

Integrated policy management framework for IaaS Cloud middleware

Mauro Canuto · Jordi Guitart

Received: 17 June 2013 / Accepted: 8 March 2014 / Published online: 23 March 2014
© Springer-Verlag Wien 2014

Abstract As a result of the rapid growth of Cloud Computing, several Cloud middleware solutions for the creation and automated management of virtual appliances have been released. Generally, these solutions offer some predefined scheduling policies to manage the provider infrastructure, but additional tuning of the policies is often needed to fully align their behavior with the provider interests. However, current middleware solutions do not offer ways to do this without stopping and recompiling the middleware. This paper proposes a solution that separates scheduling policies from the managers that interpret them, to allow the behavior of the management system to be changed without re-coding the managers. In this way, the middleware can adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting down the system. We propose a new policy language for the definition of management policies and we enable the EMOTIVE Cloud middleware to use these policies by integrating in the middleware the needed policy management framework for parsing and generating code on demand. We demonstrate with real experiments that our policy management framework mimics the expressiveness of scheduling policies in real Cloud middleware and provides more expressiveness if needed. The overhead of the policy management framework is low, but its performance degrades, especially in large datacenters, due to the low scalability of the EMOTIVE monitoring solution.

Keywords Cloud · IaaS · Middleware · Policy Manager · VM scheduling

Mathematics Subject Classification 68N20 · 68M14 · 68M20

M. Canuto · J. Guitart (✉)
Barcelona Supercomputing Center and Technical University of Catalonia,
08034 Barcelona, Spain
e-mail: jguitart@ac.upc.edu

M. Canuto
e-mail: mcanuto@ac.upc.edu

1 Introduction

Cloud computing is becoming more and more important in IT because of its new levels of efficiency, flexibility and cost saving, particularly in the Infrastructure-as-a-Service (IaaS) area. Providers offer virtual appliances to their customers, which transparently share the provider physical resources. This allows the provider reducing costs and maintenance by avoiding the expensive dedicated physical machine per service. Isolated execution of different virtual appliances in a single host is enabled by means of virtualization technology, which allows also monitoring their execution as well as dynamically managing their allocated resources. This brings new chances for resource management in Cloud providers.

As a result of this, a number of Cloud IaaS middleware have been released, such as OpenNebula [1], OpenStack [2], and Eucalyptus [3]. They allow the creation of virtual appliances on demand, generally supporting different virtualization technologies, while including also advanced features for the automated management of those virtual appliances, such as monitoring, resource and network management, accounting, and security.

According to this, it is common that current Cloud providers use these middleware to automate the management of their local infrastructure. Whereas the standard scheduling policies coming in those middleware were generally enough to fulfill the needs of the average Cloud provider, they are not well aligned with the new requirements of current Cloud providers, which aim to include parameters such as the economic profit, the ecological impact, or the client satisfaction in their management policies. For this reason, Cloud providers aim for middleware solutions that can be easily configured to fulfill their goals. This is also desirable for Cloud experimenters and researchers, who can benefit from flexible middleware to reduce the time needed to test new policies. However, this is generally not possible with current middleware, which present the following limitations:

- The middleware does not offer mechanisms to change the scheduling policies. In this case, modifications require the provider to stop the middleware, modify its source code, recompile it, and deploy the middleware again. In addition, it is common that the modification of the source code requires a deep understanding about the internal operation of the middleware.
- The middleware allows selecting at runtime the management policy among some predefined policies. Whereas this allows changing the policy on demand without stopping the middleware, it could happen that none of the pre-configured policies fits with the provider goals. In this situation, we require again the modification of the source code.

Given that situation, this paper aims to expand and evolve the capabilities of Cloud IaaS middleware to overcome the aforementioned limitations. We propose a solution that separates scheduling policies from the managers that interpret them, to allow the behavior of the management system to be changed without re-coding the managers. The middleware can then adapt to changing requirements by disabling policies or replacing old policies with new ones without shutting down the system. According to this, our contributions are as follows:

- We propose a new policy language, so-called LEPIC, for the definition of scheduling policies in Cloud IaaS middleware. This includes the definition of the policy schema, as well as the grammar and the productions needed to provide a semantic and lexical analysis that a policy definition must comply with.
- We have implemented the tools needed to parse and generate code with LEPIC. These tools are built on top of SableCC [4], which is a bottom up parser that takes the approach of using object-oriented methodology for constructing parsers.
- We have implemented a policy management framework to allow the loading and enforcement of LEPIC policies dynamically, without stopping the middleware nor recompiling its whole source code. This framework uses the tools described above to parse policy statements and translate them into sequences of machine language operations which, loaded into memory and executed, carry out the intended computation.
- We have integrated the policy management framework within the EMOTIVE Cloud middleware [5].
- We present, as example, several scheduling policies that can be defined using LEPIC, which mimic and even overcome the expressiveness of policies used in commercial middleware, and we demonstrate how these policies can be loaded and enforced dynamically in the EMOTIVE middleware.

The rest of the paper is structured as follows. Section 2 presents the related work. Section 3 introduces the LEPIC language and the development of tools to parse LEPIC policies. Section 4 describes the development of the policy management framework and how it integrates in the EMOTIVE middleware. Section 5 discusses the needed adaptations to use LEPIC in other Cloud middleware. Section 6 demonstrates the use of LEPIC and evaluates its performance. Finally, Sect. 7 presents the conclusions of the paper and the future work.

2 Related work

This section presents the related work. It introduces first how scheduling is accomplished in commercial Cloud middleware and then presents alternative approaches for policy definition and management.

2.1 Scheduling in commercial Cloud middleware

In general, current commercial Cloud middleware allow selecting at runtime the management policy to schedule Virtual Machines (VM) among some predefined policies. However, using ad hoc policies require stopping the system, modifying the source code, and recompiling. In the following paragraphs, we describe the support for VM scheduling offered by the most popular Cloud middleware.

OpenNebula [1] comes with a match-making scheduler that implements the Rank Scheduling Policy. The goal of this policy is to prioritize those resources more suitable for the VM. Although it is highly modifiable and can be easily replaced by third-party developments, this cannot be done at runtime. The match-making sched-

uler works as follows [6]. First, those hosts that do not meet the VM requirements and do not have enough resources (available CPU and memory) to run the VM are filtered out. Then, the RANK expression is evaluated upon this list using the information gathered by the monitor drivers. Any variable reported by the monitor driver can be included in the RANK expression. It is possible to implement several placement heuristics depending of the RANK expression used, which defines suitable hosts for VMs. Finally, those resources with a higher rank are used first to allocate VMs.

The scheduler comes with some predefined RANK expressions that build three default policies that can be selected, namely *Packing*, which minimizes the number of hosts in use by packing the VMs in the hosts to reduce VM fragmentation, *Striping*, which maximizes the resources available for the VMs by spreading the VMs in the hosts, and *Load-aware*, which maximizes the resources available for the VMs by using those nodes with less load. The user has also the possibility to define a custom RANK expression.

