SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2012; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

EMUSIM: An Integrated Emulation and Simulation Environment for Modeling, Evaluation, and Validation of Performance of Cloud Computing Applications

Rodrigo N. Calheiros[1*†], Marco A. S. Netto[2], César A. F. De Rose[3], Rajkumar Buyya[1]

[1]*Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Australia*
[2]*IBM Research, Sao Paulo, Brazil*
[3]*Faculty of Informatics, Catholic University of Rio Grande do Sul, Brazil*

SUMMARY

Cloud computing allows the deployment and delivery of application services for users worldwide. SaaS providers with limited upfront budget can take advantage of Cloud computing and lease the required capacity in a pay-as-you-go basis, which also enables flexible and dynamic resource allocation according to service demand. One key challenge potential Cloud customers have before renting resources is to know how their services will behave in a set of resources, and the costs involved when growing and shrinking their resource pool. Most of the studies in this area rely on simulation-based experiments, which consider simplified modeling of applications and computing environment. In order to better predict service's behavior on Cloud platforms, we developed an integrated architecture that is based on both simulation and emulation. The proposed architecture, named EMUSIM, automatically extracts information from application behavior via emulation and then uses this information to generate the corresponding simulation model. We performed experiments using an image processing application as a case study, and found that EMUSIM was able to accurately model such application via emulation and use the model to supply information about its potential performance in a Cloud provider. We also discuss our experience using EMUSIM for deploying applications in a real public Cloud provider. EMUSIM is based on an open source software stack and therefore it can be extended for analysis behavior of several other applications. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Cloud Computing; Emulation; Simulation; Performance Prediction and Validation

1. INTRODUCTION

Cloud computing has become a valuable platform for startups and small companies to deploy their application services for large-scale consumption on a pay-as-you-go basis. Such platform allows businesses to lease resources that can grow and shrink according to customer services' demand, thus offering an attractive option to build and maintain a data center to cope with a peak demand.

To better exploit the elastic provisioning of Clouds, it is important that Cloud application developers, before deploying an application in the Cloud, understand its behavior when subject to different demand levels. This allows developers to understand application resource requirements and how variation in demand leads to variation in required resources. This information is paramount

*Correspondence to: Department of Computing and Information Systems, The University of Melbourne, VIC 3010, Australia
†E-mail: rncalheiros@ieee.org

to allow proper Quality of Service (QoS) to be set and to estimate the budget required to host the application in the Cloud.

Experimentation in a real environment is (i) expensive, because it requires a significant number of resources available for a large amount of time; (ii) time costly, because it depends on the application to be actually deployed and executed under different loads, and for heavy loads long delays in execution time can be expected; and (iii) not repeatable, because a number of variables that are not under control of the tester may affect experiment results, and elimination of these influences requires more repetition of experiments, making it even more time costly. Therefore, other techniques for application evaluation are preferred. Two such alternative techniques available to developers are simulation and emulation. The main difference between them is the way software is represented during the evaluation process: simulation relies on models of software and hardware for evaluation, whereas emulation uses the actual software deployed in a model of the hardware infrastructure [1].

The different characteristics of these techniques make each of them more suitable to some activities than to others. For example, simulation can be used in the early development stages to evaluate concepts and strategies to be used in a project, whereas emulation is more suitable to be used once an application software prototype is already available. Moreover, simulation typically requires less hardware resources than emulation for the experimentation, and makes it easier for developers to test the model with a large number of application execution requests received from customers of the Cloud service, because these requests are also simulated; whereas emulation requires application execution requests to be actually generated and sent to the application under test (e.g., via a benchmarking software).

Even though simulation requires a smaller hardware platform for testing and enables easier evaluation of different scenarios, its utilization is difficult because it requires developers to correctly model the application behavior. If the application is not properly modeled, results obtained during simulation may not be achieved once the application is deployed. Therefore, an accurate model of the application is paramount for accurate and relevant simulations. A way of producing a better model of an application is through acquisition of relevant parameters obtained via analysis of software behavior during its execution.

