Cooperative Resource Management in a IaaS

Giang Son Tran, Alain Tchana, Daniel Hagimont, Noel Depalma

Cooperative Resource Management in a IaaS

Giang Son Tran*, Alain Tchana†, Daniel Hagimont†, and Noel De Palma‡
*University of Science and Technology of Hanoi, Vietnam. E-mail: tran-giang.son@usth.edu.vn
*University of Toulouse, Toulouse, France. E-mail: first.last@enseeiht.fr
†University of Grenoble, Grenoble, France. E-mail: noel.depalma@imag.fr

***Abstract*—Virtualized IaaS generally rely on a server consolidation system to pack virtual machines (VMs) on as few servers as possible, for energy saving. However, two situations are not taken into account, and could enhance consolidation. First, since the managed VMs can be of various sizes (small, medium, large, etc.), VMs packing can be obstructed when sizes don't fit available spaces on servers. Therefore, we would need to "split" such VMs. Second, two VMs which host replicas of the same application server (for scalability) could be "fusionned" when they are located on the same physical server, in order to reduce virtualization overhead and VMs memory footprint. Split and fusion operations lead to the management of elastic VMs and requires cooperation between the application level and the provider level, as they impact management at both levels. In this paper, we propose a IaaS resource management system which implements elastic VMs based on split/fusion operations and cooperative management. We show its benefit with a set of experiments.**

I. Introduction

Nowadays, many organizations tend to outsource the management of their physical infrastructure to hosting centers called cloud. A majority of cloud platforms implement the Infrastructure as a Service (IaaS) model where customers buy (to providers) virtual machines (VM) with a set of reserved resources. This set of resource corresponds to a Service Level Agreement (SLA) that providers are expected to guarantee.

Both providers and customers aim at saving resources. They generally implement a resource manager which is responsible for dynamically reducing the amount of used resource. At the level of the customer, such a resource manager allocates and deallocates VMs according to applications' needs at runtime to deal with different load situations and to minimize resource cost [1]. At the provider level, the resource manager relies on VM migration to gather VMs on a reduced set of machines (according to VMs' loads) in order to switch unused machines off, thus implementing a consolidation [8], [13] strategy.

However, two situations are generally not taken into account, and could enhance consolidation. First, since the managed VMs can be of various sizes (small, medium, large, etc.), VMs packing can be obstructed when sizes don't fit available spaces on servers. Therefore, we would need to "split" such VMs. Second, two VMs which host replicas of the same application server (for scalability) could be "fusionned" when they are located on the same physical server, in order to reduce virtualization overhead (which impacts applications performance) and VMs memory footprint. Split and fusion operations lead to the management of what we call elastic VMs, i.e., VMs which size can be modified dynamically. Such an approach requires cooperation between the application (customer) level and the provider level, as they impact management at both levels (a VM split or fusion initiated by the provider modifies the architecture of the application and should therefore be taken into account at the application level).

In this paper, we propose such an elastic VM cooperative scheme between the provider and the customer levels. In this novel scheme, we consider master-slave applications where a load is distributed by a master between a set of slaves. The provider is aware of the set of VMs which host slave applications. Thanks to this knowledge, the provider can propose to the customer to split a slave VM when it could improve consolidation (better fit available spaces) and it can propose to the customer to fusion slave VMs when they are gathered on the same physical machine. This paper makes the following contributions:

1) a new resource allocation model in the cloud.
2) a novel resource management vision which involves the contribution of cloud customers.
3) a prototype which considers (1) and (2).
4) an empirical demonstration of the benefit of (1) and (2) in terms of energy consumption and virtualization impacts on customers applications.

The rest of the article is organized as follows. Section II describes the context of our work. Section III motivates our work. Sections IV-VI present our cooperative resource management model between the two layers. We evaluate and compare the effectiveness of this model in Section VII. After highlighting various related works in Section VIII, we conclude and present future works in Section IX.

II. Context

Resource management is one of the most important tasks in cloud computing. Inefficient resource management has a direct negative impact on performance and cost. Ensuring performance and effective use of resources is a challenge for both the provider and the customer. Resource management in a IaaS is mostly based on the allocation, relocation and deallocation of VMs. The provider is responsible for managing resources effectively to reach his goal: minimizing operational cost. To do this, the provider manages his physical servers and allocated VMs at run time, by (1) relocating VMs (using VM live migration), in order to span as few servers as possible, then (2) switching off or suspending the unused servers to save

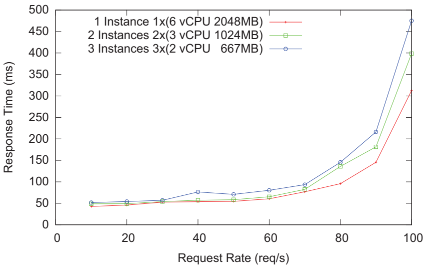

Fig. 1. Overhead caused by Collocation of VMs serving the same Tier.