OpenStack [2] uses the *nova-scheduler* service to determine how to dispatch *compute* and *volume* requests. For both types of requests, the scheduler service is configured by default as a Filter Scheduler, which supports filtering and weighting to make informed decisions on where a new instance should be created [7].

When the Filter Scheduler receives a request for a resource, it first applies filters to determine which hosts are eligible for consideration when dispatching a resource. Filters are binary: either a host is accepted by the filter, or it is rejected. In the default configuration, this scheduler will only consider hosts that are in the requested availability zone (*AvailabilityZoneFilter*), that have sufficient RAM available (*RamFilter*), and that are actually capable of servicing the request (*ComputeFilter*).

Then, the scheduler applies one or more cost functions to each resulting host from the filtering, which is multiplied by a configurable weighting constant. The scheduler selects the host that has the minimum weighted cost. In the default configuration, the scheduler will assign a cost to each host based on the amount of free RAM and will multiply each cost value by -1 . This is equivalent to selecting the host with the maximum amount of RAM available.

The Chance Scheduler, which randomly selects from the lists of filtered hosts, can also be configured as scheduler.

CloudStack [8] default allocator automatically places VM instances wherever there is sufficient physical capacity. The chosen host will always be close to where the VM virtual disk image is stored. In addition, the administrator can specify that certain hosts should have a preference for particular types of guest instances [9]. For example, he could state that a host should have a preference to run Windows guests. The host allocator will attempt to place guests of that OS type on such hosts first.

The default allocator allows both vertical allocation, which consumes all the resources of a given host before allocating any guests on a second host, and horizontal allocation, which places a guest on each host in a round-robin fashion. However, it is not clear how these different strategies can be configured [10]. It also allows CPU over-provisioning, as configured by the administrator, which commits more CPU cycles to the allocated guests than are actually available from the hardware [9].

CloudStack also provides a pluggable interface for adding new allocators, and a custom HostAllocator can be written by implementing the *allocateTo* method. However, it is necessary to restart the Management Server to load a custom HostAllocator.

Eucalyptus [3] allows setting the Cluster Controller scheduling policy through the *SCHEDPOLICY* option in the Eucalyptus configuration file. Currently, this option can be set to *GREEDY*, which chooses the first node that is found that can run the VM, *ROUNDROBIN*, which selects nodes one after another until one is found that can run the VM, and *POWERSAVE*, which puts nodes to sleep when they are not running VMs, and reawakens them when new resources are required. VMs will be placed on the first awake machine, followed by machines that are asleep.

Ganeti [11] requires the administrator to specify the exact node location for instances. This prevents a completely automatic node evacuation, and is in general a nuisance. The iallocator framework [12] enables the automatic placement via external scripts, eliminating the need to manually specify nodes in instance add, instance moves, node evacuate, etc. At configure time, the list of the directories can be selected via the *-with-iallocator-search-path=LIST* option, where LIST is a comma-separated list of directories. If not given, this defaults to *\$libdir/ganeti/iallocators*. Ganeti will then search for allocator script in the configured list, using the first one whose filename matches the one given by the user.

The protocol for communication between Ganeti and an allocator script is the following. Ganeti launches the script with a single argument, a filename that contains a JSON-encoded structure (the input message). If the script finishes with exit code different from zero, it is considered a general failure and the full output will be reported to the users; this can be the case when the allocator cannot parse the input message. If the allocator finishes with exit code zero, it is expected to output (on its stdout) a JSON-encoded structure (the response).

Finally, **EMOTIVE** [5], which is the middleware used in this paper (more details can be found in Sect. 4.1), uses by default a simplistic Round Robin policy (so-called SimpleScheduler policy). This policy can be replaced if needed, but any change in the scheduling policy requires to stop the system, recompile and execute again.

2.2 Policy definition and management

Policies are rules that govern the behavior of a system. Their management defines what actions need to be carried out when specific events occur within a system or what resources must be allocated under specific conditions.

There is considerable interest in the use of policies for security and management of large-scale networks and distributed services. However, there has been very little work on how to disseminate policies to the entities that will interpret them and how to deal with dynamic large-scale environments where policies need to be updated to cater for changing requirements.

These works include Ponder and Rei. Ponder [13], developed at the Imperial College of London, is a declarative, object-oriented language for specifying security policies with role-based access control and, unlike many other policy notations, it supports typed policy specifications. It also has a number of supporting abstractions that are

used to define policies, such as domains for hierarchically grouping managed objects and events for triggering obligation. Ponder incorporates an awareness of events and policies and implements a policy execution framework but it does not include an interaction protocol with underlying layers, which is necessary for policies running in a Cloud platform like EMOTIVE, interacting mainly with the scheduling layers. Moreover, its authorization policies permit or deny actions based upon the action, the source of the action and the target of the action, specifying rules that prevent subjects from performing actions that could be harmful for them or their domain. LEPIC instead, does not provide security policies for task execution but it allows an administrator to trigger operations adapting the system status to its needs.

Rei [14], a project at the University of Maryland Baltimore County, includes meta policy specifications for conflict resolution, speech acts for remote policy management and policy analysis specifications like what-if analysis and use-case management, making it a suitable candidate for adaptable security in the environments under consideration. Its engine, developed in XSB, reasons over Rei policies and domain knowledge in RDF and OWL to provide answers about the current permissions and obligations of an entity, which are used to guide the entity behavior. As Rei is geared towards distributed environments, it also includes conflict resolution specifications like modality preferences or priority assignments between policies or between individual rules of a policy. However, this language has different focus compared to LEPIC. Rei policies are used to guide the behavior of entities in the policy domain while LEPIC provides a more flexible environment that involves several layers and gives the opportunity to deal with external resources.

Since EMOTIVE policies deal more with VMs placement and resources utilization than security or domain aspects, Ponder and Rei analysis helped us to understand and to define some features which were not clear at the beginning such as parameterized types and providing the user with a schema to follow in order to create new scheduling operations.

An alternative strategy could be modeling a policy using a W3C XML schema and trying to load it dynamically using the JAXB APIs [15]. However, this approach lacks on flexibility when we want to define different policies with specific syntax. Developing a new language which can be parsed on demand provides us with this required flexibility.

3 Language for EMOTIVE policies integration and creation (LEPIC)

This section describes the new language that we propose for the definition of placement and scheduling policies in IaaS Cloud solutions, and how we have implemented the needed parser and compiler to work with this language.

3.1 Language schema

Our language allows writing policies according with the schema shown below (editable parts in upper case). Note that a policy can be defined only with `action` parts, `poll` parts, or both.

```

policy POLICY_NAME;
type action (operation OPERATION) {
  VARIABLES_DECLARATION;
  CODE;
  return RETURN_STATEMENT;
}

type poll POLL_NAME (time TIME) {
  VARIABLES_DECLARATION;
  CODE;
}

```

POLICY_NAME stands for the name to be assigned to the policy.

