RESOURCE MANAGEMENT OF VIRTUAL INFRASTRUCTURE FOR ON-DEMAND SAAS SERVICES

Rodrigue Chakode[1], Blaise Omer Yenke[1,2] and Jean-François Méhaut[1]
[1]*INRIA, LIG Laboratory, University of Grenoble, Grenoble, France*
[2]*Dept. of Computer Science, University of Ngaoundere, Ngaoundere, Cameroon*

Keywords: On-demand Software-as-a-Service, Cloud computing, Virtualization, Resource sharing, Scheduling.

Abstract: With the emerging of cloud computing, offering software as a Service appears to be an opportunity for software vendors. Indeed, using an on-demand model of provisioning service can improve their competitiveness through an invoicing tailored to customer needs. Virtualization has greatly assisted the emerging of on-demand based cloud platforms. Up until now, despite the huge number of projects around cloud platforms such as Infrastructure-as-a-Service, less open research activities around SaaS platforms have been carried on. This is the reason why our contribution in this work is to design an open framework that enables the implementation of on-demand SaaS clouds over a high-performance computing cluster. We have first focused on the framework design and from that have proposed an architecture that relies on a virtual infrastructure manager named OpenNebula. OpenNebula permits to deal with virtual machines life-cycle management, and is especially useful on large scale infrastructures such as clusters and grids. The work being a part of an industrial project[1], we have then considered a case where the cluster is shared among several applications owned by distinct software providers. After studying in a previous work how to implement the sharing of an infrastructure in such a context, we now propose policies and algorithms for scheduling jobs. In order to evaluate the framework, we have evaluated a prototype experimentally simulating various workload scenarios. Results have shown its ability to achieve the expected goals, while being reliable, robust and efficient.

1 INTRODUCTION

Cloud computing has highlighted the suitability of the on-demand model of invoicing against the subscription model that has been applied until recently by application service providers also know as ASP. With cloud computing, Software-as-a-Service (SaaS) appears to be a real opportunity for ASPs. Indeed, they can improve their competitiveness by enabling on-demand remote access to software (as service or utility) in addition to traditional license and/or subscription models. Virtualization has greatly assisted the emerging of cloud computing that enabled the effective use of remote computing resources as utility. Up until now, open research activities around cloud computing are mostly focused on Infrastructure-as-a-Service clouds leading to huge ecosystem of high-quality resources. There are no real open research projects around SaaS clouds.

Our contribution in this work is to design an open framework that enables the implementation of on-demand SaaS clouds over a high-performance computing (HPC) cluster. We have first proposed an architecture that relies on virtual machines to execute the jobs associated to service requests. We have built a resource manager that relies on OpenNebula, a virtual infrastructure manager that allows to deal with virtual machines life-cycle management on large scale infrastructures such as clusters and grids. The resource manager provides core functionalities to handle requests and schedule the associated jobs. Then, guided by the requirements of the Ciloe project[1], we have considered that the resources of the underlying cluster are shared among several applications owned by distinct providers, which contribute to purchase the infrastructure and keep it up. We have focused on scheduling of service requests within such a context and proposed policies and algorithms. We have implemented a prototype which is evaluated by simulating various workloads, that derive from a real production system workload. Experimental results have

[1]This work is funded by Minalogic through the project Ciloe (http://ciloe.minalogic.net). Located in France, Minalogic is a global business competitive cluster that fosters research-led innovation in intelligent miniaturized products and solutions for industry.

Chakode R., Omer Yenke B. and Méhaut J. (2011).
RESOURCE MANAGEMENT OF VIRTUAL INFRASTRUCTURE FOR ON-DEMAND SAAS SERVICES.
In *Proceedings of the 1st International Conference on Cloud Computing and Services Science*, pages 352-361
DOI: 10.5220/0003387503520361
Copyright © SciTePress

shown its ability to achieve the expected goals, while being reliable, robust and efficient.

We will first describe the SaaS cloud within the Ciloe project context, and then show the different obstacles that we have faced in order to achieve the project (Section 2). We will then present the related work around SaaS clouds and the different approaches concerning those obstacles (Section 3). After that, we will describe the design and the architecture of the proposed framework (Section 4), along with the policies and the algorithms of scheduling used by the resource manager (Section 5). After presenting the implemented prototype, we will show the results of its experimental evaluation (Section 6). Finally, we will present our conclusion and future work (Section 7).

2 CONTEXT: THE CILOE PROJECT

The Ciloe project aims at developing an open framework to enable the implementation of software-as-a-service (SaaS) (Gene K. Landy, 2008)(Turner et al., 2003) cloud computing (Vaquero et al., 2009) platforms. A SaaS cloud is illustrated on Figure 1. The cloud relies on a HPC cluster, while the model of accessing service is on-demand. A customer submits a request along data through Internet. The execution of the job that will process the request is scheduled. After processing the output is returned to the customer. Both software and computing resources (computing nodes, storage, etc.) are owned by the software vendor, which is also the service provider.