To help developers to obtain more accurate models of their applications and to estimate performance and cost of the application in the Cloud, we propose an integrated environment—called EMUSIM—that combines emulation and simulation to extract information automatically from the application behavior (via emulation) and uses this information to generate the corresponding simulation model. Such a simulation model is then used to build a simulated scenario that is closer to the actual target production environment in terms computing resources available for the application and request patterns. Moreover, EMUSIM operates uniquely with information that is available for customers of public IaaS providers, namely number and characteristics of virtual machines, to perform the evaluation. We describe how each methodology is used and how they are combined, so data about application behavior is extracted during emulation and used to generate a more accurate simulation model of the application behavior. EMUSIM is built on top of two software systems: Automated Emulation Framework (AEF) [2] for emulation and CloudSim [3] for simulation. The proposed environment can also be extended to support other tools for these activities, as long as they provide an API that can be used to automate the process.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the motivation for the development of EMUSIM and a brief overview of the systems used in our architecture. Section 4 introduces EMUSIM and describes its architecture and operation details. Section 5 describes a use case to evaluate our approach. Section 6 discusses our experience developing EMUSIM, its limitations, and how to apply our approach with other simulation and emulation tools and Section 7 concludes the paper and proposes future research directions.

2. RELATED WORK

The use of discrete event simulation for evaluation of response time of applications in the Cloud has been successfully achieved in the CloudAnalyst project [4]. CloudAnalyst allows users to

DOI: 10.1002/spe

model scenarios where SaaS data centers and users are in different geographic locations. Output of experiments consists of response time for requests and cost of keeping the infrastructure, based on the Amazon EC2 cost policy. EMUSIM differs from CloudAnalyst in two ways. First, EMUSIM does not enforce any specific simulation scenario. Second, the data for modeling application behavior is extracted automatically from an emulation experiment.

The idea of combining simulation and emulation to leverage distributed systems experiments has been explored with different goals and different techniques.

GRAS [5], part of the SimGrid [6] project, allows researchers to use the same code for both simulation and actual deployment. Such a code uses a special library to allow communication among processes. Our approach, on the other hand, extracts performance information, via emulation, of already implemented software that may be written in any language and for any operating system that may run in a virtual machine.

RC2Sim [7] is a simulation and emulation-based tool for testing large-scale Cloud management systems. The system emulates an underlying Cloud provider and simulates virtual machine images on such infrastructure. The simulated virtual machines are subject to the management policies aimed at being evaluated by the tool. Our approach, on the other hand, focuses on the evaluation of applications running in the Cloud rather than the software managing the application and the platform.

Emuproxi [8] applies emulation to execute actual applications in a test environment and simulation to model network behavior in such an environment. Communication of the real application is forwarded to the simulation engine via VPN, and delivery of network packets in the destination is delayed according to simulation results. The main goal of simulation in Emuproxi is modeling the network of the emulation experiment, and it is applied simultaneously with emulation, whereas EMUSIM applies emulation and simulation in different stages of the experiment with different goals.

Netbed [9] is a platform that supports both simulation and emulation of networks for testing of distributed applications. Wide-area links are emulated in the links between actual cluster nodes used by the platform. Moreover, the system can be scaled with simulated nodes, links, and Internet traffic, allowing tests in larger scales than what is supported by the actual cluster nodes. The goal of simulation and emulation in Netbed is to increase scalability of the system, and therefore it does not support modeling of application behavior as does EMUSIM.

BigSim [10] supports simulation and emulation of parallel machines such as IBM's BlueGene. Emulation is used to execute actual applications, whereas simulation is used to model network latency and execute models of the application obtained during emulation. The target application of BigSim is MPI applications, which are the typical target for HPC machines. Our architecture focuses in master/slave applications, which represents applications more suitable to Cloud providers. Moreover, because BigSim does not rely on virtualization technology, tests in larger scales than what is supported by the cluster is only possible via simulation, whereas emulation allows larger scale of emulation experiments with utilization of virtual machines.

NEPI [11] is a tool that enables execution of simulation, emulation, and in-situ experiments from a single API and front end. NEPI provides a single description language and a GUI that is used as input for the experiments. NEPI has backends to several simulation, emulation, and in-situ tools that translate the NEPI input to the specific tool's input. Therefore, with a single input, users can create experiments combining approaches, or run the same experiment using different approaches. However, the goal of NEPI is to facilitate execution of experiments, and not to increase accuracy and scalability of experiments, as does our approach.

WORKEM [12] is an application service emulator for workflow applications. It receives tasks from workflow engines and, instead of executing them in actual resources, it emulates execution of the application using information available about it. WORKEM's goal is helping in designing, planning, and debugging of workflow applications, whereas the goal of emulation in EMUSIM is supplying an accurate representation of the application for a simulator. Because EMUSIM does not support emulation of workflow applications, WORKEM and EMUSIM could be used together for enabling accurate simulation of this type of application.