When type is *action*, the policy code (VARIABLES_DECLARATION + CODE) will be executed when the corresponding OPERATION takes place. Currently, we can define OPERATION as *create*, which stands for the creation of a virtual machine, and as *destroy*, which stands for the destruction of a VM. RETURN_STATEMENT can return values to be used in the corresponding operation. For instance, we can return to the creation operation the identifier of the physical host where this VM should be created.

When type is *poll* the policy code will be executed periodically according to the specified time interval TIME (default 500 ms). This kind of policy can be easily used to monitor system parameters and trigger actions when some conditions are satisfied. For instance, migrate a VM when the load of a physical host exceeds a threshold. Note that we can define multiple polling parts with different intervals in the same policy.

VARIABLES_DECLARATION follows a C-like syntax and the initialization can also be done in this section. The following types are supported in LEPIC:

- void
- char: initialization using single quotes (e.g. `char var = 'a';`)
- int (e.g. `int num = 0;`)
- boolean: it can have *true* or *false* value (e.g. `boolean b = false;`)
- float (e.g. `float fnum = 1.0;`)
- string: initialization using double quotes (e.g. `string vm = "id";`)
- uri: initialization using double quotes and it has to be an http URI (e.g. `URI node = "http://localhost";`)

CODE section can be programmed using C-like statements. They are listed in the following lines with some examples.

- **Variable assignment:** `var = value;`
- **Expressions:** `var = 5 + 10;`
- **Conditionals¹:**

```

if (condition) {           if (condition) {
  statement;               statement;
}                           }
                           else {
                           statement;
                           }

```

¹ Unlike C language, braces must be always present even if there is only one statement.

– Loops:

```
while (condition) {      for (init; condition; incr) {
    statement;           statement;
}
```

Note that if a variable is initialized in the `VARIABLES_DECLARATION` section, that value will be assigned to the variable only the first time the policy is loaded. Contrariwise, if we need that a variable must have an initial value every time the policy runs, we have to initialize it in the `CODE` section using a *variable assignment*.

`CODE` section can include also invocations to functions that allow interacting with the underlying Cloud middleware, for instance, to retrieve monitoring information or to trigger the execution of actions. This is implemented by programming Java wrappers that invoke the methods of the middleware API. Note that the specific list of functions available will depend on the underlying middleware. Section 4.4 describes the functions available in the LEPIC implementation for the EMOTIVE middleware.

3.2 Creating a compiler for LEPIC language

In order to parse and to have the input represented in the form of an abstract syntax tree that defines the order of operations and allows us to traverse the different parts of the expression individually, SableCC [4], which is a Java-based tool that performs the same job as Flex/Bison, has been used. It is a bottom-up parser, which takes an unconventional and interesting approach of using object oriented methodology for constructing compilers. SableCC also keeps a clean separation between machine-generated and user-written code which leads to a shorter development cycle and results in easy to maintain code for the generated compiler. The standard output from SableCC is Java code. Thus, the compiler can be easily incorporated into the middleware source code.

Producing a compiler using SableCC requires the following steps:

1. Creation of a SableCC specification file containing the lexical definitions and the grammar of the language to be compiled. This file includes the definition of helpers, which are like macros that when found in a regular expression are replaced semantically (not textually) with its declared character set or regular expression, language tokens, and grammar productions. The complete list of LEPIC helpers, tokens, and productions can be found in [16].
2. Launch of SableCC on the specification file to generate a framework.
3. Creation one or more working classes possibly inheriting from classes generated by SableCC. In our case, this includes:
 - A stack-based class that does a depth first traversal of the Abstract Syntax Tree (AST) and translates assignment and operation statements by pushing all the factors on a stack and popping the results into variables. All the variables are stored in a HashMap structure used to save or retrieve values later on. This class also handles conditional and loop statements and function invocations.

- A class that ensures that the statements defined in a LEPIC policy are grammatically correct via a depth first traversal of the AST, while storing information about identifiers in a symbol table and checking for errors.
 - A class to check if two operands have valid types to perform an assignment or arithmetical operation.
 - The class that implements the wrappers of the methods of the underlying middleware API that allow retrieving monitoring information or triggering the execution of actions from the policy code.
4. Creation of a main compiler class that activates lexer, parser, and working classes.
 5. Compilation of the compiler.

4 LEPIC-based policy management framework in EMOTIVE

This section describes the policy management framework supporting LEPIC that we have built within the EMOTIVE Cloud middleware.

4.1 EMOTIVE architecture

This section introduces Elastic Management of Tasks in Virtualized Environments (EMOTIVE) [5], the Barcelona Supercomputing Center (BSC) and BarcelonaTech (UPC) IaaS open-source solution for Cloud Computing. EMOTIVE provides users with elastic fully customized virtual environments in which to execute their applications. Further, it simplifies the development of new middleware services for managing Cloud systems by supporting resource allocation and monitoring, data management, live migration, and checkpoints.

As shown in Fig. 1, EMOTIVE Cloud is mainly composed by three different layers: *Virtualization Fabrics*, *Virtual Machine Manager*, and *Virtual Machine Scheduler*. Note that main layers involved in this paper appear emphasized.

The *Virtualization Fabrics* layer comprises the physical resources where the VM will run. This layer wraps the virtualized resources and offers them to the upper layers. Furthermore, it implements a distributed shared file system (DFS) that supports efficient VM creation, migration, and checkpointing.

The *Virtual Machine Manager* layer is in charge of creating and maintaining the whole virtual machine life cycle (create, destroy, migrate, etc.). VMs are created on demand, according to the application requirements, both hardware (CPU type, amount of resources required) and software (required packages). In addition, this layer comprises all the local resource management decisions (i.e. in a single host) and the status monitoring capabilities.

Finally, the *Virtual Machine Scheduler* layer comprises all the global VM placement decisions, both among different providers in a Cloud federation and different hosts in a single provider. This layer is in charge of deciding where a VM will be executed and managing its location during the execution (e.g. migration of VMs across provider hosts, cancellation of VMs, resumption of VM execution from a checkpoint upon hardware failure, etc.).

