Resource Management for Service Level Aware Cloud Applications

Cristian Klein,
Francisco Hernández-Rodriguez
Department of Computer Science
Umeå University, Sweden

Martina Maggio,
Karl-Erik Årzén
Department of Automatic Control
Lund University, Sweden

Abstract

Resource allocation in clouds is mostly done assuming hard requirements, time-sensitive 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 mitigate this issue, we introduce service-level awareness in clouds, assuming applications contain some optional code that can be dynamically deactivated as needed. We propose a resource manager that allocates resources to multiple service-level-aware applications in a fair manner. To show the practical applicability, we implemented service-level-aware versions of RUBiS and RUBBoS, two popular cloud benchmarks, together with our resource manager. Experiments show that service-level awareness helps in withstanding flash-crowds or failures, opening up more flexibility in cloud resource management.

1. Introduction

Cloud computing radically changed the management of data-centers [5]. 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. This issue was further aggravated by the growing number of cores per PM, driven by the end of frequency scaling, which increased the amount of unused hardware per node. However, thanks to advances in cloud computing technologies, applications are now wrapped inside Virtual Machines (VMs) and consolidated onto fewer PMs [20].

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 given the available hardware. In this area, there has been a tremendous amount of work, mostly assuming that applications are time-sensitive – lengthy responses may lead to dissatisfied users – and their resource requirements are not flexible – the application is either given the needed amount of resources or fails. Combined with the fact that most cloud applications have dynamic resource requirements [23], 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.

For increased resource management flexibility, 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 offer end-users recommendations of similar products they might be interested in. No doubt, recommender engines greatly increase user experience. However, due to their sophistication, they are highly demanding on computing resources [18]. By selectively activating or deactivating the corresponding code, proportionally to the service-level, resource consumption can be controlled and data-center overload can be avoided 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

This work was partially supported by the Swedish Research Council (VR) under contract number C0590801 for the project Cloud Control and through the LCCC Linnaeus Center. Also, we received partial support from the EU project VISION Cloud under grant agreement no. 257019, the Swedish Government's strategic effort eSSENCE and from the ELLIIT Excellence Center.

times [3]. 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 [20]. 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 [2, 15]. 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, delivering partial content in a timely manner is better than overloading the data-center and rendering hosted applications unresponsive.

In this article we build the necessary software infrastructure to support SL-aware cloud applications. We focus on the resources of a single PM, leaving multiple-PM extensions for future work. We assume that the application developer followed the guidelines to produce SL-aware application presented in Section 2. We propose a Resource Manager (RM) that coordinates the resource allocation among applications competing for the same resources (Section 3). The highlight of our contribution is that the design is backed up by theoretical results from game theory. Our system provides specific guarantees on desirable properties such as convergence and fairness among the applications, which translates to withstanding capacity shortages predictably. We evaluate our approach, in Section 4, using two well-known cloud benchmark applications, RUBiS [24] and RUBBoS [4], that are extended with SL-aware recommender engines. To foster further research and pursue repeatability we have made all source code publicly available[1].

2. Application model

In this section we describe the application model that we expect developers to follow. We assume that every application i is composed of time-sensitive requests, which have to be executed before a soft deadline expires: Exceeding it should be minimized, to avoid user dissatisfaction. As an example, such applications can be made Service-Level (SL)-aware, by marking a part of the request as *optional*. Being able to run optional computations is desirable, as they would improve end-user experience, however, deactivating them is preferred to missing a deadline. Let the probability of executing optional computations between time[2] k and $k+1$ be equal to the SL of the application s_i^k. Consequently, the capacity required by the application is proportional to s_i^k.

Every application is requested to regularly update the Resource Manager (RM) about its performance. More precisely, a **matching value** respecting three properties should be computed. 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 f_i^k as follows:

$$f_i^k = 1 - t_i^k / \bar{t}_i \tag{1}$$

where $\bar{t}_i$ is the desired deadline and t_i^k is the maximum response-time of requests served from $k-1$ to k. The matching value f_i^k 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**. At time j the application i updates its service level according to

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

that can be different for each application. This internal feedback loop belongs to the application and the RM is not informed about its behavior, nor about its execution interval (the distance between j and $j+1$). Examples of how to design such loop can be found in [17]. As a result, both the SL s_i^j 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. Two proposals for update rules are described in [17]. Note that our framework allows application to be non-cooperative, i.e, SL-unaware, as

[1]GitHub repository: https://github.com/cristiklein/cloudish

[2]Throughout, time is assumed discrete and denoted with k or j, while i always represents the application.

most existing applications are. If no matching value is communicated, the RM simply assumes it to be zero.

To clarify the above concepts, we sketch an e-commerce website as an example of an SL-aware application. We consider the visualization of a product's page as one request. The optional code of such a request consists in retrieving recommendations of similar products. For each request, besides retrieving the product information, the application runs the recommender engine with a probability s_i^j. Increasing s_i^j increases the amount of served recommendations, thus increasing end-user experience, but also the capacity requirements of the application. To avoid saturation, the application is made self-adaptive by controlling the parameter s_i^j so as to keep the maximum response-time around a configured deadline.

