ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/268474807

Proposal of cost-effective tenant-based resource allocation model for a SaaS system

Conference Paper · September 2013

DOI: 10.13140/2.1.4143.0726

CITATIONS
7

READS
117

2 authors, including:

Marek Sławomir Woda
Wroclaw University of Science and Technology

47 PUBLICATIONS **107** CITATIONS

SEE PROFILE

All content following this page was uploaded by Marek Sławomir Woda on 19 November 2014.

The user has requested enhancement of the downloaded file.

Proposal of cost-effective tenant-based resource allocation model for a SaaS system

Wojciech STOLARZ, Marek WODA[1]

[1] Instytut Informatyki, Automatyki i Robotyki, Politechnika Wrocławska,
voytec0dh@gmail.com, marek.woda@pwr.wroc.pl

Abstract. Software-as-a-Service (SaaS) is a software distribution paradigm in cloud computing and represents the highest, software layer in the cloud stack. Since most cloud services providers charge for the resource use it is important to create resource efficient applications. One of the ways to achieve that is multi-tenant architecture of SaaS applications. It allows the application for efficient self-managing of the resources. In this paper the influence of tenant-based resource allocation model on cost-effectiveness of SaaS systems is investigated. The tenant-based resource allocation model is one of the methods to tackle under-optimal resource utilization. When compared to traditional resource scaling it can reduce the costs of running SaaS systems in cloud environments. The more tenant-oriented the SaaS systems are the more benefits that model can provide.

1 Introduction

One of recent solutions for over- and underutilization problems may be a tenant-based resource allocation model (TBRAM) for SaaS applications. That solution was introduced and tested with regard to CPU and memory utilization by authors of [8]. They proved the validity of TBRAM by reduction of used server-hours as well as improving the resources utilization. However, the authors deployed their solution into a private cloud, which can only imitate public cloud environment. They tested cases with incremental and peak workload of the system. In this paper it was waned to check whether the TBRAM is really worth to adhere.
Examining that system in public and commercial cloud environment could deliver the answer for that question. Therefore, the main aim of the paper is further TBRAM approach validation, as it was proposed in future research part of the base work [8]. If the results of the research will confirm usefulness of the model then it could be considered as the solution for previous mentioned provisioning problems.

Despite that in the cloud one can automatically receive on-demand resources one can still encounter problems related to inappropriate resource pool at the time. These are over- an underutilization which exists because of not fully elastic pay-per-use model used nowadays [17]. Over provisioning exhibits when, after receiv-

ing additional resources (in reply for peak loads), they are being kept even if they are no longer needed. Thus we are affected from underutilization. Under provisioning (saturation) exhibits when we cannot deliver required level of service because of insufficient performance. This is also known as an overutilization. It leads to the customers' turnover and revenue losses [3]. For example Amazon Elastic Cloud Computing (EC2) service charge users for every partial hour they reserve each EC2 node. Paying for server-hours is common among cloud providers. That is why it is very important to utilize fully given resources in order to really pay just for what we use.

We are still in early stages of cloud computing development. No one shall expect cost effective pay-per-use model for SaaS application after just deploying it in the cloud. What is more, automatic cloud scalability will not work efficiently that way [20]. To achieve desired scalability one need to design a SaaS application with that in mind. In order to do that, the application must be aware how it is used [11]. One can use multitenant architecture to manage the application behavior. It allows using a single instance of the program by many users. It works in similar way like a singleton class in object programming languages, which can supervise creation and life cycle of objects derived from that class. Supporting multiple users is very important design step for SaaS applications [5]. One can distinguish two kinds of multi-tenancy patterns: multiple instances (every tenant has got its own instance running on shared resources) and native multi-tenancy (single instance running on distributed resources) [2, 5]. First pattern scales quite well for small number of users, but if there are more than hundreds it is highly advisable to use the second one.