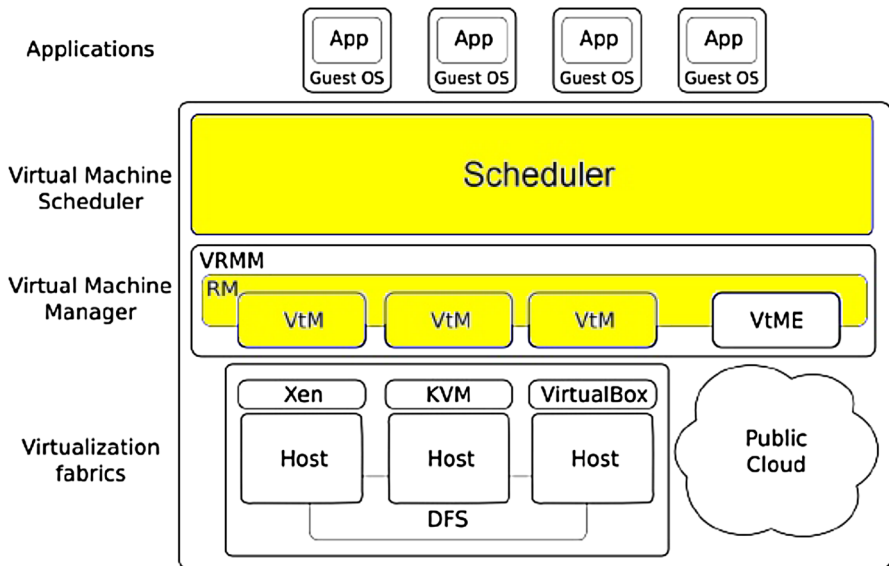


Fig. 1 EMOTIVE architecture

Moreover, this framework allows multiple schedulers with different policies and capabilities such as machine learning, prediction, economic, fault tolerance, semantic description, or SLA enforcement. This is achieved thanks to the usage of a common interface that allows developing new schedulers with different features and policies. In particular, we encourage using the Open Cloud Computing Interface (OCCI) [17], which allows EMOTIVE to be interoperable with other Cloud middleware supporting this interface. Nevertheless, the scheduling policy to be used has to be decided before the middleware is compiled and executed. By default, EMOTIVE uses a simplistic Round Robin policy (so-called SimpleScheduler policy). Any change in the scheduling policy requires stopping the system, recompiling, and executing again.

4.2 EMOTIVE API extension

To allow the EMOTIVE scheduling policy to be replaced on demand, we have extended the EMOTIVE API with a method to upload a new policy. In addition, we have updated the standard EMOTIVE command-line client to allow the invocation of this new method. According to this, we can change the EMOTIVE scheduling policy by typing `policy PolicyFile`. Of course, it is possible to have the system back to the default scheduler just using the command `policy SimpleScheduler`.

The client-side application constructs an outgoing multipart/form-data message containing the policy and sends it to the EMOTIVE server side. Upon reception of a request to change the policy, EMOTIVE creates and writes a new file containing its definition (*PATH: /policy/PolicyFile*). Then, the new Policy Manager component

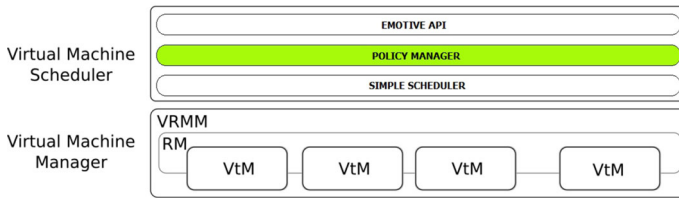


Fig. 2 Updated virtual machine scheduler layer

takes the responsibility to parse the policy file for the first time (storing variables in a `HashMap`) and set the new policy to be used from now on. We describe this process in the following section.

4.3 Policy Manager

As shown in Fig. 2, we have modified the *Virtual Machine Scheduler* layer in *EMOTIVE* to include a *Policy Manager* component, which allows loading different scheduling policies on demand without requiring to stop the middleware or recompile the source code. To this end, it implements the LEPIC language support (loading, parsing, and compiling) and enables the interaction of LEPIC policies with the underlying *Virtual Machine Manager* layer.

In addition, *Policy Manager* overrides the functionality of the original *SimpleScheduler*, so it is responsible for managing all the scheduling operations. In fact, it extends the *Scheduler* class. In this way, if an operation is not defined in the new loaded policy the default one will be used, avoiding errors or system crashes.

When a new policy is uploaded for the first time, *Policy Manager* initiates the parsing process and checks for semantic errors (using the classes part of the LEPIC framework described in Sect. 3.2), differentiating the several *action* and *poll* parts that comprise the policy.

For *action* parts, the compiler checks what operations (i.e. create and/or destroy) have an associated code that must be executed when these operations are invoked. Later on, when any of these operations is invoked, *Policy Manager* will parse the corresponding code and compute the variable values of the policy in order to determine its return value. For create operations, we define this return value to be the URI of the target node where to execute the newly created VM. For destroy operations, there is not return value by default. In this case, the action part is used to trigger the scheduling policy when a VM is destroyed. However, for the sake of demonstrating the possibilities of action parts, we added a new destroy operation without parameters that delegates to the middleware the selection of the VM to be destroyed. In this case, the return value is the ID of the VM to be destroyed.

For *poll* parts, *Policy Manager* creates a thread pool, having one thread for each polling part in the policy. Each thread will periodically (according to the defined time interval) parse the corresponding code, compute the variable values and do the required actions. When a new policy is loaded, all the running threads which were performing polling operations are stopped.

Table 1 EMOTIVE functionalities available for LEPIC policies

Function	Returned value
int size();	Number of active nodes in the private Cloud
URI getURI(int i);	URI of the node i
int numVMnode(int/URI i);	Number of VMs running in the node i
float freeCPU(int/URI i);	Percentage of free CPU in the node i
void destroyVM(string s);	Destroy the VM whose Id is s . It does not return any value
boolean exists(string id);	True if the node or the VM passed as parameter exists
int numVMs();	The number of VMs running in the private Cloud
string getVMId(int i);	VM Id whose index in the domains list is i
float getVMCPU(int/string i);	Amount of used CPU by the VM whose Id or index is i
float getVMCPUnode(int/URI i, int/string j);	Amount of used CPU by the VM whose Id or index is j in the node i
int getVMMem(int/string i);	Amount of memory needed by the VM whose Id or index is i
int freeMem(int/URI i);	Free memory in the node whose URI or index in the list of nodes is i

4.4 Access to EMOTIVE functionalities from LEPIC policies

As mentioned before, *Policy Manager* provides Java wrappers that allow using some methods of the underlying middleware API within LEPIC policies. In particular, this allows retrieving monitoring information that can be used to check conditions or triggering the execution of actions from the policy code. The methods currently available are listed in Table 1. Note that this list can be extended when new functionalities are required in the policies or new methods are added in the middleware API.

5 Porting LEPIC policy management framework to other middleware

Although our LEPIC-based policy management framework was originally designed to be used in the EMOTIVE Cloud middleware, its modular design makes it easily portable to other Cloud middleware. In fact, the language schema and the classes needed to analyze, parse and compile the policy files are independent of the underlying middleware. According to this, only some adaptations to port the Policy Manager are required to use LEPIC in other middleware. These adaptations are as follows:

1. Extend the API of the middleware to offer a new method to load scheduling policies on demand.
2. Integrate *Policy Manager*, which is available as a Java library, within the middleware to enable the parsing of LEPIC policies when they are loaded and the execution of policy code when corresponding operations are invoked and defined intervals elapse.
3. Implement Java wrappers that allow the middleware functionalities to be invoked from the policies code.