energy. On the customer side, allocated resources can also be managed: the more unused VMs, the more wasting for the customer. The objective for the customer is also to minimize operational cost. To achieve this goal, the customer tends to minimize the number and size of his allocated VMs, thanks to an on-demand resource allocation policy [5]: it actively monitors the application load, detects underload and overload situations and reconfigures the application accordingly. In this paper we consider master-slave applications for the customer. Master-slave refers to a fundamental and commonly implemented pattern in distributed applications. It consists of a master component and multiple slave components, where the master distributes its workload (requests) between the associated set of slaves. The slaves execute the received requests and return the results to the master. A typical web applications in Java Platform Enterprise Edition (JEE) is a popular example of a master-slave architecture. Each of its tier (web, application and database) is replicated. Such a replicated architecture is a means for implementing scalability by cloud users in order to dynamically add or remove tier instances according to the load.

III. Motivation

Splitting and merging VMs help optimizing resource usage for the provider and performance for the customer. The main purpose of splitting VM is to improve resource utilization ratio in the provider's infrastructure: a VM can be split to fit available resource slots in physical machines. On the other hand, merging VMs allows the customer to have lower performance overhead for his application. This overhead is caused [17] by collocating several VMs on top of the same physical machine (PM). We design and evaluate a benchmark in order to confirm this performance overhead when the collocation concerns VMs belonging to the same application tier. We generate requests to a typical multi-tier web application with Apache/MySQL/PHP software. This application can be instantiated several times, each instance being encapsulated in a VM. We start the benchmark with one instance in one VM occupying one PM. We repeat the benchmark with an increased number of instances, collocated on the same PM. In each benchmark run, we configure the size of allocated VMs so that the total amount of resource is fixed (total 6 vCPUs and 2048MB memory). We gradually increase the request rate and measure the application's response time. Application response time for each benchmark (with 1, 2, or 3 application instances) is summarized on Figure 1. From this figure, we can see that when the generated request rate is higher, a higher number of VMs for providing the same amount of resources has a higher response time. These differences are due to the multiple VMs that can be merged when collocated.

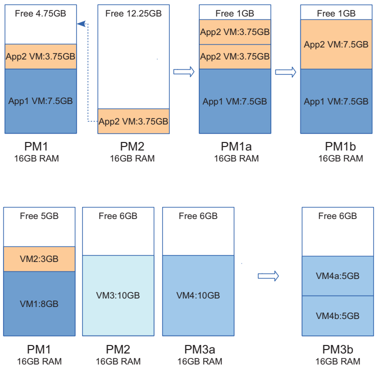

Fig. 2. The Needs of VM Merging (top) and Splitting (bottom)

In addition, merging VMs allows the provider to reduce resource waste due to VMs footprint. Therefore, a consolidation process can result in non-optimal resource management. In this situation, merging VMs can be of great interest. Figure 2 top shows an example where a VM of application 2 is migrated from PM2 to PM1 (where another VM of application 2 runs), so that the provider can shutdown or suspend PM2.

The consolidation process can also result in a situation where there would be enough available free memory to further consolidate, but this free memory is fragmented over several machines, as illustrated on Figure 2 bottom. PM3a denotes a case where the customer uses a big VM. The provider does not have the ability to migrate it to PM1 or PM2, because the free memory on PM1 or PM2 is not enough to host this big VM. In this situation, although the total free memory (11GB) is enough for VM4 (10GB), the provider still needs to keep all 3 PMs running. In contrast, if the provided VM can be split into two VMs (PM3b), the provider has the ability to migrate these VMs to PM1 and PM2 and therefore to switch PM3 off.

This section described and showed the need for the ability to split or merge VMs. These operations must be performed in accordance with cloud users since they imply the reconfiguration of their applications. The next sections present our cooperative resource management policy.

IV. General Principle of A Cooperative Resource Management

Split and merge operations lead to the management of elastic VMs, as VMs are sized according to available free space in the IaaS. We show in the following that the implementation of such a scheme requires a close cooperation between the customer and the provider levels.

Currently, requests in most *traditional IaaS* systems are in one direction only. The customer has his own application manager (*AppManager*), while the provider has his infrastructure manager (*IaaSManager*), providing and managing fixed-size VMs. The *AppManager* can invoke services from the *IaaSManager* with various types of API calls, provided by the provider. The most popular calls include: allocate, deallocate, start, restart or stop VMs. In current IaaS systems, the provider usually does not send any notifications (nor share information) about the infrastructure changes to the customer, e.g. VMs of the customer have been migrated. Hosts are transparent for the customer.

Unlike traditional counterparts, we propose a **cooperative IaaS** with the insistence on sharing knowledge about applications and VMs between the two actors, in order to improve mutual benefit and raise possibilities to improve resource management. Particularly, the customer provides information about his application (workload characteristics, tiers, etc) to the provider. In case of a multi-tier application, the shared knowledge includes tier information (which VMs are in each tier, this kind of information is typically not shared in a conventional IaaS). In our cooperative IaaS, once the information about application tiers is shared, the provider can propose to split or to merge VMs at runtime, based on the current VM placement.