1.1 Overutilization

The term "point of exhaustion" is often used in relation to overutilization. It is described as a point when some resource if fully utilized, for example 100 % CPU usage or all memory is consumed [16]. This definition tends to be accurate in many simple cases. However, in case of cloud computing that definition seem to be an oversimplification. Authors of [12] propose another definition, according to which the point of exhaustion is a maximal payload that can be assigned to a single virtual machine without decreasing its throughput in the same time. Readings above the exhaustion point describe a saturated machine. This new exhaustion point definition requires measuring a VM throughput together with the resource utilization (CPU or memory). Authorial test SaaS system uses the JMeter tool combined with CloudWatch network related metrics in order to calculate that. The system is stressed with HTTP requests generated by the tool. Then it calculates the throughput of the system by dividing the number of HTTP request by the time from start of the first request to the end of the last one. By measuring it this way we can include all the processing time between the requests as well. In previous works [12, 13, 15, 16, 18, 19] authors focused on throughput to discover inflection

points. Whenever the throughput was dropping while the VM utilization was rising an inflection point was found. In this work the same approach was used.

Authorial SaaS system used above mentioned inflection points to detect an overutilization of given virtual machine. All the VMs with Tomcat are monitored by gathering resource utilization metrics and throughput. The metrics used are as follows: CPU usage and Java virtual machine heap memory consumption. Based on that measures it can be told with good accuracy weather a VM is saturated in given moment or not.

Generally, when the VM is saturated the operating system processes start to use more and more resources making user's processes execution even slower. It has a negative influence on system responsiveness and therefore, on user experience. That is why it is always a good idea to avoid saturation. Even despite it does not have a direct influence on cost (we do not pay extra for high VM usage rate), it can drive to users turnover because of a poor performance.

1.2 Underutilization

From the economical point of view underutilization is just a waste of money. It means that we pay for something we do not need or not even use. From the end user perspective it is hardly noticeable, so from the provider perspective this extra money is spent almost on nothing. More formally, from the definition [4, 7] an underutilization describes a situation when some of the cloud resources are not being used by the working virtual machine. Off course it is almost impossible to assure 100% resource usage all the time so some kind of underutilization is inevitable. Underutilization in a cloud can be measured by the amount of the resources available for use. According to [4] resource is wasted when we can reallocate given resource utilization into another VM without exceeding its maximal quantity allowed. It means that for example: the payload for two VMs could be easily allocated just in one VM making the other VM unused. In order to check if given VM can be allocated to another one we need to calculate combinations of VMs according to some resource. One way to solve that problem is by using the knapsack algorithm. As proposed in the base article [8], the amount of used resources to the knapsack items' weights was assigned. As a value of an item the available quantity of the resource in other VMs was taken. In case of Java heap memory the item's value equals an amount of heap memory that still can be used (available memory). Thus, the most valuable items are the less used ones. The capacity of the knapsack is the amount of available resource of a VM we try to assign the workload to. By using this approach we obtain the maximum number of VM than can be potentially released. That VM number is used to measure underutilization. The lower that number is the better the resources are used.

1.3 Cost

Running the system in a public cloud gives us yet another way to assess the cost effectiveness. Almost every action made in cloud is registered and added to our bill. We pay for sent Internet requests, storage, VM hours and many more. Therefore, the billing statement yields arguably the most accurate estimation of cost-effectiveness. At the end of the day it is the price we need to pay for our cloud service. During the tests the Amazon CloudWatch service was collecting metrics about the cloud environment usage. Both SaaS systems (Base System and TBRAM) are tested against the same test plan, so the requests number is exactly the same. The main difference between them can occur in the number of used virtual machines. That difference should be reflected on the bill statement. The comparison of costs of running the SaaS systems will show if there is any economic improvement with using the TBRAM approach over the traditional resource scaling approach.

2 Related work

Authors in [5] propose profiles approach to scaling in the cloud. They try to use best practices and their knowledge in order to create scalable profiles. The profile contains information that helps to characterize a server in terms of its capabilities. When the scaling activity is fired it takes the profile information into account. In [9] authors propose a toolkit using Java mechanism to support multi-tenancy. They use context elements to track applications running on Java Virtual Machine. That in turn allows distinguishing each tenant. That information can be later used in order to estimate given tenant's resource usage. The tenant context can also be used for billing each tenant's activities. In [6] authors consider an intelligent resource mapping as well as an efficient virtual machines (VM) management. It is a very important problem that greatly influences costs of running applications in a cloud. In [10] authors describe three major components which influence virtual machines performance. These are: measurement, modeling and resource management. They introduce a decomposition model for estimating potential performance loss while consolidating VMs. Amazon proposes its Auto Scaling tool [1] to manage VM instances using predefined or user-defined triggers. It is the Amazon EC2 platform specific mechanism based on resource utilization. In [8] authors implements a tenant-based resource allocation model for their SaaS application deployed in private Eucalyptus cloud. The authors performed tests with incremental and peak workload simulation. In the research they achieved significant reduce of server-hours compared to traditional resource scaling model. The tenant-based model improved also utilization of cloud resources by their SaaS system. Moreover, they introduce formal measures for under and over provisioning of virtual resources. The measures are designed specifically for SaaS applications with respect