One of the main differences between this work and similar research in the context of embedded systems [7, 19] 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 g_i.

3. Resource management

The role of the RM is to select the capacity of the Physical Machine (PM) that each application is allowed to use. In many works cited in Section 5, 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, our RM does not directly monitor the resource usage of the applications but uses information on the applications' performance that are conveyed through the matching value defined in Eq. (1)[3] 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. At initialization, the RM sets the capacities to $c_i^0 = 1/n$ where n is the number of applications. Subsequently, at the beginning of each control interval, it first retrieves measurements for all the matching values f_i^k — as defined in Eq. (1) — then updates each capacity according to

$$c_i^{k+1} = c_i^k - \varepsilon_{rm}\left(f_i^k - c_i^k \cdot \sum_p f_p^k\right) \tag{3}$$

where ε_{rm} is a design constant. Given that initially $\sum_i c_i^0 = 1$, one can prove through induction that:

$$\sum_i c_i^k = 1 \tag{4}$$

i.e., the RM enforces that the total allocated capacity does not exceed the available one. 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. The RM's complexity is **linear** with respect to the number of applications, which allows its implementation to have low overhead.

Let us summarize the convergence analysis of the designed system; detailed proof can be found in [7, Section IV]. Using game-theory and treating applications as players bidding for resources, 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. It was also proven 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.

4. Experimental evaluation

Experimental setup. Our testbed is a single PM equipped with two AMD Opteron$^{\text{TM}}$ 6272 processors[4] and 56 GB of memory, which hosts several Virtual Machines (VMs). We used Xen 4.1.2 as a hypervisor and Ubuntu 12.04.2 LTS 64-bits with Linux kernel

[3]As long as the matching value respects the three introduced properties, its formulation can be changed.

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

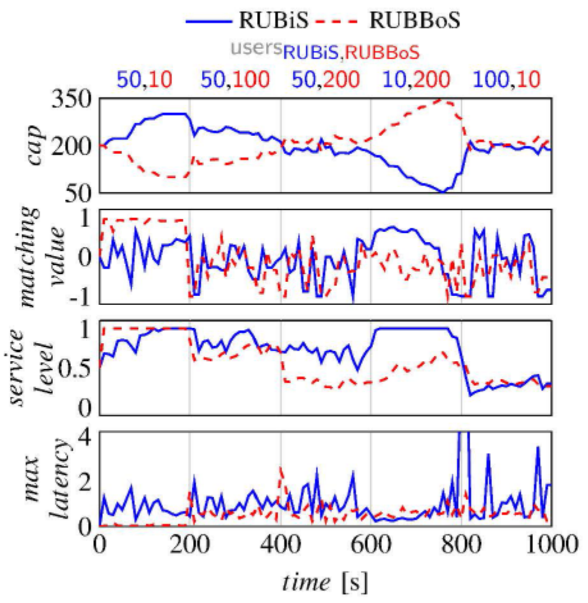

Figure 1: Resource manager and two applications.

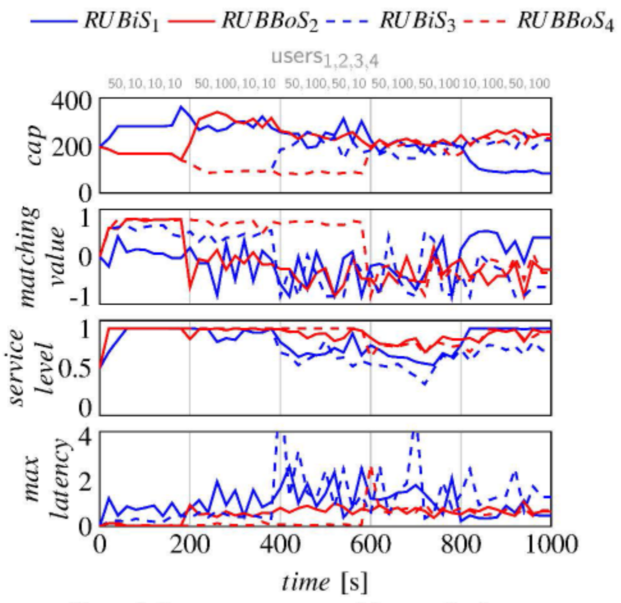

Figure 2: Resource manager and four applications.

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 — 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 among $\mathsf{Dom_U}$, and the RM inside $\mathsf{Dom_0}$. Each application's VM contains the self-adaptive version of the application and all tiers belonging to it — Apache web server, PHP interpreter, MySQL server. Since we focus on CPU allocations, we ensured that the database could be fully cached in memory.

Experimental methodology. To simulate the users' behavior, we dynamically select a think-time and a number of users. Each user runs an infinite loop, which waits for a random time and then issues a request. The random waiting time is chosen from an exponential distribution, whose rate is given by the think-time parameter. 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 $\mathsf{Dom_0}$ 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. The RUBiS and RUBBoS applications are made SL-aware as described in [17], with desired deadlines of 1 and 0.5 seconds, respectively.