In our previous work (Chakode et al., 2010), we have mentioned the case of clusters that can be shared among several businesses in order to minimize the impact of an initial deployment on their budgets. This is the reason why this particular project involves three small and medium software vendors. We have also shown that when using this model of service providing, we can face different obstacles concerning job and resource management, such as:

- **Schedule of on-Demand Request.** Allocate resources for executing requests would be transparent: It would be achieved internally without explicit constraints provided by customers. Additionally, the allocation would ensure some prioritization among customers: the providers can differentiate the end-to-end quality of service offered to their customers, distinguishing for instance partners or regular customers against punctual customers. However, the delivered service has to be well defined (for instance in term of reasonably delivery time) to ensure customers satisfaction.
- **Fair-share of Resource.** The underlying HPC cluster is shared among several businesses collaborating to purchase the cluster and keep it in operational condition. Therefore, the sharing has to be fair: the application of each business has to be guaranteed with a share or ratio of resource usage according to its investment on the infrastructure.
- **Efficient Use of Resource.** When resources are idle, the software providers can use them to run internal jobs, such as regression testing. However, the jobs associated to such tasks are less prioritized than the customer ones.

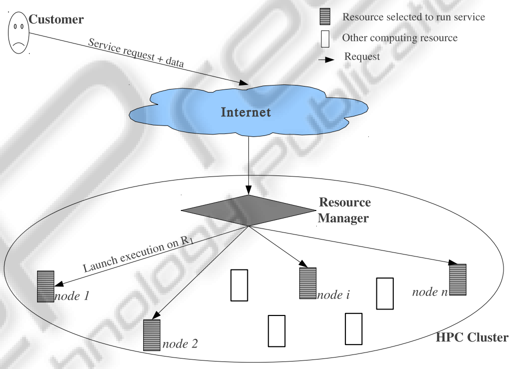

Figure 1: The model of the expected infrastructure for SaaS on-demand.

3 RELATED WORK

Up until now, the implementation of on-demand SaaS platforms was studied by industries, leading to many platforms such as SalesForce (for,), Google (goo,), Adobe (ado,), Intacct (int,), etc. Since these platforms are closed, there are not open documentation about how they deal with resource management. However, in the area of high performance computing (HPC), the problems of sharing computing resources and to use them efficiently have been studied.

3.1 Sharing of Computing Resources

Most batch job schedulers implement fair-share disciplines, from native operating systems (e.g. Linux), to high level batch job schedulers such as Maui (Jackson et al., 2001). In fair-sharing systems, resources are equitably allocated to run jobs or group of jobs accordingly to shares assigned to them (Kay, J. and Lauder, P., 1988)(Li and Franks, 2009). Fair-share scheduling permits to avoid starvation, while allowing jobs to benefit from the aggregated power of processing resources. Fair-share can be observed over long periods of time (Jackson et al., 2001; Kay, J. and Lauder, P., 1988; Jackson et al., 2001) or over few clock cycles (Linux systems). In most of scheduling systems, it is implemented through dynamic priority policies guaranteeing higher priorities to jobs or groups of jobs that have used few resources.

This approach does not seem to be suitable in the Ciloe case. Indeed, the execution of some tasks could be significantly delay if all resources are used by long-term jobs. Yet, even if the partners want to benefit from the aggregated power of computing nodes, they would expect a reasonable waiting time for their tasks. For a given partner, this expectation could be particularly high when the amount of resources he has used is less than the ratio of resources for which he has invested.

3.2 Efficient Use of Computing Resources

The need of using computing resources efficiently led to the emerging of advanced scheduling policies such as backfilling (Lawson and Smirni, 2002), against the classical first-come first-serve (FCFS) policy adopted in earlier job schedulers. A backfilling-applying scheduler allows newer jobs requiring less resources to be executed than the former ones. Despite the fact that it may lead to starvation issues for big jobs, the backfilling approach allows a more effective utilization of resources avoiding wasting idle time.

Our approach is to carry out several novelties. The design of the SaaS resource manager we have proposed is opened from its architecture to internal policies and algorithms of scheduling jobs. It relies on a generic model of resource management that considers that the underlying computing infrastructure is shared among several applications owned by distinct SaaS providers. The policies and algorithms used to enforce this sharing permit to guarantee shares of resource use to each application, even in period of high load and/or high competition to accessing resources. Since the effectiveness of the utilization of the whole resources is a major point in our project, we also allow backfilling.

4 THE PROPOSED FRAMEWORK

As previously mentioned (Chakode et al., 2010), it has been shown that scheduling SaaS requests on-demand upon such a shared cluster should enable flexibility and easy reconfigurability in the management of resources. A dynamic approach of allocating resources has been proposed. This approach aimed at guaranteeing fair-sharing statistically, while improving the utilization of whole system resources.