In our cooperative IaaS, the resource management policy **shifts the decision to add or remove VMs from the customer to the provider**. It means that instead of requesting the provider to allocate or deallocate individual VMs as being done currently in traditional IaaS, the customer only needs to request the total computing power (amount of CPU capacity, amount of memory, etc.) that he really needs. According to these required parameters, the *IaaSManager* automatically decides how many VMs will be allocated and how big each VM will be. When the customer changes his requested resources, the *IaaSManager* either scales the application tier *horizontally* (adding/removing more VMs), *vertically* (increasing/decreasing size of the existing VMs), or both. When a consolidation decision is made, the *IaaSManager* can split and merge VMs in order to optimize consolidation, also relying on horizontal and vertical scaling to implement such split and merge operations.

We implemented a prototype of our two-level cooperative resource management system using our autonomic management system TUNe, developed in our research team [2]. To remain within page length, we do not present this implementation in this paper.

V. Cooperation Protocol

The cooperation protocol we propose is defined as a sequence of cooperative calls at runtime to achieve a particular goal. A cooperation call is similar to cloud API calls in conventional IaaS (allowing the customer to issue requests to the provider), but is extended to be used in both directions (from the provider to the customer and vice versa). We identified the following main operations in the design of the protocol:

- Subscription of an application tier by the *AppManager*;
- Modification of the amount of resource (also called quota) for an application tier, triggered by the *AppManager*;
- Splitting or merging VMs associated with an application tier, triggered by the *IaaSManager*.

This section describes the actions performed by each actor in each operation. We divide the cooperation calls into two main types: **Upcall** and **Downcall**, according to the direction of the call. A downcall is made from the customer to the provider. An upcall is in the opposite direction, from the provider to the customer. An example of cooperation call with the above operations is illustrated in Figure 3.

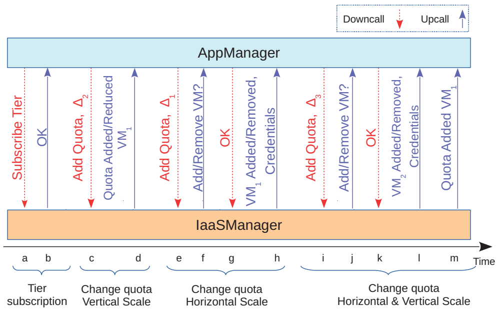
Downcall
Upcall
AppManager
Subscribe Tier
OK
Add Quota, Δ_2
Quota Added/Reduced VM$_1$
Add Quota, Δ_1
Add/Remove VM?
OK
VM$_1$ Added/Removed, Credentials
Add Quota, Δ_3
Add/Remove VM?
OK
VM$_2$ Added/Removed, Credentials
Quota Added VM$_1$
IaaSManager
a b c d e f g h i j k l m
Time
Tier subscription
Change quota Vertical Scale
Change quota Horizontal Scale
Change quota Horizontal & Vertical Scale

Fig. 3. Actions for subscription and resource addition

A. *Tier subscription*

The first operation of the cooperation resource management policy is to share the knowledge of the application tiers. The customer initiates the cooperation with a tier subscription. By subscribing all tiers and providing tier name in subsequent calls, the *AppManager* provides the notion of application instance group to the *IaaSManager*. Based on the provided group notion, the latter can perform tasks at runtime for tier and VM placement optimization.

B. *Changing tier resource*

At runtime, based on the actual application needs, the *AppManager* on the customer side can request to modify resources allocated to a specific application tier, either adding resources, or removing resources. Like previously described, our cooperative resource management policy uses elastic VMs at runtime. As a result, there are several possible solutions to respond to a single quota modification request. Based on the actual physical server usage and VM placement, the *IaaSManager* can: (1) scale the tier horizontally (add/remove VMs); (2) scale the tier vertically (add/reduce resources associated with running VMs); or (3) a combination of both. Regardless the chosen solution, the *IaaSManager* always notifies the *AppManager* so that it reconfigures the involved balancer (the master) to take into account the new weights of its instances VMs or the addition/removal of VMs. Algorithms for managing application tiers according to a resource modification request will be detailed in the next section.

C. Splitting or merging tier instances

At runtime, if the *IaaSManager* finds an opportunity to optimize its physical resource usage or application performance with VM live migration, it can propose to split a big VM of an application tier into two smaller ones (in order to fill resource holes), or propose to merge small VMs into a bigger one (in order to reduce virtualization and balancer overheads). Note that, these elastic splits or fusions can be rejected by the *AppManager* depending on the customer's goal. For example, application A may need two application instances of the same tier to implement fault tolerance, thus merging these two instances is not acceptable.