Table 2 Specifications of the experimental testbed

	pctomeu	pctinet	cloud00
Kernel	Linux 2.6.18-xen	Linux 2.6.18.8-xen	Linux 2.6.18.8-xen
Processor	Intel Xeon CPU X5355	Intel Xeon CPU 5160	Intel Xeon CPU E5440
Clock speed (GHz)	2.66	3.00	2.83
Number of processors	4	4	8
RAM (Gb)	16	16	16
Libvirt library	libvirt 0.7.6	libvirt 0.8.3	libvirt 0.8.4
Hypervisor	Xen 3.1.0	Xen 3.3.0	Xen 3.3.0

6 Evaluation

In this section, we show how LEPIC can be used to write scheduling policies for Cloud IaaS middleware. First, we use LEPIC to write the same scheduling policies used in OpenNebula. Then, we demonstrate LEPIC advanced functionalities, such as polling policies and policies combining actions and polling.

6.1 Experimental setup

Case studies and experimental tests have been conducted in a cluster consisting of three physical hosts, so-called *pctomeu*, *pctinet*, and *cloud00*, running with the Xen virtualization hypervisor. Table 2 describes the characteristics of these hosts.

Unless otherwise specified, VMs are created with four virtual CPUs and 256 Mb RAM. To generate CPU load on the VMs, we have used the *stress* workload generator [18]. Note that, according to our notation, CPU utilizations are quantified using the typical Linux CPU usage metric (i.e. for a computer with 4 CPUs, the maximum utilization of CPU will be 400 %).

6.2 Case studies

6.2.1 Packing policy

When using the packing policy, new VMs will be created in the node with more VMs running. This can be useful to reduce the number of active nodes in the system. The definition of this policy using LEPIC language is shown in Listing 1.

Figure 3 shows the placement of nine identical VMs over time in our testbed infrastructure when using the Packing policy. Note that initially all VMs go to *cloud00* until the maximum number of VMs per node (i.e. 3) is reached. At this point, VMs are placed in *pctinet*. Similarly, when this node becomes full, VMs go to *pctomeu*. Note that any subsequent VM reaching the provider when these three nodes are full will be rejected.

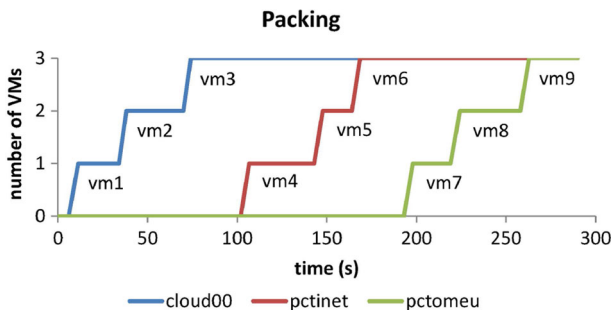
Listing 1 Packing policy with LEPIC

```

/* use node with more VMs running first */
policy policy_packing;
type action (operation create) {
    URI nodeToSelect;
    int i, node, num, size, max;
    int maxVMnode = 2;

    max = -1;
    node = -1;
    size = size();
    for (i=0; i < size; i++) {
        num = numVMnode(i);
        if (num > max) {
            if (num < maxVMnode) {
                max = num;
                node = i;
            }
        }
    }
    if (node != -1) {
        nodeToSelect = getURI(node);
    }
    else {
        nodeToSelect = null;
    }
    return nodeToSelect;
}

```

**Fig. 3** VMs placement when using packing policy

6.2.2 Striping policy

When using the striping policy, nodes with less VMs running are used first. This allows maximizing the resources available to VMs in a node. The definition of this policy using LEPIC language is shown in Listing 2.

Figure 4 shows the placement of nine identical VMs over time in our testbed infrastructure when using the Striping policy. Note that the VMs are being placed

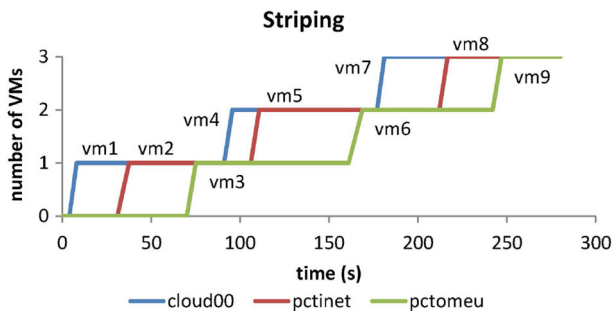
Listing 2 Striping policy with LEPIC

```

/* use node with less VMs running first */
policy policy_striping;
type action (operation create) {
    URI nodeToSelect;
    int i,node,num,size,min;
    int maxVMnode = 3;

    min = 100;
    node = -1;
    size = size();
    for (i=0; i < size; i++) {
        num = numVMnode(i);
        if (num < min) {
            if (num < maxVMnode) {
                min = num;
                node = i;
            }
        }
    }
    if (node != -1) {
        nodeToSelect = getURI(node);
    }
    else {
        nodeToSelect = null;
    }
    return nodeToSelect;
}

```

**Fig. 4** VMs placement when using striping policy

in the three nodes in a Round-robin fashion. Again, any subsequent VM reaching the provider when these three nodes are full will be rejected.

6.2.3 Load-aware policy

The Load-aware policy creates VMs in the node with less CPU load first. Listing 3 shows the definition of this policy using LEPIC language. Note how we use the *freeCPU()* function to get information about the amount of free CPU on each node.

Listing 3 Load-aware policy with LEPIC

```

/* use node with less CPU load first */
policy policy_load_cpu;
type action (operation create) {
    URI nodeToSelect;
    int i,node,num,size;
    float max,cpu;
    int maxVMnode = 3;

    max = 0.0;
    node = -1;
    size = size();
    for (i=0; i < size; i++) {
        cpu = freeCPU(i);
        num = numVMnode(i);
        if (cpu > max) {
            if (num < maxVMnode) {
                max = cpu;
                node = i;
            }
        }
    }
    if (node != -1) {
        nodeToSelect = getURI(node);
    }
    else {
        nodeToSelect = null;
    }
    return nodeToSelect;
}

```

Figure 5 shows the placement of nine identical VMs over time in our testbed infrastructure when using the Load-aware policy. This figure also provides information about the amount of free CPU on each node and the CPU load of each VM executed there. Initially, vm1 and vm2 go to *cloud00* because the amount of free CPU in that node is higher, but when we stress vm2 this reduces the free CPU in that host, and vm3 must go to *pctinet*. The same criteria is used to place the rest of VMs. Note that sometimes the difference between the amount of free CPU of several nodes can be very small (for instance, when placing vm4 and vm7 both *pctinet* and *pctomeu* are almost totally available) but even in those situations the node with more free CPU is chosen. The placement of vm9 deserves special attention because, according to the amount of free CPU in the nodes, it should be placed in *cloud00*. However, as this node (and node *pctomeu* too) has already the maximum number of VMs running, vm9 is placed in *pctinet*.