to CPU and memory utilization. In this paper cost-effective tenant-based resource allocation model of SaaS system is presented. It will be referred as the base based system.

3 System design

The TBRAM consists of three approaches that leverage multi-tenancy to achieve its goals. The first of them is tenant-based isolation, which separates contexts for different tenants. It was implemented with tenant-based authentication and data persistence as a part of the SaaS platform (Tomcat instances). The second way is to use tenant-based VM allocation. With that approach it was able to calculate the actual number of needed VMs by each tenant in given moment. The last but not least is the tenant-based load balancing (due to paper length it won't be described). It allows distributing virtual machines' load with respect to certain tenant. An overview of the architecture is presented in Figure 1. The dashed line in the picture denotes communication to web services. We can notice that the SCWA element in the Figure was the only change made to the original test bed [10]. That element embraced proposed TBRAM approach.

3.1 Tenant-based isolation

To assure that system worked properly it needed to isolate one tenant form another. A situation when one tenant can access and affect data that do not belong to him/her is unacceptable in any commercial solution. The TBRAM approach proposes low level isolation as it improves its scalability [2]. The tenant-based isolation of TBRAM could be split into two implementations. One was based on data persistence and the other one was based on authentication mechanisms. In this place is worth to mention that tenant based isolation was also used in the Base System. That was because both systems were using the same multi-tenant database. What is more that technique was practically affecting only the SaaS platform, so it is isolated from the SCWA concept. Thanks to that in both systems the SaaS platform was exactly the same, thus minimizing its influence on the results.

In the persistence layer the authors propose Shared Database – Shared Schema as it has the lowest hardware cost and the largest number of tenants per server [14]. To logically separate data the Tenant ID field is used for each database table. From technical point of view JoSQL libraries were used, which let to perform SQL-like queries over Java collections. Those libraries were used by Struts2 interceptors to achieve multitenant preprocessing. Java annotations were used to mark the places in code that needed this kind of tenant-based behavior. That was arguably the most efficient way to implement multi-tenancy since the data were first fetched and then filtered. It could be achieved using SQL selection mechanisms.

Interceptors as an implementation of aspect oriented programming postulates had many advantages as well. The main was that all the code was in one place but could affect any class marked with the annotation. Secondly, that annotation was the only change that needed to be made to an existing application code to enable multi-tenancy. Therefore it could possibly be the most common way to add that tenant layer to existing applications.

Fig. 1 TBRAM system architecture

Tenant-based authentication was the second concept used to achieve tenants' isolation. As proposed by TBRAM it should be implemented into the core application of the system which is SCWA. During the authentication every user was linked to its Tenant ID. From now on the user could access only the data the certain tenant has rights to. Needless to say that one could not access any data before the authentication. The TBRAM also suggest using an Access Control Lists (ACL), which it was decided to omit as it introduces just unneeded complication for me. It was decided to give full access to all SaaS applications to all users for simplicity. It was necessary to receive the tenant information from any point in the SaaS system. The TBRAM proposes a mechanism based on cookies and the servlet context. The authors [8] used a local Tomcat cluster to deploy their solution. In my case the SaaS system was deployed into the Amazon cloud infrastructure and that solution did not worked for me. It was decided not to use Tomcat instances running in cluster mode. That was because an overhead related to sharing session information between all cluster's nodes was a concern. If the nodes are

running in different networks it was thought that it can introduce non negligible influence.

Currently, Tomcat 7 version supports all-to-all session replication. It was used a special web service to serve the tenant information instead. It was more platform-independent and it could work in both cases. Whenever any user was accessing given VM for the first time, the SaaS platform was checking in SCWA if that user was authenticated and authorized to do that. If it was, then its specific data were saved locally in a session context so the next request from that user didn't require further communication to the SCWA's centralized web service. Therefore, only VMs that needed that specific information were acquiring it. There are also other methods of session replication like session persistence (shared file system or database) which are outside the research scope.