We think that using virtual machines would be a suitable solution to implement this approach. The suitability of virtual machines for sharing computing resources has been studied (Borja et al., 2007). The authors have claimed and shown that using virtual machines allows to overcome some scheduling problems, such as schedule interactive applications, real-time applications, or applications requiring co-scheduling. Furthermore, virtual machines enable safe partitioning of resources. Being easily allocable and re-allocable, they also enable easy reconfigurability of computing environments. While the main virtual machines drawback has been performance overhead, it has been shown that this overhead can be significantly reduced with specific tuning, see for example the works introduced in (Intel Corporation, 2006), (AMD, 2005), (Yu and Vetter, 2008), (Jone,), (Mergen et al., 2006). Tuning being typically implementation-dependent, we do not consider this aspect in this work.

4.1 Global Architecture

In the model we have proposed, jobs would be run within virtual machines to ensure safe node partitioning. The cluster is viewed as a reconfigurable virtualized infrastructure (VI), upon which we build a resource manager component to deal with handling requests, and scheduling the associated jobs on the underlying resources pool. This system architecture is shown on Figure 2. At the infrastructure level, the scheduler relies on a VI Manager (VIM), to deal with usual virtual machine live-cycle management capabilities (creation, deployment, etc.), over large scale infrastructures resources. Among the leading VIMs which are OpenNebula (Sotomayor et al., 2009b), Nimbus (Keahey et al., 2005), Eucalyptus (Nurmi et al., 2009), Enomaly ECP (eno,), VMware vSphere (vmw,), we have chosen OpenNebula considering the following key-points, which are further detailed in (Sotomayor et al., 2009a). OpenNebula is scalable (tested with up to $16,000$ virtual machines), open source and it uses open-standards. It enables the programmability of its core functionalities through application programmable interface (API), and supports all of popular Virtual Machine Monitor (VMM) including Xen, KVM, VMware, and VirtualBox.

The proposed resource manager replaces the default OpenNebula scheduler (mm_sched), and acts as the entry-point for all requests submitted on the infrastructure. As mentioned above, it provides core functionalities for handling requests, scheduling the execution of the associated jobs, monitoring their execution, etc. To be more precise, it is in charge of various tasks: selecting jobs to run, preparing virtual machines template along with suitable software stacks (virtual appliances), selecting nodes onto which the virtual machines will be run, and to then request OpenNebula to perform the deployment. The scheduler relies on the XML-RPC interface of OpenNebula to communicate with it.

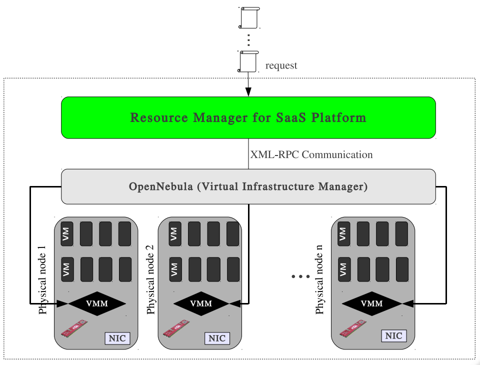

Figure 2: Model of the Software-as-a-Service platform.

4.2 Resource Manager for SaaS Platform

Like classical bash schedulers such as PBS (H. Feng and Rubenstein, 2007) and OAR (Capit et al., 2005), the scheduling system is designed following a client/server architecture. In this section, we will present the design of the server part consisting of four main modules: an admission module, a scheduling module, an executive module, and a monitoring module. Schematically, the modules work together following the flow described on Figure 3.

4.2.1 Admission Module

In charge of handling user requests, it processes the following algorithm.

1: Listen for new request.
2: Once a request arrives, the module parses the request to check its validity (the syntax of valid requests is well defined and known by the system). If the request is not valid, it is rejected and the user is notified while the processing returns to step 1.
3: When the request is valid, the associated job is created, masked as in *pending*, and added to suitable service queue.
4: A wake-up signal is emitted towards the scheduling module while the processing returns to step 1.

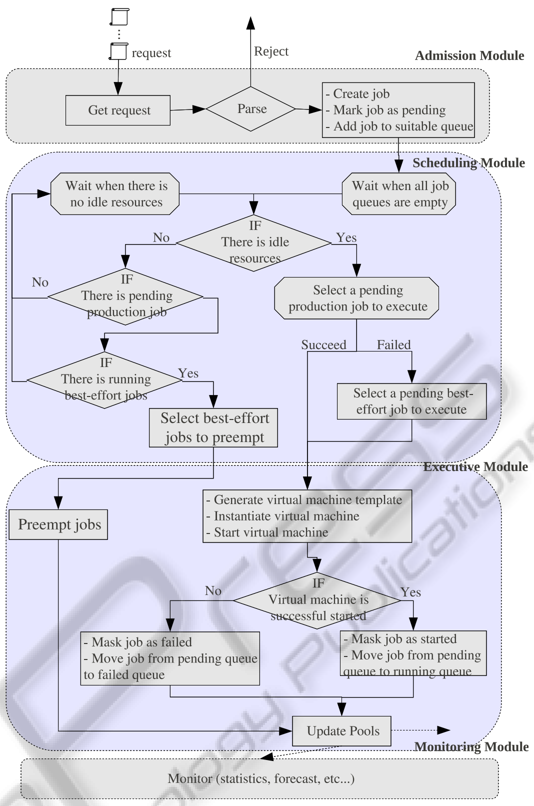
request
Reject
Admission Module
Get request
Parse
- Create job
- Mark job as pending
- Add job to suitable queue
Scheduling Module
Wait when there is no idle resources
Wait when all job queues are empty
IF There is idle resources
No
Yes
IF There is pending production job
Select a pending production job to execute
IF There is running best-effort jobs
Yes
Select best-effort jobs to preempt
Succeed
Failed
Select a pending best-effort job to execute
Executive Module
Preempt jobs
- Generate virtual machine template
- Instantiate virtual machine
- Start virtual machine
IF Virtual machine is successful started
- Mask job as failed
- Move job from pending queue to failed queue
- Mask job as started
- Move job from pending queue to running queue
Update Pools
Monitoring Module
Monitor (statistics, forecast, etc...)