6.2.4 Policy with polling parts

In order to demonstrate how a polling scheduling operation works, we define a policy that destroys a VM if it requires more than 768 MB of memory. This policy checks every 20,000 ms the memory required by every VM present in the system. If this value

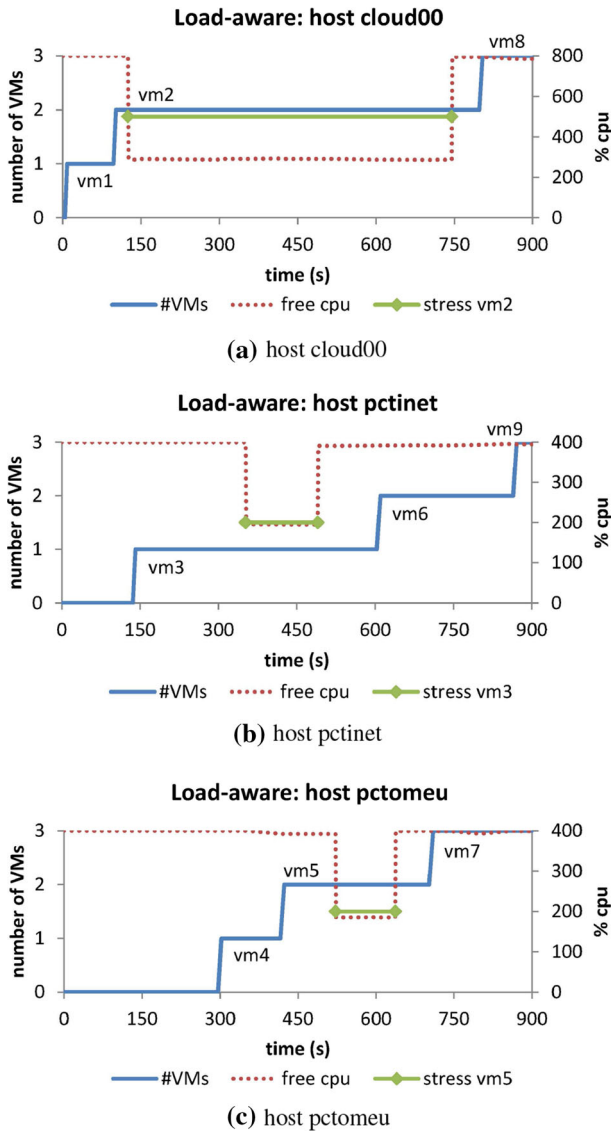


Fig. 5 VMs placement when using Load-aware policy

is greater than 768, the corresponding VM is destroyed. This policy can be useful, for instance, to reduce the amount of memory used in a node when it is running out of it. The definition of this policy using LEPIC language is shown in Listing 4.

Figure 6 shows the placement over time of several VMs with different requirements of memory in our testbed infrastructure when using the Polling policy described above. This figure also provides information about the memory requirement of each VM, the limit for the maximum amount of memory which can be required by a VM (i.e. 768 Gb), and the polling interval used by our policy to check the memory of each VM (i.e. 20 s).

Listing 4 Polling policy with LEPIC

```

policy monitoring_mem_policy;
type poll CheckMem (time 20{,}000) {
    int i, n, mem;
    string vm;

    n = numVMs();
    for (i=0; i < n; i++) {
        mem = getVMMem(i);
        if (mem > 768) {
            vm = getVMId(i);
            destroyVM(vm);
        }
    }
}

```

Note that the placement of VMs is decided by using the default Round-robin policy coming with EMOTIVE. As shown in the figure, vm4 is destroyed when the polling policy detects at time 77 that it requires 1,024 Gb. Similarly, vm6 is also destroyed at time 97 due to the same reason.

6.2.5 Combined policy

This policy combines both action and polling parts. The action part applies the Load-aware policy to decide the placement of a VM when it is created. The poll part checks every 40,000 ms the amount of free CPU on each node and if it is lower than 10 %, the VM with higher CPU load in that node will be destroyed. The definition of this policy using LEPIC language is shown in Listing 5.

Figure 7 shows the placement of several identical VMs over time in our testbed infrastructure when using the Combined policy described above. This figure also provides information about the amount of free CPU on each node, the CPU load of each VM executed there, the limit for the minimum amount of free CPU that we require on each node (i.e. 10 %), and the polling interval used by our policy to check the free CPU on each node (i.e. 40 s).

Initially, vm1, vm2, and vm3 go to *cloud00* because the amount of free CPU in that node is higher, but when we stress vm3 this reduces the free CPU in that host, and vm4 must go to *pctinet*. Similarly, vm5 must go to *pctomeu*, because at that time we are also stressing vm4 at *pctinet*.

In the background, the polling part of the policy is monitoring the amount of free CPU on each node. After stressing vm1, the policy realizes at time 86 that free CPU on *cloud00* is below the 10 % threshold. According to this, it checks the CPU load of each VM running on that node and destroys the VM with higher CPU load, in this case, vm1. Similarly, at time 250, the free CPU threshold is surpassed again, because vm2 and vm3 are being stressed at the same time. As the CPU load of vm3 is slightly higher, this VM is destroyed.

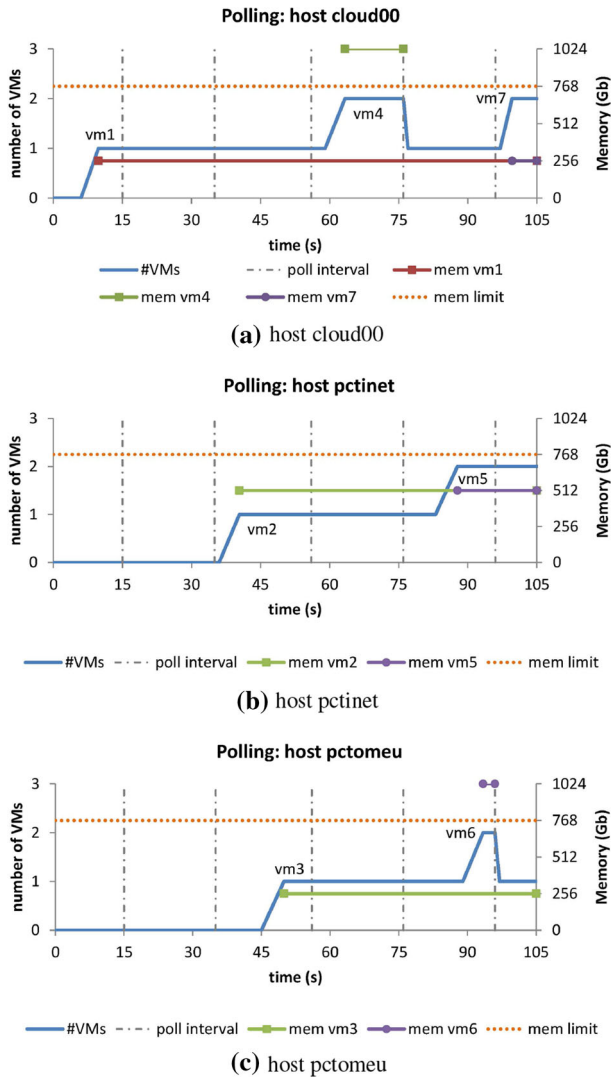


Fig. 6 VMs placement when using a Polling policy

6.3 Performance analysis

This section evaluates the overhead incurred by our policy management framework.