According to TBRAM a Tenant Context object was conceptualized. It contained information about tenant ID, active users and their VM assignations etc. A Tenant Context Manager object in turn was used to manage all the underlying Tenant Context objects. Thanks to that information about the tenant's state was available to all other services. The Tenant Context allowed isolating each request sent to the platform based on given user's tenant information. We can see several users from two different tenants (subscribers). Despite they physically share the same SaaS applications and the persistence layer they are still logically isolated by their tenant contexts. These context objects help to achieve native multi-tenancy of the applications. The users have no idea they are sharing the same resources.

3.2 Tenant-based VM allocation

Tenant-based VM allocation was used to determine the number of VM instances needed for given tenant in given moment. It combined the concept of profile approach with monitoring services implemented within the SaaS system. A profile used for the test bed in the base paper was a small virtual machine profile. It was meant to substitute the m1.small EC2 instance in Amazon cloud. The profile was as follows: 1 CPU core, 1 GB of RAM, 800 MB to the JVM heap memory and 100 as the number of users the VM can handle ($Threads_{max}$= 200). In this work similar profile was used since the SaaS platform was deployed in actual m1.small instance. The main difference was the maximal number of users set to 50 in that case. This profile information together with current readings from metering services was used to calculate required number of VM instances.

The Tenant Context Manager was responsible for assigning the weights to each Tenant Context. These weights were later used for VM calculations. The TBRAM proposes the following formula:

$$TenantContextWeight = users_{active} * (heapSize/Threads_{max}) \quad (1)$$

where active users are those whose session has not expired. Heap size is the amount of memory assigned to JVM (set in profile) and the $Threads_{max}$ is a maximal allowed number of concurrent threads for the SaaS platform. The fragment in parenthesis could be treated as an average memory usage per thread for given profile. Therefore the formula above is an estimation of required memory for given number of active users. The second formula is used to calculate the VM capacity:

$$capacity_{VM} = heapSize - ((heapSize/Threads_{max}) * Threads_{platform}) \quad (2)$$

The formula subtracts current memory usage from the maximum allowed amount described in the profile. The current memory consumption is calculated by multiplying number of SaaS platform's threads (when in idle) by the average memory per thread.

From this formula we know how much memory is available solely for users of given SaaS platform. This is because from the amount of memory assigned to JVM some part is consumed by the Tomcat's and SaaS platform's threads just to start the service. All the following threads were created to serve each user. Thanks to that it was able to estimate actual initial resources available.

The TBRAM suggest use of a knapsack algorithm to calculate the minimum number of instances needed to allocate current workload. This number was the only result yield by the algorithm since it was not interested in actual tenants' assignations to available VMs. The algorithm used the values returned by the above formulas (Formula 1, 2). Dynamic programming method was used to solve the knapsack problem quickly. This whole idea was conceptualized within Tenant-Based VM Calculator. The results of these calculations determined the number of VM instances requested from AWS cloud by the VM Manager. So the first source of information about needed number of instances came from knapsack algorithm. Yet, it was not the only one. Sometimes even the most advanced estimations are inaccurate, thus leading to discrepancy between reality and its state kept by an application. That is why it was decided to add also user factor. If several subsequent request dispatches failed then a new VM instance was requested from VM Manager on user's behalf.

4. Preliminary test results

This chapter presents the results of conducted initial tests. It presents the results in terms of server-hours, over- and underutilization as well as the cost.
When the tests were over it was time to collect measured data. All of them where gathered by Amazon CloudWatch monitoring service. That tool allows viewing some basic statistics of data in form of charts. However, in order to perform more advance analysis it was needed to download the raw data for further processing.

On the chart (Figure 2) we can see the comparison of the Base System and the TBRAM system in terms of combined resource underutilization. The results come from the incremental workload simulation tests. We can notice that during the first four months of simulated year the utilization problem did not exist in case of the TBRAM system. In the middle of the year the systems are comparable, but at the end of the year the Base System was significantly more efficient. In overall we can say that in case of the incremental tests both systems yield approximately the same resource waste (underutilization).