Figure 3: Overview of the resource manager workflow.

4.2.2 Scheduling Module

In charge of selecting the job to execute it processes the selection algorithms described in the subsection 5.4.1. Its processing workflow is described as follows.

1: Wait when there is no queued job.
2: Once waking up, check if there are idle resources. If there are any, go to step 4; Otherwise, continue to step 3.
3: If there are queued production jobs while there are running best-effort jobs, it tries to preempt some of the later to free sufficient resources to run the former. Otherwise, it waits for resources to become idle (a wake up signal is emitted when a job completes). The processing returns to step 2 after the waking up.
4: Check whether or not there are queued production jobs. If there are any, select a job to execute using the algorithm of the subsection 5.4.1. Otherwise, select a queued best-effort job using the algorithm of the subsection 5.4.2.

5: After selecting a job, it forwards the job to the executive module (that is responsible for launching it), and it resumes the processing to step 2.

4.2.3 Executive Module

It is in charge of allocating resources and launching the execution of jobs. Aware that in general case some applications can be distributed, we have decided to focus mainly on applications that required only one virtual machine, while the virtual machine can have several CPU/cores. Otherwise, we should have introduced a synchronization so that the job starts at the end of virtual machines startup. The executive module processes as follows:

1: Wait when there is any task to carry out.
2: Once a job has to be launched, it generates the configuration of the associated virtual machine. The configuration data consist of a virtual machine template (indicating among other things, memory, CPU, swap, location of image file, etc.). The template and the image file contain contextualization data that permit: (i) to automatically set the execution environment and start the job at the virtual machine startup; (ii) send back a notification when the job is completed.
3: Select a host which free resources can match the virtual machine requirements. The host selection is based on a greedy algorithm that iterates through the host pool until reach a suitable host.
4: Request the underlying virtual infrastructure manager to instantiate the virtual machine.
5: Request the underlying virtual infrastructure to deploy the virtual machine onto the selected host.
6: Wait for the job to start and mark it as in ”running”, update pools, and resume the processing to step 1.

4.2.4 Monitoring Module

This is a utility module. Among other things, it periodically gathers statistics (concerning for instance resource usage, queues state, etc.), and logs them. Such information can be used, for example, for analyzing trends and making forecasts.

5 FAIR AND EFFICIENT SCHEDULING

It can be deduced from the workflow introduced in the subsection 4.2 that the guarantee of fairness and efficiency highly depends on algorithms of scheduling jobs.

5.1 Scheduling Assumptions

Two classes of jobs can be distinguished: production jobs and best-effort jobs. Their scheduling obeys to some assumptions that aim at making tradeoffs so to be fair regarding the sharing, while being effective as possible regarding resource utilization.

5.1.1 Production Jobs

They consist of jobs associated to the execution of customer requests. For this reason, it is considered that they can not be stopped or preempted so to guarantee end-to-end performance. Such jobs are *higher prioritized* to accessing resources than best-effort jobs (introduced below). We also distinct two subclasses of production jobs: *long-term jobs* and *short-term jobs* according to a given threshold duration.

Concerning production jobs, the tradeoffs consists in applying the fair-guaranteed only on long-term jobs. This would permit to overlap the fragmentation or the wasting of resources caused by static partitioning by short-term jobs (Chakode et al., 2010), which can be executed regardless of the sharing constraint. Therefore, resources can be assigned for executing long-term jobs if the share-guaranteed is respected. Furthermore, with this assumption we expect a certain flexibility in order to improve the utilization of resources, while guaranteeing that a possible overuse (that can occur after allocating resources to short-term jobs) does not persist too long.

5.1.2 Best-effort Jobs

They consist of jobs associated to requests submitted by the software providers. These jobs are less prioritized than production jobs. They should be preempted when there aren't any sufficient resources to run some queued production jobs. The main idea behind allowing best-effort is to improve resource utilization. Since they can be preempted every time, best-effort jobs are scheduled regardless the constraints on fair-sharing. Thus, as long as there are sufficient free resources, those resources can be used to execute the jobs. In short, the policy used to select the jobs to be preempted is based on *newer-job first* order, which means that newer jobs are preempted before the former ones. This policy aims at reducing the time required to preempt jobs (a short-time running job would take less time).