6.3.1 Overhead of action parts

In order to evaluate the overhead of action policies, we measured the average time needed to schedule the creation of a VM in our testbed infrastructure. The experiment consisted of creating ten VMs that are deployed by the scheduler which runs in cloud00

Listing 5 Policy combining action and polling parts

```

policy combined_policy;
/* use node with less CPU load first */
type action (operation create) {
    URI nodeToSelect;
    int i,node,num,size;
    float max,cpu;
    int maxVMnode = 3;

    max = 0.0;
    node = -1;
    size = size();
    for (i=0; i < size; i++) {
        cpu = freeCPU(i);
        num = numVMnode(i);
        if (cpu > max) {
            if (num < maxVMnode) {
                max = cpu;
                node = i;
            }
        }
    }
    if (node != -1) {
        nodeToSelect = getURI(node);
    }
    else {
        nodeToSelect = null;
    }
    return nodeToSelect;
}

/* destroy the VM with higher CPU load
when the node free CPU is lower than 10% */
type poll CheckCPU (time 40{,}000) {
    int i,j,num,size,vmi;
    string vm;
    float cpu,max;

    size = size();
    for (i=0; i < size; i++) {
        cpu = freeCPU(i);
        if (cpu < 10.00) {
            max = 0.0;
            num = numVMnode(i);
            for (j=0; j < num; j++) {
                cpu = getVMCPUNode(i,j);
                if (cpu > max) {
                    max = cpu;
                    vmi = j;
                }
            }
            vm = getVMId(vmi);
            destroyVM(vm);
        }
    }
}

```

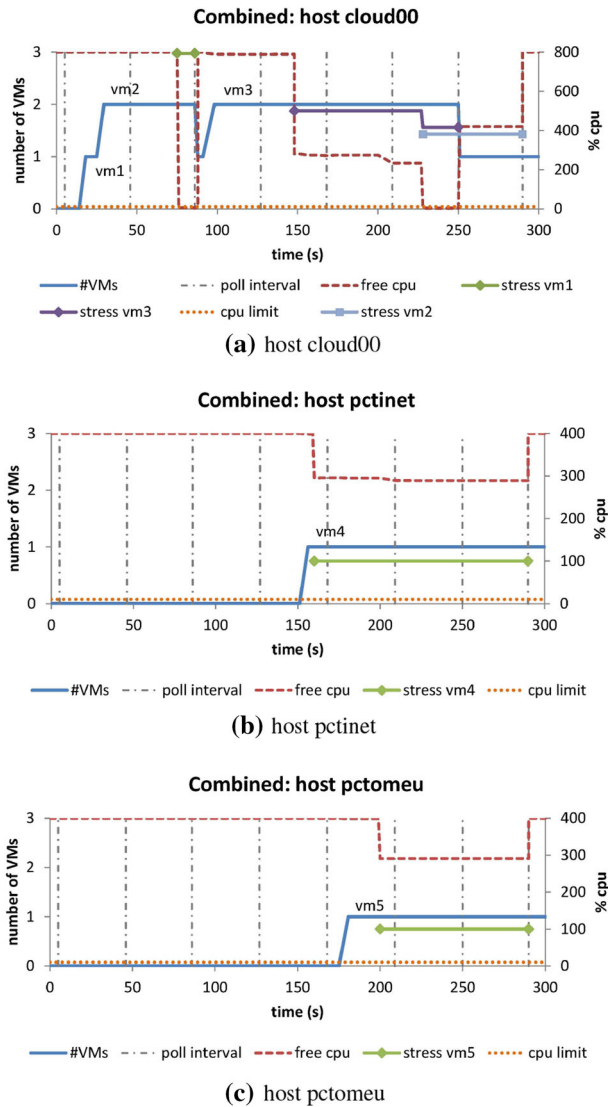


Fig. 7 VMs placement when using combined policy

node. We compared the time needed when EMOTIVE used its default Round-Robin scheduler with the time needed when EMOTIVE used different LEPIC policies and the time required by OpenNebula (version 4.2) with its corresponding policies. Note that to allow a fair comparison between EMOTIVE and OpenNebula, we measured only the time to schedule the creation of the VM, not the time to create the VM entirely.

To evaluate the scalability of our approach, we measured also the impact of the number of nodes in the testbed. This is a key parameter for scalability since all the

Table 3 Average time (in ms) to schedule the creation of a VM with various action policies

	Default RR	LEPIC RR	Packing	Striping	Load-aware
EMOTIVE					
Nodes = 3	25 ± 2	40 ± 4	155 ± 15	161 ± 17	364 ± 36
Nodes = 10	26 ± 2	43 ± 2	568 ± 39	572 ± 49	1,098 ± 85
Nodes = 25	25 ± 3	40 ± 4	1,256 ± 386	1,075 ± 25	2,901 ± 315
Nodes = 50	25 ± 10	41 ± 4	2,526 ± 503	2,764 ± 533	4,851 ± 329
OpenNebula					
Nodes = 3			82 ± 19	68 ± 6	71 ± 8
Nodes = 10			128 ± 72	126 ± 34	107 ± 22
Nodes = 25			270 ± 188	213 ± 21	217 ± 54
Nodes = 50			390 ± 45	416 ± 67	399 ± 52

presented policies iterate over all the nodes in the testbed. Note that as our testbed has only three machines, we made the policies iterate repeatedly over these machines to emulate a bigger testbed.

As shown in Table 3, the default EMOTIVE scheduler and the equivalent Round-Robin (RR) policy defined using LEPIC language obtain close scheduling times, so this means that the overhead of the LEPIC policy management framework is small. The scheduling time when using policies such as Packing, Striping and Load-aware is greater, which is not surprising given that the complexity of these policies is higher. The increment depends on the number of nodes, so for big datacenters it can be significant. Note that this dependence occurs both for LEPIC policies and its OpenNebula counterparts.