Implementation
Splitting a VM is not implemented by really "cutting" a VM. It is implemented with the following sequence:

- allocation of a new VM
- vertical down scaling of the original VM

In contrast, an elastic merge of two VMs is implemented with the following sequence:

- A deallocation of the first VM
- A vertical up scaling of the second VM

In summary, split and fusion operations are implemented using two well known operations: replication and vertical scaling. The latter are suitable for master-slave applications and are widely adopted by cloud users. This makes our contribution suitable to applications which follow the master-slave pattern, on which many applications are based (e.g. web application tiers). Notice that, in such patterns, applications are not necessarily stateless. The only capability which is mandatory for stateful applications is a way to reconciliate statefull replicas. For instance, applications such as Tomcat (with shared sessions), Joram JMS or MySQL servers are stateful and replicate-able. This capability is not proper to our contribution since it is already required by all auto scaling services (Amazon Web Services).

VI. Resource quota modification algorithms

The task of managing the group of VMs which hosts a whole application tier is shifted from the customer to the provider. While describing our cooperative IaaS approach in previous sections, we mentioned that the *AppManager* monitors the tier loads and sends requests to change the quota (the size) of the whole tier. On the other side, the *IaaSManager* is responsible for the organization of the VMs in the group to fit the required computing power, including the placement and size of each VM. When receiving a downcall from the *AppManager* to change a quota, according to the actual VM allocations status on servers, the *IaaSManager* can have multiple choices to serve this request. This section introduces the algorithm being used in the *IaaSManager* in order to handle such requests. We don't claim that our algorithm is optimal since resource management is a NP-hard problem.

For the sake of brevity, we use a single resource dimension to the description our algorithm (it can be applied to all dimensions: CPU, memory, disk and network). We use the following definitions:

- m: number of machines in the server pool
- $\psi = \{M_j, \quad 0 \leq j < m\}$: the set of running servers
- φ_j: remaining resources on M_j
- n: number of allocated VMs for a *tier*
- $\chi = \{V_k, \quad 0 \leq k < n\}$: set of running VMs for the current tier
- α_k: amount of allocated resources for V_k.

For a quota modification request in a given tier, Δ_q is the amount of resources being modified. Δ_q can be negative (reduction) or positive (increase). We identify four possible solutions to deal with a quota increase request ($\Delta_q > 0$). We prioritize vertical scaling of one or several VMs to avoid adding VMs because VM allocations are costly both in terms of time and performance

(1) **Vertical scaling of an existing VM**: the *IaaSManager* can add a specific amount of resource Δ_q to an existing VM V_k: $\alpha_k = \alpha_k + \Delta_q$, such that its hosting server M_j includes enough free resources for this vertical scaling:

$$\exists k \mid 0 \leq k < n, 0 < \Delta_q \leq \varphi_j, V_k \in M_j \tag{1}$$

The *IaaSManager* can parse the group of VMs of the tier to find a possible VM for this action. If not found, it tries the next action (see below).

(2) **Distribute the required quota change among existing VMs**. The *IaaSManager* tries to split the required quota change (Δ_q) into $p \leq n$ smaller sub-quota changes δ_i:

$$\Delta_q = \sum_{i=0}^{p-1} \delta_i \tag{2}$$

such that these sub-quota changes can be applied to a set S consisting of p VMs of the involved tier:

$$S = \{V_{s_0}, V_{s_2} ..., V_{s_{p-1}}\}, S \subset \chi, 0 \leq s_i < n, \forall i \in [0, p-1] \tag{3}$$

If it is possible to find such VM set in the tier, the *IaaSManager* then scales them vertically:

$$\alpha_{s_i} = \alpha_{s_i} + \delta_i, \forall i \in [0, p-1] \tag{4}$$

thus avoiding the need of a VM allocation. However, similarly to the previous solution, the free-resource constraints must be satisfied:

$$\delta_i \leq \varphi_j, V_{s_i} \in M_j, \forall i \in [0, p-1] \tag{5}$$

(3) **Allocation of a new VM**: if the two previous solutions cannot be applied because of the free-resource constraints (1, 5), the *IaaSManager* creates a VM V_n with $\alpha_n = \Delta_q$ and asks the *AppManager* to deploy an application instance on it.

(4) **Combination of the previous solutions** is the last solution in case each single one cannot work.

In contrast, a quota reduction request ($\Delta_q < 0$) is easier to handle:

(1) **Deallocations of running VMs**: this action has the highest priority because it is less expensive than a down

scaling. The *IaaSManager* first tries to find a VM V_k with $\alpha_k \leq \Delta_q$, and if found, proposes a VM removal to the *AppManager*. It repeats this action until there isn't any VM which is small enough to be removed.

(2) **Vertical scaling of an existing VM**: The *IaaSManager* then reduces resources from one VM allocated to the tier. It can be selected to leave the biggest space on the hosting server.

VII. Evaluations

This section demonstrates the effectiveness of our approach:

1) Validation of the ability to split and merge elastic VMs;
2) Evaluation of performance overhead reductions;
3) Evaluation of resource usage improvement.