5.2 Scheduling Policies

The jobs scheduling is based on a multi-queues system. Each application has two queues: one for production jobs and the other one.

5.2.1 Queue Selection Policy

In order to avoid the starvation of some queues, the selection of the service's queue, in which a job will be selected to execute, relies on a round-robin policy. All available queues are grouped in two lists of queues (a list of production queues and a list of best-effort queues). Each list is reordered after selecting and planning a job to be executed. The reordering moves the queue in which the job has been chosen to the end of the related queue list.

5.2.2 Production Job Selection Policy

The selection of the queued production job to be executed is based on a priority policy. Indeed, each request is assigned with a priority according to a weight assigned to the associated customer. The weights are related to the customers importance to software providers. Backfilling is allowed when selecting a job from each queue.

5.2.3 Best-effort Job Selection Policy

The selection of the best-effort job to be executed is based on a First-Come First-Served (FCFS) policy. The queued jobs are planned onto resources according the their arrival date, as long as there are sufficient idle resources. The policy also allows backfilling.

5.3 Share Guaranteed and System Utilization

The fair-sharing is enforced assigning a ratio of resource usage to each application. Before executing a job, the scheduler should check if the associated service's resource usage is less or equal than the granted ratio after allocating resources. Resource usage is then evaluated through a weighting relation among computational and I/O resources. The weighting is similar to that of Maui *job priority factors*.

$$\begin{aligned} usage = w_1 * mem\_usage + w_2 * cpu\_usage \\ + w_3 * network\_usage + \ldots \end{aligned}$$

Each w_i is a weight assigned to each kind of resources. However, we have only considered memory and CPU resources, and used a simple weighting rule expressed as follows:

$$usage = Max(mem\_usage, cpu\_usage)$$

5.4 Job Selection Algorithms

We will present the essential of the algorithm instead of specific implementation details. We assume that we know (or evaluate in a prior stage) the CPU and memory requirements, and the job duration.

5.4.1 Production Jobs Selection

The selection of production jobs to execute takes into account the available idle resources, priorities among jobs, the assigned ratios of resource usage, job duration, and submission dates. It allows backfilling, and as noted in the subsection 5.1, the constraint of share-guaranteed is only applied on long-term jobs. An overview of the algorithm is shown on Figure 4, and further described in the Algorithm 1

5.4.2 Best-effort Jobs Selection

The algorithm used for selecting the best-effort job to be executed is similar to Algorithm 1. However, there is one main difference. Indeed, resource selection is not subjected to compliance with shares-guaranteed, and then the associated usage of resources is not accounted when checking it (steps 10 and next).

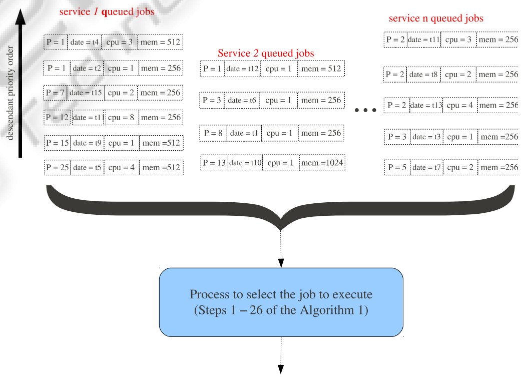
service 1 queued jobs
Service 2 queued jobs
service n queued jobs
descendant priority order
P = 1 | date = t4 | cpu = 3 | mem = 512
P = 1 | date = t2 | cpu = 1 | mem = 256
P = 7 | date = t15 | cpu = 2 | mem = 256
P = 12 | date = t11 | cpu = 8 | mem = 256
P = 15 | date = t9 | cpu = 1 | mem =512
P = 25 | date = t5 | cpu = 4 | mem =512
P = 1 | date = t12 | cpu = 1 | mem = 512
P = 3 | date = t6 | cpu = 1 | mem = 256
P = 8 | date = t1 | cpu = 1 | mem = 256
P = 13 | date = t10 | cpu = 1 | mem =1024
P = 2 | date = t11 | cpu = 3 | mem = 256
P = 2 | date = t8 | cpu = 2 | mem = 256
P = 2 | date = t13 | cpu = 4 | mem = 256
P = 3 | date = t3 | cpu = 1 | mem =256
P = 5 | date = t7 | cpu = 2 | mem =256
Process to select the job to execute (Steps 1 – 26 of the Algorithm 1)

Figure 4: Overview of the job selection principle. Production queues are sorted according to job priorities.

6 EVALUATION

In order to validate the proposed resource manager, we have built a prototype named Smart Virtual Machines Scheduler (*SVMSched*). It is a multi-process application, which processes represent the components introduced in subsection 4.2.

6.1 Experimental Environment

The experimental platform has consisted of 8 multi-

Algorithm 1: Select the production job to execute.