DOI: 10.1002/spe

Integrated simulation/emulation tools were also applied successfully in Sensor Networks. J-Sim [13] applies simulation in the application layer of sensor networks and emulation in the network level, so actual network packages are exchanged between simulated sensor nodes. Girod *et al.*, applied combined mixed simulation/emulation tools to support research and development of heterogeneous sensor network systems [14]. Our target environment, on the other hand, is distributed systems, and we apply emulation to extract information from actual applications and use simulation to test applications operating in large scales.

3. MOTIVATION AND BACKGROUND TECHNOLOGIES

Consider the scenario where a company wants to provide a rendering service to create videos based on frame descriptions submitted by their customers (referred as users in Figure 1). The application offered can be considered as a Software as a Service (SaaS) application. Moreover, the application is intrinsically CPU-intensive and can be parallelized by distributing different frames, or part of frames, to multiple CPUs to be processed.

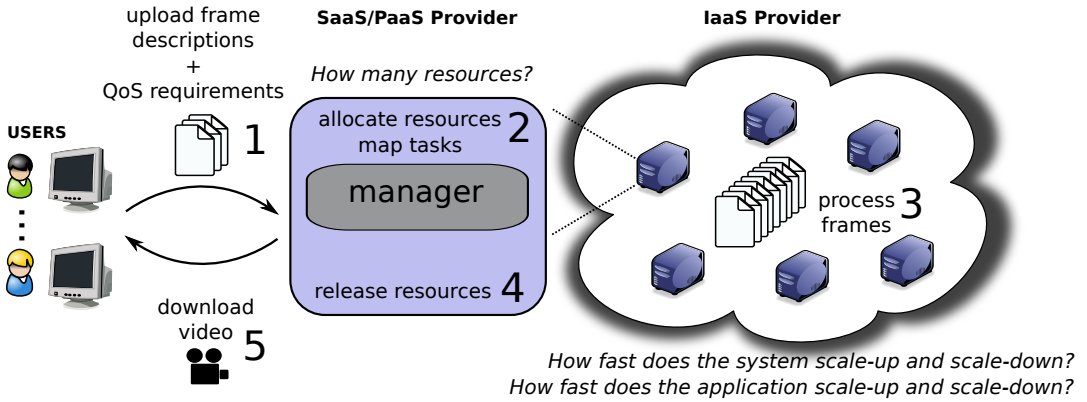

Figure 1. Case study scenario with a rendering service in a Cloud provider.

One of the main challenges of the service provider is to know how the application response time is affected by variable demand for the service [15]. Because multiple users can simultaneously request the service, they can share the same resources. The service provider may also want to scale-up the resource pool in order to meet users' QoS. However, time and cost are involved in the process of scaling up or scaling down both the rendering software and computing resources.

One method that could be used to define performance of the application depending on the number of requests and number of machines is via deployment and benchmarking of the application. However, this method requires deployment of the application in a physical infrastructure, which limits the scale of the experiment in terms of available servers or budget for running the test. Moreover, reproducing the experiment with different conditions requires generation, submission, and processing of user requests, which may be time-costly.

Without actually executing the application, it is difficult to model the application in a way it could run in a simulator, since its performance depends heavily not only on its own but also on characteristics of the infrastructure hosting the application. Therefore, if information about application performance under different loads is not available, the service provider needs another mechanism to improve the understanding of both its application and Cloud resources without having to deploy the services in the Cloud. This would then require paying for a large number of resources that will be used for testing purposes only, and so they are not generating any revenue for the service provider.

Our solution relies on improving simulation accuracy by extracting relevant application information via emulated executions of the service provider's application in a virtualized

DOI: 10.1002/spe

environment. Emulation allows execution of the actual application in a small scale environment that models the actual production infrastructure [1], whereas simulation allows assessment of how a system/application behaves in response to different conditions, such as different request arrival times and amount of work, in a controlled manner. Moreover, our solution operates uniquely with information that is available for customers of public IaaS providers, namely number and characteristics of virtual machines, to perform the evaluation. Information that is typically not disclosed by platform owners, such as location of virtual machines and number of virtual machines per host in a given time, is not required by EMUSIM.