Fig. 2 Resource underutilization during the incremental workload tests of the systems

Now let us compare the systems in case of the peak-based workload tests. The following chart presents the results (Figure 2). Now the improvement from using the TBRAM is clearly visible. For most of the simulated year the TBRAM system was much more efficient than the Base System in terms of combined resource underutilization. Often the improvement was by 50% and more. The dynamic VM fleet tenant-based management showed its superiority when the workload was rapidly changing.

It is important to notice that both averages for %UU are generally lower that in case of traditional scaling system. But, in order to confirm that one average is statistically different than the other a t-student test was used. In our case we wanted to check if the TBRAM system was a significant improvement to the Base System.

Apart from the CloudWatch data one could also assess the systems on a financial basis. The Amazon's billing **statement** together with the AWS Simple Monthly Calculator was the data sources for the economic cost analysis. It was interested in the difference between the systems cost more than in total cost itself. Therefore in cost comparison it was taken into account only the parts that differs the both systems.

Fig. 3 Resource underutilization during the peak-based workload tests of the systems

That was the size of SaaS platform VMs fleet and the type of load balancer. All other test bed parts like the database, RCM or JMeter cluster were excluded from the price comparison. Since the both test beds were stressed with exactly the same workload by performing the same test plan, the usage of database and other resources was the same. It was thought that excluding these elements can only increase the clarity of such a comparison. Finally, in the comparison it was taken into account only the cost of EC2 instances with the SaaS platform and the load balancers. Below Amazon EC2 prices for EU (Ireland) region for 2012 are presented.

Table 1 AWS EC2 cost of on-demand instances in EU

	EC2 INSTANCE TYPE		
	m1.small	*m1.medium*	*ELB*
Cost per hour	0.085 USD	0.170 USD	0.028 USD

Table 2 AWS EC2 data transfer cost in EU region

	DATA TRANSFER TYPE		
	Transfer out	ELB in/out	ELB data proc
Cost per GB	0.120 USD	0.010 USD	0.008 USD

Table 1 shows the cost for every started hour of EC2 instance depending on its type. As explained in test bed deployment the SaaS platform VMs were deployed into m1.small EC2 instances. The SCWA element of TBRAM system was deployed into m1.medium instance as an equivalent of the Base System's ELB. The

ELB is a special EC2 instance type and it is a part of AWS EC2 services. Table 2 in turn shows the cost per GB of data transferred through AWS. Normally we pay only for data transferred out of the cloud environment. In case of the ELB there is small difference, we pay also for data transferred in/out of the ELB (even when deployed into the same Availability Zone of a region) as well as for each GB processed by it. After this short introduction to payment details of AWS it is the time to present the actual economical difference between the systems. In Table 3 EC2 cost of the differing parts of both systems is presented. One can notice that both workload types of tests where cheaper to conduct on TBRAM system with the overall cost reduction of 15.58%.

Table 3 EC2 cost difference between the systems

	Incremental	Peak-based	**Total**
Base System	13.40 USD	21.74 USD	**35.14 USD**
TBRAM	11.82 USD	17.85 USD	**29.67 USD**
Total	**25,22 USD**	**39,59 USD**	**64,80 USD**

5. Conclusions and future works

In this paper, cost-effectiveness was scrutinized from two different perspectives. First was based on server-hours number consumed by the systems during the test. The second one was based on the billing statement for AWS resources usage from Amazon. The TBRAM system used about 20% less server-hours in case of the incremental workload test and over 30% less in case of peak-based tests.

That system was also over 15% cheaper than the Base System. Therefore it can be said that the resource allocation based on tenants definitely influenced cost-effectiveness of the SaaS applications. So the answer for the first research question is positive. The second thing to examine was to check if and how the TBRAM improves a SaaS system. The results showed that this model statistically (with 97.5% accuracy) improved under-utilization of the cloud resources in case of peak-based workload. It is worth to add that other system characteristic were generally slightly improved or the same as for the Base System. According to that one can also state that the TBRAM improved the authorial SaaS system.

This research showed that TBRAM can improve the cost-effectiveness. However, this is just one side of a medal. Conformance to that model introduces non negligible development overhead.

In the course of further research authors will focus on introduction of authorial load balancer into TBRAM architecture to compare results from a SaaS system based on ELB in terms of tenant-based resource scaling.