Require: sorted job queues: Let j and j' be two jobs, q_p be a queue of production jobs, and Q_p the set of all production queues. $j \neq j' \in q_p \in Q_p$; $priority(j)$ a function that returns the priority of a given job j; $subTime(j)$ a function that returns the submission time of a given job j; $rank(j, q)$ a function that returns the rank of a given job j in the queue q. Then, if $priority(j) < priority(j')$ or if $priority(j) = priority(j')$ while $subTime(j) < subTime(j') \Rightarrow rank(j, q_p) < rank(j', q_p)$.

Ensure: If the variable *found* is **true**, then the variable *selectedJob* contains the job to be executed. Otherwise, it means that no job can be executed while respecting the scheduling assumptions. When *found* is **true**, the value of the variable *totalCompliance* (which can be **true** or **false**) allows to know whether the fair-sharing compliance will be overridden after execution.

```
 1: found ← false
 2: totalCompliance ← false
 3: q ← getNextCandidateQueue ($Q_p$)
 4: while ( totalCompliance = false )
          and ( q != endOfQueue_set ($Q_p$) ) do
 5:   service ← getService (q)
 6:   grantedUsage ← getGrantedUsage (service)
 7:   job ← getFirstJob (q)
 8:   while ( job != endOfQueue (q) )
            and ( totalCompliance = false ) ) do
 9:     if ( sufficientResourcesToRun (job) ) then
10:       nextUsage ← expectedUsage(job, service)
11:       if ( nextUsage ≤ grantedUsage )
             ( ( nextUsage > grantedUsage )
                 and ( hasShortDuration (job) ) ) then
12:         selectedQueue ← q
13:         selectedJob ← job
14:         found ← true;
15:         if ( nextUsage ≤ grantedUsage ) then
16:           totalCompliance ← true
17:         end if
18:       end if
19:     end if
20:     job ← getNextJob( q )
21:   end while
22:   q ← getNextCandidateQueue ($Q_p$)
23: end while
24: if ( found = true) then
25:   updateQueuesOrder (selectedQueue, $Q_p$)
26: end if
```

core nodes of the Grid5000[2] Genepi cluster located in Grenoble. Each one of the nodes consists of 2 chips (CPU) of 4 Xeon cores (2.27Ghz), 24GB RAM, 2 NICs (Ethernet 1Gbit/s, Infiniband 40Gbit/s). We have used Xen version 3.4.2 as the backend VMM for OpenNebula which has been deployed on six of the nodes working as cluster nodes. OpenNebula version 1.4 and *SVMSched* have been installed on the two other nodes. We have used an NFS-based OpenNebula deployment where the images of virtual machines are stored in an NFS-attached repository. The NFS and the OpenNebula server have been mutualized on the same node. Each virtual machine has to have a round not-nil number of CPU-cores dedicated, allowing up to 48 virtual machines in the virtual infrastructure. The NFS system relies on the Infiniband network through IP-over-Infiniband, while the Ethernet network supports the infrastructure virtual network.

6.2 Workload

We have simulated the system workload using a filtered workload from the SHARCNET log[3]. We have extracted the last 96 jobs related to partitions 8, 4, and 2 with a number of allocated CPU less or equal to 8. The resulting workload consists of 96 jobs among which 65 of them are related to partition 8, 12 of them to partition 4, and 19 of them to partition 2. In order to transpose this workload in our context, we have assumed that the requests related to partitions 8, 4, 2 are associated to three applications or services denoted $App1$, $App2$, and $App3$, respectively.

6.3 Ability of Enforcing Share-guaranteed

In a first experiment, we have evaluated the behavior of the resource manager when the constraint of share-guaranteed is applied to all jobs, regardless of their duration. Precisely, we have evaluated the ability to enforce the policy denoted *Rigid Policy*, and the level of resource utilization (amount of CPU/cores used). Intuitively and regardless of each application load demand, we have considered that a same ratio of resources usage is assigned to each one of them. Thus, it can use up to 1 out 3 resources, equivalent to 16 cores.

On Figure 5, we have plotted the amount of CPU-cores used by each one of the applications over time. The graphs shows that the system manages to guarantee that none of the applications uses more resources than the amount allowed. For instance, even if $App1$ has high demand, it won't use more than 16 cores. On the contrary, we can observe that some applications, namely $App2$ and $App3$, use less resources than allowed. In this situation, we can expect such a result (Chakode et al., 2010). During the experiment, the maximum of resources used by all the applications is 27 cores, against an average of 15 cores used over time, for a total duration of more than 35.000 seconds.

[2]https://www.grid5000.fr

[3]http://www.cs.huji.ac.il/labs/parallel/workload/l_sharcnet/index.html

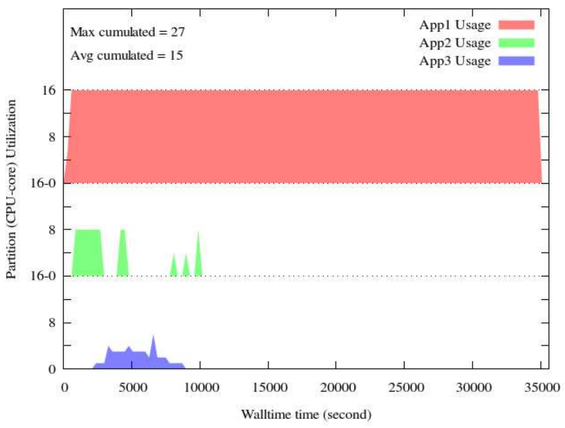

Figure 5: Plot of time versus the amount of CPU-cores used when the *Rigid Policy* is enforced. All of jobs related to a given application can not use more than 16 cores.