These experiments were performed using real machines in a private cluster.

A. Experimental Setup

Hardware. The private IaaS is composed of two clusters. The first cluster (*SlowCluster*, virtualized) consists of 5 identical nodes Dell Optiplex 755, each node equipped with an Intel Core 2 Duo 2.66GHz and 4GB RAM. They are used as the resource pool. They are all installed with Debian Squeeze on top of Xen 4.1.5 and connected with 1Gbps switch. We configure each dom0 (the host operating system) in *SlowCluster* to have $^1/_3$ of a PM. The second cluster (*FastCluster*, unvirtualized) consists of two HP Compaq Elite 8300, each equipped with an Intel Core i7-3770 3.4GHz and 8GB RAM. Management systems (*IaaSManager*, NFS server and additional networking services (DNS, DHCP)) are installed on this cluster.

Software. Our target application is a multi-tier application named RUBiS [3], an implementation of eBay-like auction system.

Metrics. We define several metrics for our evaluations to measure the effectiveness of cooperative resource management:

(1) Response time is the average response time of the RUBiS application. The customer bases on it to scale his application.

(2) Physical machine utilization (ψ) is the accumulated number of powered-on physical servers for every second.

(3) VM occupation: Given a cap (the capacity of CPU resource) value $0 < c_{k,i} \leq 2$ (our *SlowCluster* has 2 cores on each machine) during a time period $t_{k,i}$ (with $0 \leq k < n$) allocated to a VM V_k, we define *the VM occupation* ω_k in our experiments as follows:

$$\omega_k = \sum_{i=1}^{t_max} c_{k,i} \times t_{k,i} \tag{6}$$

From this, we define the occupation Ω_j of an application App_j as:

$$\Omega_j = \sum_{k=1}^{m} \omega_k \tag{7}$$

Workload Profile. We generate a synthetic workload in which three different customers share the same IaaS. The workload for each application is generated by a RUBiS benchmark tool. As a result, we have three different workloads for three RUBiS applications like illustrated in Figure 4.

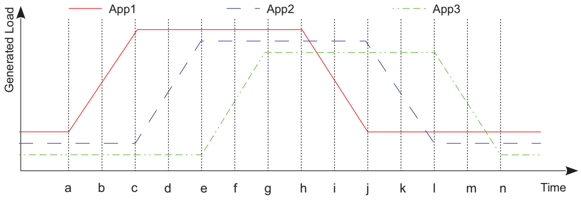

Fig. 4. Generated workload

Scenarios. We define a total of four configurations for our experiments.

Static configuration (Static). In this situation, one big VM of each RUBiS application occupies a whole physical server. For each VM, the amount of allocated resource is sufficient to deal with our experiment's workload profile. This configuration is expected to have the *lowest response time* for the customer's application and can be considered as an ”ideal” for maximizing application's performance. However, this configuration clearly wastes resources as VMs are statically oversized.

Server Consolidation Only (SCO) is a less static configuration, in which the customer does not have on-demand resource manager (i.e. without the *AppManager*), but the provider implements his *IaaSManager* with server consolidation. In other words, a fixed number of instances for each tier (two in our experiments) is provisioned and allocated for the application lifetime, even when it is idle. This configuration is expected to have the *highest application response time*. It is also expected to have best hardware utilization with VM migration based on CPU load.

Both Level, Independent (BLI) is the two-level, non-cooperative configuration. In this situation, the *IaaSManager* and *AppManager* work without any coordination: the *IaaSManager* migrates VMs to implement server consolidation, while the *AppManager* minimizes the number of application instances. In BLI, the allocated VMs' size is $^1/_3$ of a physical machine (memory and CPU). This configuration is similar to resource management in many conventional IaaS.

Both Level, Cooperative (BLC) corresponds to BLI with the cooperation of the IaaS and the applications users.

B. Scalability and Elasticity

First, we confirm scalability and elasticity of VMs with our cooperative IaaS, i.e. the ability to scale (both horizontally and vertically), merge and split VMs in BLC.

The VM and quota allocations of all applications in BLC are shown in Figure 5. Each RUBiS *AppManager* uses down calls to request quota increases during the ramp up phase of its workload (950th, 1050th, 1450th, 1550th, ... second). Depending on the VM placement and available resources on each PM at the time of those down calls, the *IaaSManager* either **vertically scales** an existing tier VM (1050th, 1550th