It is expected that dynamic resource scaling based on authorial approach significantly reduce server-hours.

References

[1] Amazon Auto Scaling: http://aws.amazon.com/autoscaling/. Accessed: 2012-12-07.
[2] Architecture Strategies for Catching the Long Tail: 2006. http://msdn.microsoft.com/enus/ library/aa479069.aspx. Accessed: 2012-12-07.
[3] Armbrust, M. et al. 2009. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report #UCB/EECS-2009-28. Electrical Engineering and Computer Sciences University of California at Berkeley.
[4] Bientinesi, P. et al. 2010. HPC on Competitive Cloud Resources. Handbook of Cloud Computing. B. Furht and A. Escalante, eds. Springer US. 493–516.
[5] Chang Jie Guo et al. 2007. A framework for native multi-tenancy application development and management. 2007 9th IEEE International Conference on e-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-Commerce, and e-Services, 23-26 July 2007 (Piscataway, NJ, USA, 2007), 470–7.
[6] Chen, Y. et al. 2009. An Efficient Resource Management System for On-Line Virtual Cluster Provision. IEEE International Conference on Cloud Computing, 2009. CLOUD '09 (Sep. 2009),72 –79.
[7] Dyachuk, D. and Deters, R. 2009. A solution to resource underutilization for web services hosted in the cloud. Confederated International Conferences on On the Move to Meaningful Internet Systems, OTM 2009: CoopIS 2009, (Vilamoura, Portugal, 2009), 567–584.
[8] Espadas, J. et al. 2011. A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures. (2011).
[9] Hong Cai et al. 2009. An end-to-end methodology and toolkit for fine granularity SaaS-ization. 2009 IEEE International Conference on Cloud Computing (CLOUD), 21-25 Sept. 2009 (Piscataway, NJ, USA, 2009), 101–8.
[10] Iyer, R. et al. 2009. VM3: Measuring, modeling and managing VM shared resources. Computer Networks. 53, 17 (Dec. 2009), 2873–2887.
[11] Mc Evoy, G.V. and Schulze, B. 2008. Using clouds to address grid limitations. 6th International Workshop on Middleware for Grid Computing, MGC'08, held at the ACM/IFIP/USENIX 9th International Middleware Conference, (Leuven, Belgium, 2008).
[12] Meng, X. et al. 2010. Efficient resource provisioning in compute clouds via VM multiplexing. 7th IEEE/ACM International Conference on Autonomic Computing and Communications, ICAC-2010 (Washington, DC, United states, 2010), 11–20.
[13] Mishra, A.K. et al. 2010. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. Performance Evaluation Review. 37, 4 (Mar. 2010), 34–41.
[14] Multi-Tenant Data Architecture: 2006. http://msdn.microsoft.com/en-us/library/aa479086.aspx.Accessed: 2012-09-07.
[15] Paroux, G. et al. 2004. A Java CPU calibration tool for load balancing in distributed applications.Proceedings - ISPDC 2004: Third International Symposium on Parallel and Distributed Computing/HeteroPar '04: Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, (Cork, Ireland, 2004), 155–159.
[16] SaaS Capacity Planning: Transaction Cost Analysis Revisited: 2008. http://msdn.microsoft.com/en-us/library/cc261632.Aspx. Accessed: 2012-09-07.
[17] Stillwell, M. et al. 2010. Resource allocation algorithms for virtualized service hosting platforms. Journal of Parallel and Distributed Computing. 70, 9 (2010), 962–974.
[18] Wee, S. and Liu, H. 2010. Client-side load balancer using cloud. 25th Annual ACM Symposium on Applied Computing, SAC 2010, (Sierre, Switzerland, 2010), 399–405.
[19] Wu, Q. and Wang, Y. 2010. Performance testing and optimization of J2EE-based web applications. 2nd International Workshop on Education Technology and Computer Science, ETCS 2010, March 6, 2010 - March 7, 2010 (Wuhan, Hubei, China, 2010), 681–683.
[20] Yang, J. et al. 2009. A profile-based approach to just-in-time scalability for cloud applications.CLOUD 2009 - 2009 IEEE International Conference on Cloud Computing, September 21, 2009- September 25, 2009 (Bangalore, India, 2009), 9–16.

View publication stats