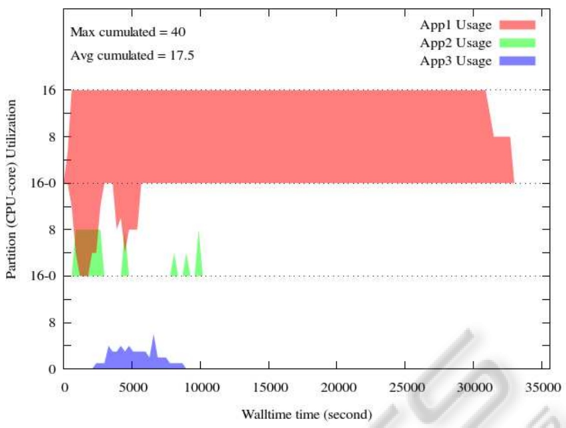

Figure 6: Plot of time versus the amount of CPU-cores used when the *Flexible Policy* is enforced. The long-term jobs related to an given application can not used more than 16 cores.

6.4 Benefit of the Flexibility: Stay Fair while Improving System Utilization

In a second experiment, we have evaluated the benefit of the *Flexible Policy* that consisted in applying the constraint of share-guaranteed only on long-term jobs. In this experiment, jobs with a duration of more than 30 minutes are considered as long-term jobs. For information, the total duration of such jobs represents more than 80% of the total duration of all jobs.

Figure 6 has shown that this policy permits to improve the utilization of the whole system resources. With a maximum of 40 and an average of 17.5 cores against respectively 27 and 15 cores in the first experiment, the gain is equivalent to about 2.5 cores during the experiment, and thus permits to reduce the total duration to around 32700 seconds against 3500 seconds for the first experiment. Furthermore, executing short-term jobs related to a given application does not significantly delay the starting of the jobs related to other applications. For instance, in the first 10000 seconds, the delay that could have caused the overuse of resources by $App1$ is not perceptible on the starting of jobs related to $App2$ and $App3$.

6.5 Utilization, Reliability, Performance, Robustness

In the previous results, it has appeared that the equal repartition does not reflect the demand of resources by each application. This has led to a low utilization of the available resources. Ideally, in the context of sharing like the one we have considered (see section 2), the software vendors sharing the infrastructure would invest to get ratios of resource usage according to their needs, which would be evaluated and dimensioned suitably. Therefore, in the next two experiments we have considered that each application has a ratio of usage proportional to the number of jobs related to it. This corresponds to respectively assigning the equivalent of 32.5, 9.5 and 6 cores to $App1$, $App2$ and $App3$. Additionally, in order to evaluate the behavior of the resource manager facing high loads, we have also considered that all of the 96 jobs have been submitted consecutively without any delay in the system.

On Figure 7, the graphs have shown that the system is still able to enforce the selected policy. We also can observe that with a suitable dimensioning of ratios, the utilization of resource is significantly improved with an average of 27 cores and another one of 34.5 cores used during the experiments, against 15 and 17.5 cores in the first two experiments. Therefore, this has led to reducing the time required to complete all jobs.

We have observed that the resource manager has successfully efficient upon high loads. Indeed, in both experiments, all of jobs submitted have been successfully handled and the associated jobs have been added in suitable queues. The system has required less than three seconds to handle the 96 jobs and add them to the queues.

7 CONCLUSIONS

In this paper, we have designed a resource manager to deal with the implementation of software-as-a-service platform over a HPC cluster. We have first studied the design and proposed an architecture that relies on ex-