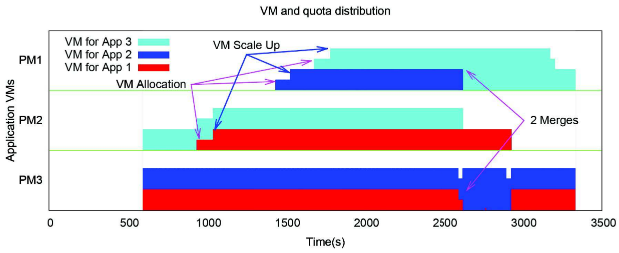

Fig. 5. BLC: VM placement and quota distribution

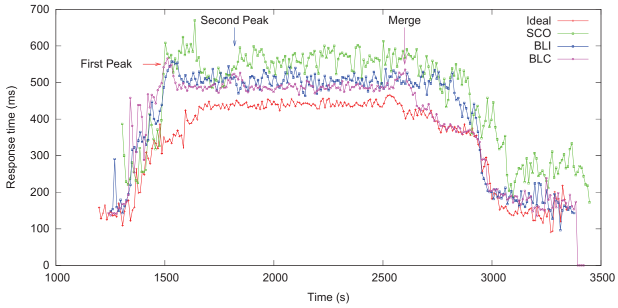

Fig. 6. Response time of App3

and 1750th second) or **horizontally scales** (allocates a new VM) the associated tier (950th, 1450th and 1650th second). Similarly, the possibilities to decrease size for a tier's VM or to remove a VM are handled in the ramp down phase of the workload. The customer's *AppManager* asks for a quota decrease with its down calls. The *IaaSManager* then decides to **reduce the size** (vertically scale) of a tier's VM (2600th, 2900th and 3150th second) or to remove it (at 2650th and 2950th second).

Notice that at runtime, with the tier knowledge provided by the *AppManagers*, the *IaaSManager* proposes to **merge small VMs into bigger ones**, in attempts to reduce overhead. For example, a cooperative merge happens at the 2650th second: the *IaaSManager* merges two VMs for application 2 (in PM1 and PM3) into one big VM (in PM3). Additionally, after a quota reduction for a VM of application 2 at 2950th second, the IaaS migrates the VM of application 1 from PM2 to PM3. It then turns PM2 off, and the provider benefits from energy saving.

C. Performance Overhead

Performance overhead for each configuration is evaluated as the difference between the response time of the considered configuration and the Static configuration (the "ideal"). We claimed that (1) performance overhead is generated by the virtualization layer; and (2) overhead can be lowered by reducing the number of VMs of the same application tier (MySQL in our experiments) when they are collocated on the same physical server.

To evaluate the performance benefit, we compare our cooperative IaaS with Static, SCO (upper bound of response time) and BLI (being used in conventional IaaS). Figure 6 compares the average response of the mentioned configurations (for the third RUBiS application instance App3). This figure confirms the response time's lower and upper bound with Static and SCO configurations, respectively. The response time of SCO during the plateau period is approximately 15%-20% higher than with Static, because of the overheads.

When compared with a static tier configuration (SCO), BLC has a more stable response time thanks to elastic VMs: additional required resources can be added on-demand and instantly (1750th second in Figure 5). Additionally, BLC does not suffer from VM migration's overhead when dealing with peak loads, unlike SCO which has a VM migration at 1750th second to deal with the increasing load, and has therefore an increased response time – SCO curve, 1750th second in Figure 6.

Compared with a non-cooperative resource management system in a conventional IaaS (BLI), BLC has a similar response time in the *ramp up* and *plateau* phases. However, the benefit of cooperation appears in the *ramp down* phase: two small VMs of App3 are merged (2650th second in Figure 5). After this merge, the whole PM1 is occupied by only one big VM for App3. This situation is similar to **Static**: only one VM for each RUBiS application instance, each physical server hosting only one VM, from 2650th to 2950th second. As a result, response time of App3, after 2650th second, stays very close to Static ("ideal" performance). BLI does not have this merge, and therefore, has higher response time, up to 10-15%. This phase clearly shows the cooperative IaaS benefit in terms of performance optimization for the customer's application.

D. VM Occupation

The customer saves cost if the resource management policy provides low Ω in the experiment. Figure 7 top summarizes the calculated Ω for the defined configurations. As can be seen, both Static and SCO have the highest VM occupations: the customer's VMs are preallocated and not scaled at runtime.

BLI and BLC have much better occupation rates, because the allocated VMs are either well used (loaded) or they are removed by the *AppManager* to improve utilization rate and to reduce costs for the customer. In our experiment, BLC has better utilization rate than BLI, with $\Omega = 2083$ and $\Omega = 2136$, respectively. Although BLC's total VM size at runtime is quite similar to BLI in all phases, BLC's improvement over BLI in terms of virtualized resources savings is shown when a new VM is allocated. BLI allocates VMs which size is $^1/_3$ of a physical machine. Reducing this size would be costly in terms of allocation time and performance overhead (too many VMs). BLC allocates VMs which size is $^1/_6$ of a physical machine, but the size of such VMs will be increased (by $^1/_6$ of a PM) as needed. Therefore, BLC implements an **intermediate step** (granularity) between no-allocation and full allocation ($^1/_3$ of a PM).