However, the scheduling time when using OpenNebula policies is clearly lower than the time required by LEPIC policies, especially when the number of nodes increases. Note that this is not because of an overhead of the LEPIC policy management framework, but due to the low scalability of the EMOTIVE monitoring solution. We have determined that the scheduling time with LEPIC policies is directly related with the number of invocations to the middleware API to get monitoring information or to trigger actions. Recall that each of those invocations requires a REST Web Service communication. This is a good solution for the extensibility and portability of EMOTIVE and it allows the scheduling policies to get the most up-to-date status of the infrastructure, but it has a noticeable impact on its scalability.

On the other side, OpenNebula implements a monitoring subsystem where the Information Manager (IM) driver [19] monitors periodically the status of the system by executing probes on the nodes and stores the information in a database. The OpenNebula scheduler can easily query the system status in the database. Depending on the monitoring interval, it is possible that this status is not totally up-to-date, but the scalability of the monitoring system is clearly better. We strongly believe that implementing such a monitoring solution in EMOTIVE would bring the scheduling time with LEPIC to comparable values to OpenNebula.

Table 4 Average time (in ms) to run a task with various polling policies

Host	Baseline	T = 10,000	T = 1,000	T = 100
		<i>CheckMem</i>		
cloud00	50,300 ± 89	51,248 ± 60	50,998 ± 597	51,608 ± 131
pctinet	62,642 ± 657	64,533 ± 555	64,149 ± 520	64,175 ± 607
pctomeu	80,954 ± 1,928	83,932 ± 1,348	83,294 ± 1,011	83,259 ± 2,356
		<i>CheckCPU</i>		
		50,461 ± 560	51,502 ± 515	54,886 ± 3,356
		63,616 ± 443	63,554 ± 547	64,113 ± 114
		84,008 ± 1,263	83,647 ± 263	87,121 ± 853

6.3.2 Overhead of polling parts

In order to evaluate the overhead of polling policies, we have measured the average time to execute a given task consisting of compressing some files by using the *tar* utility when a polling policy was loaded and we have compared this time with the one needed when no polling was performed. In particular, we have compared the *CheckMem* policy presented in Listing 4 and the *CheckCPU* policy presented in Listing 5 when using different polling intervals (T).

As shown in Table 4, the overhead introduced by polling parts is very low, although one could expect some increment according to the complexity of the polling code. The impact of the polling interval is not noticeable in the experiments. Only the *CheckCPU* policy when T = 100 ms causes some increment in the execution time. Again, depending on the complexity of the policy code, lower polling intervals could have slightly higher impacts.

7 Conclusion and future work

In this paper, we have presented a solution to enhance the capabilities of Cloud IaaS middleware by allowing the behavior of the management system to be changed without re-coding the managers and shutting down the system. We have introduced the LEPIC language for the definition of management policies, the needed tools for parsing and generating code with LEPIC, and a policy management framework to allow the loading and enforcement of LEPIC policies dynamically within a Cloud middleware. We have demonstrated how LEPIC can be used in a real Cloud middleware, namely EMOTIVE Cloud, by defining some proof-of-concept policies and enforcing them in a real testbed. In addition, we have evaluated the overhead introduced by our policy management system, demonstrating that it is small. However, the low scalability of the EMOTIVE monitoring solution makes that the performance of the LEPIC management framework degrades quickly when the number of nodes in the infrastructure increases. Whereas the better scalability of the OpenNebula monitoring solution makes that its scheduler provides better times than LEPIC, especially for large datacenters, OpenNebula does not provide the reconfiguration advantages introduced by LEPIC.

We believe that this work can be very helpful for the administrators of Cloud providers, by giving them the needed flexibility to adapt the behavior of their system to their interests in every situation. Our future work includes the definition of complex policies with LEPIC, in order to detect potential limitations and new requirements. In addition, we will extend the functionalities of the underlying middleware that can be used in the policies, for instance, allowing to migrate VMs or to turn on/off physical nodes. Finally, we plan to port our solution to other Cloud middleware, such as OpenStack and OpenNebula.

Acknowledgments This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2012-34557, and by the Generalitat de Catalunya under contract 2009-SGR-980.

References

1. OpenNebula Home Page. <http://www.opennebula.org/>
2. OpenStack Home Page. <http://www.openstack.org/>
3. Eucalyptus Home Page. <http://www.eucalyptus.com/>
4. Gagnon EM, Hendren LJ (1998) SableCC, an object-oriented compiler framework. TOOLS 1998: 26th international conference on technology of object-oriented languages and systems, Santa Barbara, pp 140–154
5. EMOTIVE Cloud (Elastic Management Of Tasks In Virtualized Environments) Home Page. <http://www.emotivecloud.net/>
6. OpenNebula Scheduler Guide, Version 4.2. Tech. rep., C12G Labs (2012) <http://archives.opennebula.org/documentation:archives:rel4.2:schg>
7. OpenStack Configuration Reference, Havana Version. Tech. rep., OpenStack Foundation (2013) <http://docs.openstack.org/trunk/config-reference/content/index.html>
8. CloudStack Home Page. <http://incubator.apache.org/cloudstack/>
9. CloudStack Administration Guide, Version 3.0. Tech. rep., Citrix Systems Inc. (2012) <http://download.citrix.com/releases/3.0.0/CloudStack3.0AdminGuide>
10. CloudStack Advanced Installation Guide, Version 3.0. Tech. rep., Citrix Systems Inc. (2012) http://support.citrix.com/servlet/KbServlet/download/30165-102-685339/CloudStack3.0.0_3.0.2InstallGuide
11. Ganeti Home Page. <http://code.google.com/p/ganeti/>
12. Ganeti automatic instance allocation, Version 2.6. Tech. rep., Google Inc. (2012) <http://docs.ganeti.org/ganeti/current/html/iallocator.html>
13. Damianou N, Dulay N, Lupu E, Sloman M (2001) The Ponder policy specification language. In: Policies for distributed systems and networks, international workshop, POLICY 2001 Bristol, pp 18–38
14. Kagal L, Finin TW, Joshi A (2003) A policy language for a pervasive computing environment. In: 4th IEEE international workshop on policies for distributed systems and networks (POLICY 2003), Lake Como, p 63
15. JAXB project: Reference Implementation of JAXB, the Java Architecture for XML Binding. <https://jaxb.java.net/>
16. Canuto M (2012) Integrated policy management framework for the EMOTIVE Cloud middleware. Master's thesis, Universitat Politècnica de Catalunya and Politecnico di Torino
17. Metsch T, Edmonds A (2011) Open Cloud Computing Interface (OCCI) Infrastructure, Version 1. Tech. rep., Open Grid Forum. http://forge.ogf.org/sf/docman/do/downloadDocument/projects.oggi-wg/docman.root.drafts.oggi_specification/doc16162
18. Stress Workload Generator (2009) <http://people.seas.harvard.edu/apw/stress/>
19. OpenNebula Information Manager Drivers, Version 4.2. Tech. rep., C12G Labs (2012) <http://archives.opennebula.org/documentation:archives:rel4.2:devel-im>