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, as described in Eq. (3), based on the matching values they send, computed according to Eq. (1). 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 at time 200, 400, 600 and 800, and observe the behavior of the RM and applications.

Results. Figure 1, displaying the results, is structured as follows. Four metrics are plotted as a function of time for each of the two applications: the cap chosen by the RM, the matching value, the SL and the maximum user-perceived 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, it increases the SL to maximum. Similarly, the adaptive RUBiS 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. RUBBoS reacts to avoid overload and reduces the SL. Furthermore, the RM increases its cap and decreases RUBiS's cap. RUBiS reduces its SL to deal with the new resource allocation. Thus, the system approaches 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 desired deadline of each application (1 second and 0.5 seconds). Similarly, new stationary points are reached after the changes in number of users occurring at 400, 600 and 800.

To further 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. 2, applications that do not run at full SL are assigned equal caps, whose value we call *fair cap*. In other words, despite targeting different desired deadlines 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 settle around 200 or applications 2, 3, 4 in the 5th interval, whose caps settle 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.

Note that in both Figs. 1 and 2, latencies may temporarily increase above the desired deadline. This is expected, since applications continuously try to maximize their service-level, hence, latencies may shortly overshoot. To conclude, we experimentally showed that the RM behaves as theoretically designed, avoiding overload while respecting fairness among applications.

5. Related work

Managing resources in clouds is a challenging task. Resource management schemes are either application or infrastructure-centric. Performing *application-centric* resource allocation (e.g. [6, 8, 26]) 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.

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. [25] proposes Kingfisher, a system that tries to minimize the cloud tenant's deployment cost while reacting to workload changes. However, none of these works take into account the capacity limitations of the cloud provider.

Although some works deal with performance differentiation for multiple classes of clients [21], to our knowledge, the only cloud application that comes close to being SL aware is Harmony [9]. It adjusts 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.

Infrastructure-centric resource allocation strategies like [12, 27] 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) [28]. In [11, 22] the RM is assumed to know the minimum application requirements a priori, which is not a valid assumption in a cloud environment. In [16], application requirements are modeled as random variables and statistical analysis is applied to avoid data-center overload. In [14] 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 [29] 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.

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 [13] — 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 [10].

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.

6. Conclusion

In this paper we discussed a proposal for resource allocation to service-level aware cloud applications. We proposed a game-theoretic resource manager to coordinate the demands of multiple applications in a predictable and fair way. These applications can reduce the burden they inflict on the cloud infrastructure, therefore cooperating to the better management of the available resources, in particular to avoid data-center overload. We implemented the framework and tested it with real-life experiments, demonstrating that we allocate resources fairly to the running applications.

References

[1] D. Ardagna, B. Panicucci, and M. Passacantando. "A game theoretic formulation of the service provisioning problem in cloud systems". In: *WWW*. 2011.

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

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

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

[5] 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).

[6] E. Caron, F. Desprez, and A. Muresan. "Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients". In: *J. Grid Comput.* 9.1 (2011), pp. 49–64.

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

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

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

[10] 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.

[11] T. Cucinotta et al. "On the Integration of Application Level and Resource Level QoS Control for Real-Time Applications". In: *Industrial Informatics, IEEE Transactions on* 6.4 (2010), pp. 479–491.

[12] A. Fedorova, M. Seltzer, and M. D. Smith. "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler". In: *PACT*. 2007.

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

[14] R. Ghosh and V. K. Naik. "Biting Off Safely More Than You Can Chew: Predictive Analytics for Resource Over-Commit in IaaS Cloud". In: *CLOUD*. 2012.

[15] J. Hamilton. "On designing and deploying internet-scale services". In: *LISA*. 2007, pp. 1–12.

[16] I. Hwang and M. Pedram. "Portfolio Theory-Based Resource Assignment in a Cloud Computing System". In: *CLOUD*. 2012.

[17] C. Klein et al. *Introducing Service-level Awareness in the Cloud*. Tech. rep. ISRN LUTFD2/TFRT-7641-SE. Lund University, July 2013.

[18] J. A. Konstan and J. Riedl. "Recommended to you". In: *IEEE Spectum* (Oct. 2012).

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

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

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

[22] R. Rajkumar et al. "A resource allocation model for QoS management". In: *RTSS*. 1997.

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

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

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

[26] U. Sharma, P. Shenoy, and D. F. Towsley. "Provisioning multi-tier cloud applications using statistical bounds on sojourn time". In: *ICAC*. 2012.

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

[28] L. Tomas and J. Tordsson. "Improving Cloud Infrastructure Utilization through Overbooking". In: *CAC*. 2013.

[29] L. Wang, R. Hosn, and C. Tang. "Remediating Overload in Over-Subscribed Computing Environments". In: *CLOUD*. 2012.