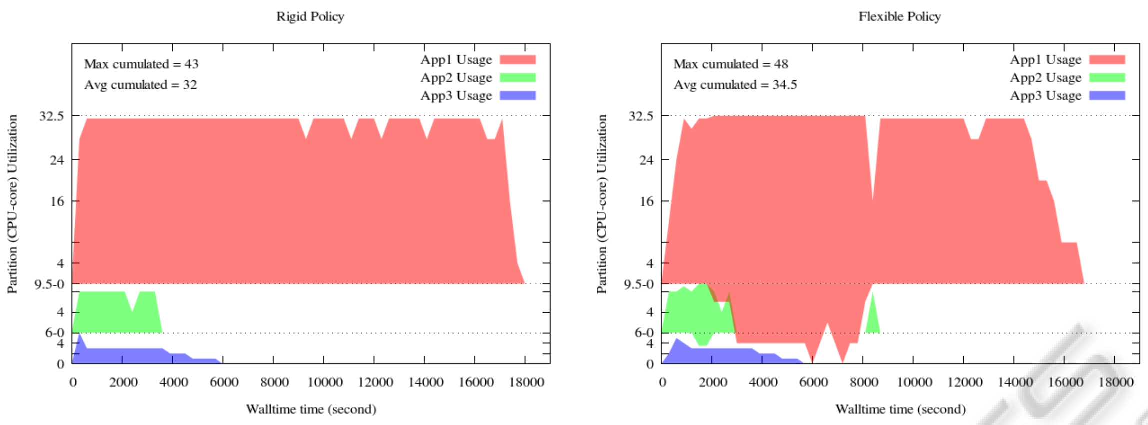

Figure 7: Plot of time versus the amount of CPUs used when the ratio of resource usage granted to each application is related to its demand.

ecuting the jobs related to service requests onto virtual machines. The proposed system relies on OpenNebula to deal with virtual machines life-cycle management. We have then considered a case in which the cluster is shared among a limited number of applications owned by distinct software providers. We have proposed policies and algorithms to scheduling jobs so to fairly allocate resources regarding the applications, while improving the utilization of the cluster resources. We have implemented a prototype that has been evaluated. Experimental results have shown the ability of the system to achieve the expected goals, while being reliable, robust and efficient. We think that the proposed framework would be useful for both industries and academics interested in SaaS platforms. It can be also easily extended to support the implementation of Platform-as-a-Service (PaaS) clouds.

After the prototype stage, some features should be either improved or added in the proposed resource manager before releasing a production-ready version. For instance, since it currently only relies on the OpenNebula's XML-RPC authentification mechanism, this resource manager would need a strong credential mechanism in order to be more secure. Furthermore, since it only supports single virtual machine-based applications, it should be extended in order to support distributed applications. We intend to deal with these issues in our future work.

REFERENCES

Adobe PDF Online. http://createpdf.adobe.com/.

Enomaly Home. http://www.enomaly.com.

Force.com. http://www.salesforce.com/platform/.

Google Apps. http://www.google.com/apps.

Intacct Home. http://www.intacct.com/.

VMware Home. http://www.vmware.com/.

AMD (2005). Amd64 virtualization codenamed asia pacific technology: Secure virtual machine architecture reference manual. (Publication No. 33047, Revision 3.01).

Borja, S., Kate, K., Ian, F., and Tim, F. (2007). Enabling cost-effective resource leases with virtual machines. In *Hot Topics session in ACM/IEEE International Symposium on High Performance Distributed Computing*.

Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P., and Richard, O. (2005). A batch scheduler with high level components. In *Cluster computing and Grid*.

Chakode, R., Méhaut, J.-F., and Charlet, F. (2010). High Performance Computing on Demand: Sharing and Mutualization of Clusters. In *Proceedings of the 24th IEEE International conference on Advanced Information Networking and Applications*, pages 126–133.

Gene K. Landy, A. J. M. (2008). *The IT / Digital Legal Companion: A Comprehensive Business Guide to Software, IT, Internet, Media and IP Law*, pages 351–374. Burlington: Elsevier.

H. Feng, V. M. and Rubenstein, D. (2007). PBS: a unified priority-based scheduler. In *SIGMETRICS*, pages 203–214.

Intel Corporation (2006). Intel Virtualization Technology. *Intel Technology Journal*, 10(3).

Jackson, D. B., Snell, Q., and Clement, M. J. (2001). Core algorithms of the maui scheduler. In *Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer-Verlag.

Jone, T. Linux virtualization and pci passthrough. http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/.

Kay, J. and Lauder, P. (1988). A fair share scheduler. *Commun. ACM*, 31(1):44–55.

Keahey, K., Foster, I., Freeman, T., and Zhang, X. (2005). Virtual workspaces: Achieving quality of service and quality of life in the grid. *Sci. Program.*, 13:265–275.

Lawson, B. G. and Smirni, E. (2002). Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In *In Job Scheduling Strategies for Parallel Processing*, pages 72–87. Springer-Verlag.

Li, L. and Franks, G. (2009). Performance modeling of systems using fair share scheduling with layered queueing networks. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems. MASCOTS '09. IEEE International Symposium on*, pages 1 –10.

Mergen, M. F., Uhlig, V., Krieger, O., and Xenidis, J. (2006). Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. (2009). The Eucalyptus open-source cloud-computing system. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, volume 0, pages 124–131. IEEE.

Sotomayor, B., Montero, R. S., and Foster, I. (2009a). An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds. *Preprint ANL/MCS-P1649-0709*, 13.

Sotomayor, B., Montero, R. S., Llorente, I. M., and Foster, I. (2009b). Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13:14–22.

Turner, M., Budgen, D., and Brereton, P. (2003). Turning Software into a Service. *Computer*, 36(10):38–44.

Vaquero, L. M., Rodero-M., L., Caceres, J., and Lindner, M. (2009). A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55.

Yu, W. and Vetter, J. S. (2008). Xen-Based HPC: A Parallel I/O Perspective. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:154–161.