustRunIt: experiment-based management of virtualized data centers," in *ATC*. USENIX, 2009, pp. 18–28.

[35] C. Stewart and K. Shen, "Performance modeling and system management for multi-component online services," in *NSDI*. USENIX, 2005, pp. 71–84.

[36] N. Vasić *et al.*, "DejaVu: accelerating resource allocation in virtualized environments," in *ASPLOS*. ACM, 2012.

[37] C. Stewart *et al.*, "Exploiting nonstationarity for performance prediction," in *EuroSys*. ACM, 2007.

[38] Y. Chen *et al.*, "SLA decomposition: Translating service level objectives to system level thresholds," in *ICAC*. IEEE, 2007.

[39] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: A cautionary tale," in *Networked Systems Design and Implementation (NSDI)*, 2006.

[40] E. B. Lakew *et al.*, "Towards faster response time models for vertical elasticity," in *6th Cloud Control Workshop, part of the Proceedings of the 2014 IEEE Conference on Utility and Cloud Computing (UCC 2014)*, 2014, pp. 560–565.