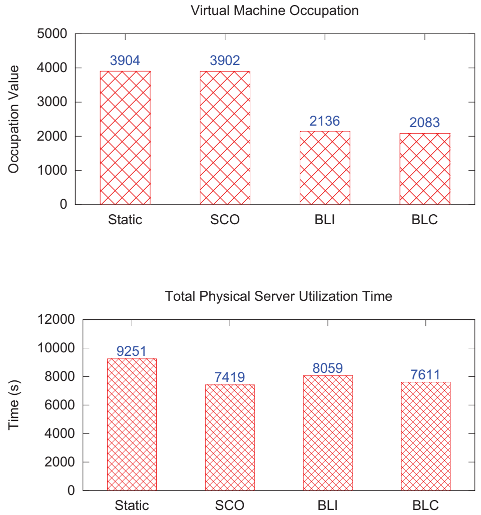

Fig. 7. Comparison of (top) VM occupation and (bottom) PM utilization

E. Physical Server Utilization

The higher PM utilization time, the more energy the provider will consume. The comparison of the defined configurations is shown in Figure 7 bottom. Static is the worst configuration for physical server utilization: it occupies all servers at runtime and there is no migration. In contrast, SCO's benefit is confirmed: it minimizes the number of physical machines being used by the provider (7419s) by packing as many VMs into as few physical servers as possible. However SCO does not have dynamic application sizing, which would save costs for the customer and allow handling peak loads. Our BLC saved an average of 5% of utilization time for all PMs when compared with BLI (7611s and 8059s, respectively). Although we cannot reach the lower bound of server utilization like SCO (7419s), this 2.5% server usage overhead in BLC allows reducing application performance by 10%-20% (Figure 6).

F. Discussion

During this research work, we identified interesting ideas, opened for discussion. When a customer's VM is split into two smaller ones, this action poses a disadvantage to the customer: reduced amount of "effective resource" allocated for his application. This can be explained as the number of customer's operating system instance is doubled, from 1 to 2. Each operating system consumes VM resources, while the total amount of resources for two small VMs is unchanged (matches with the original big VM). As a result, resources for the customer application (what is really available) is reduced: the customer may not prefer to accept split proposals at runtime. On the other hand, a VM fusion reduces the number of operating system instances and increases the amount of "effective resource" allocated to the customer application. As a result, the provider wastes resources.

To increase the chance of split acceptance, we think that an extra amount of resources should be offered to the customer when a split is proposed. This extra encourages customers to allow more splits and helps the provider to optimize his infrastructure. However, considering this extra in a production environment could conflict with current pay-as-you-go billing model. Therefore, we would need a precise resource accounting system, which is not in the scope of this paper and is considered as a perspective of our research contribution. Regarding resource waste caused by VM fusion, the provider can reduce the size of the resulting VM by the amount of resource corresponding to the execution of an operating system.

Regarding our solution, one can ask the following question: is there a lot of additional work for cloud users which could limit the adoption of our cooperative model? The AppManager is a generic framework which needs to be adapted in order to implement how new software replicas are integrated in the user's application. This generic framework already implements all the negotiation protocol so that the cloud user only focuses on his application core business. The user already (without our system) had to implement elasticity (replica management: addition, removal) for his application. So additional work is very limited.

VIII. Related Work

a) Memory footprint improvements: Significant research has been devoted to improving workload consolidation in data centers. Some studies have investigated reducing VM memory footprint to increase the VMs consolidation ratio as do. Among these, memory compression and memory over commitment ([12], [14]) are very promising. In the same vein, [11] extends the VM ballooning technique to software to increase the density of software collocation on the same VM. Xen offers what it called "stub domain"[1]. This is a lightweight VM which requires very few memory (about 32MB) for its execution.

b) Uncoordinated Policies: Many research works focus on improving resource management on the customer side [18], [6], [9]. These works aim at improving the workload prediction and the allocation of VMs for replication. On the provider sides, research works mainly focus on (1) *size of resource slices*, i.e. provided VM's size; or (2) *virtual machine placement*, i.e. allocation and migration of virtual machines among physical servers to improve infrastructure utilization ratio. Various algorithms are proposed to solve the VM packing problem [4], [15], taking into account various factors like real resource usage, VM loads, etc.

c) Cooperative Policies: [10] describes a model to coordinate different resource management policies from both cloud actors' point of views. The proposed approach allows the customer to specify his resource management constraints, including computing capacity, load thresholds for each host and for each subnet before an allocation of a new VM, etc. The authors also describe a set of affinity rules for constraining VM's collocation in the IaaS, which is a form of knowledge sharing. The authors claimed that this model allows an efficient

[1] http://wiki.xen.org/wiki/StubDom

allocation of services on virtualized resources. This work is a first step in the direction of coordinated policies.

[16] is closely-related to ours about knowledge sharing in two level resource management. The authors proposed an autonomic resource management system to deal with the requirements of dynamic provisioning and placement of VMs, taking both application level SLA and resource cost into account, and to support various types of applications and workloads. Globally, the authors clearly separate two levels of resource management: Local Decision Modules and the Global Decision Module (similar to our *AppManager* and *IaaSManager*, respectively). These two decision modules work cooperatively: the LDM makes requests to the GDM to allocate and deallocate VMs, the GDM may request changes to the LDMs about allocated virtual machines. [7] presents Quasar, a cluster (not virtualized as we study in this paper) management solution which adopts a solution which is philosophically close to our solution. [7] claims that cluster users are not able to correctly estimate the amount of resource needed by their applications to run efficiently. It allows users to express their needs in terms of QoS constraints, instead of low level resource requirements, and the management system will allocate the appropriate amount of resources that will ensure that QoS, while increasing physical machines utilization rate. Even if this work does not consider IaaS environments, like our solution it ships knowledge to the system about applications and their expected QoS, thus enabling a smarter resource management.

IX. Conclusions and Perspectives

This paper proposed a direction to use the combination of cooperative resource management with elastic VMs. Information about the customer's application (e.g. tier instances) is shared with the IaaS provider. Using the shared knowledge, the provider can propose to split or to merge VMs of the customer. The evaluations showed that our cooperative IaaS outperforms traditional two-level non-cooperative resource management with: (1) lower performance overhead (better response time for the customer's application), (2) better VM usage (reducing costs for the customer with finer grain resource blocks), and (3) better physical resource usage (reducing energy and costs for the provider).

During this research work, we identified interesting ideas, opened for discussion. When a customer's VM is split into two smaller ones, this action poses a disadvantage to the customer: reduced amount of "effective resource" allocated for his application. This can be explained as the number of customer's operating system instance is doubled. We can convince users to adopt this scheme by 2 means: (1) providing a fair/precise accounting service in order to enforce that whenever a VM V is split into 2 VMs V1 and V2, application_performance(V) = application_performance(V1 + V2), (2) the provider can propose an attractive/incentive pricing policy.

Acknowledgment

The work reported in this article benefited from the support of the French National Research Agency through project Ctrl-Green and from the French FSN (Fond national pour la Societe Numerique) through project OpenCloudWare.

References

[1] Hady AbdelSalam, Kurt Maly, Ravi Mukkamala, Mohammad Zubair, and David Kaminsky, 'Towards Energy Efficient Change Management in a Cloud Computing Environment," AIMS 2009.

[2] Alain Tchana, Suzy Temate, Laurent Broto, and Daniel Hagimont, 'TUNeEngine: An Adaptable Autonomic Administration System," JSCSE 2013.

[3] RUBiS, 'http://www.rubis.fr/," visited on May 2014.

[4] Norman Bobroff, Andrzej Kochut, and Kirk Beaty, 'Dynamic placement of virtual machines for managing SLA violations," IM 2007.

[5] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopa, 'Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," HPCC 2008.

[6] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato, 'Optimal virtual machine placement across multiple cloud providers," APSCC 2008.

[7] Christina Delimitrou and Christos Kozyrakis, 'Quasar: Resource-Efficient and QoS-Aware Cluster Management," ASPLOS 2014.

[8] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall, 'Entropy: A consolidation manager for clusters," VEE 2009.

[9] Deepal Jayasinghe, Calton Pu, Tamar Eilam, Malgorzata Steinder, Ian Whalley, and Ed Snible, 'Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement," SCC 2011.

[10] Kleopatra Konstanteli, Tommaso Cucinotta, Konstantinos Psychas, and Theodora Varvarigou, 'Admission control for elastic cloud services," CLOUD 2012.

[11] Tudor-Ioan Salomie, Gustavo Alonso, Timothy Roscoe, and Kevin Elphinstone. "Application Level Ballooning for Efficient Server Consolidation," EuroSys 2013.

[12] Prateek Sharma and Purushottam Kulkarni. "Singleton: System-wide Page Deduplication in Virtual Environments," HPDC 2012.

[13] Liang Liu, Hao Wang, Xue Liu, Xing Jin, WenBo He, QingBo Wang, and Ying Chen, 'Greencloud: a new architecture for green data center," ICAC-INDST 2009.

[14] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. "An Empirical Study of Memory Sharing in Virtual Machines," USENIX ATC 2012.

[15] Hidemoto Nakada, Takahiro Hirofuchi, Hirotaka Ogawa, and Satoshi Itoh, 'Toward virtual machine packing optimization based on genetic algorithm," Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living 2009.

[16] Hien Nguyen Van and Frederic Dang Tran, 'Autonomic virtual resource management for service hosting platforms," CLOUD 2009.

[17] Indrani Paul, Sudhakar Yalamanchili, and Lizy K. John, 'Performance impact of virtual machine placement in a datacenter," IPCCC 2012.

[18] Andres Quiroz, Hyunjoo Kim, Manish Parashar, Nathan Gnanasambandam, and Naveen Sharma, 'Towards autonomic workload provisioning for enterprise grids and clouds," GRID 2009.