Users of our solution are service providers, which are able, with the help of EMUSIM, to answer questions such as (i) *"how many resources are required to have a given response time considering a specific service request arrival distribution?"* (ii) *"how changes in the request arrival rate affect application response time?"*; and (iii) *"how changes in the number of resources affect application response time?"*

The answers of these questions can be used by Cloud software platforms to allow more efficient deployment of the SaaS application in the actual Cloud provider. Moreover, the estimation given by EMUSIM in terms of resources required by the software platform can be used to help application service providers to estimate the cost of running the application in public Cloud providers. In the rest of this section we briefly describe the technologies supporting our proposed solution for emulation and simulation of Cloud applications.

3.1. Automated Emulation Framework

Automated Emulation Framework (AEF) [2] is a framework for automated emulation of distributed systems. The target platform for AEF is a set of computer resources running a virtual machine manager (its current version supports Xen [16]). AEF requires two XML files for an emulation experiment. The first one describes the virtual environment to be used in the experiment. It consists of one or more sites containing a number of machines connected through a virtual WAN. Therefore, description of such an environment contains characteristics of machines on each site (e.g., memory, disk, and operating system) and characteristics of the virtual WAN (latency and bandwidth). The second file describes the application to be executed (for each machine, which application has to be executed, and files to be transferred).

Each machine defined by the user is converted into a virtual machine that is automatically mapped onto a computer node by AEF and deployed on the chosen host. More than one virtual machine may be created in a single host, as long as the amount of resources required by the VMs does not exceed the node's capacity. Network information is used to automatically configure a virtual distributed system in the computer infrastructure, in a way that isolation among virtual sites is respected and communication between sites occurs according to WAN parameters defined in the input file.

3.2. QAppDeployer

QoS-Aware Application Deployer [17] is responsible for managing the execution of applications and is started in one virtual machine by AEF during the emulation process. Similarly to Grid brokers [18–20] focusing on parameter sweeping executions [21], QAppDeployer is responsible for mapping application tasks to VMs, transferring application input data to each VM, starting application execution, and collecting the results from the VMs to the front-end node.

Figure 2 illustrates the main modules of QAppDeployer. The *task generator* receives the application and its parameters, and at the same time the AEF generates a machine file with all the available resources (step 1 and 2). The *scheduler* then selects a set of VMs that meet the defined requirements (step 3). The *task manager* starts an executor on each VM and transfers all required files for the application (step 4). The executors then fetch and execute tasks (step 5). Every time an executor finishes a task, it sends a message to the task manager asking for another task. QAppDeployer can be configured to send a group of tasks to each executor rather than a single task.

DOI: 10.1002/spe

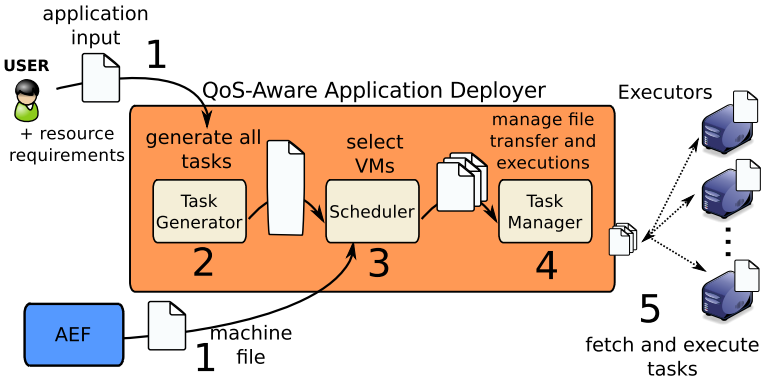

Figure 2. QoS-Aware Application Deployer.

3.3. CloudSim

CloudSim [3] is a development toolkit for simulation of Cloud scenarios. It supports modeling and simulation of data centers containing hosts with different hardware capacities; simulation of users accessing such services; and modeling of different algorithms for resource and VM provisioning, and scheduling of CPUs to VMs at virtual machine monitor level.

CloudSim is not a framework, i.e., it does not provide a ready to use environment for execution of a complete scenario with a specific input. Instead, users of CloudSim have to develop the Cloud scenario they wish to evaluate, define the required output, and provide the input parameters. Description of the scenario includes the definition of number of data centers in the simulation and their characteristics; definition of number of hosts on each data center; definition of hardware configuration of each host (memory, bandwidth, number of cores, and power of each core); and policies for allocation of hosts and CPU cores to virtual machines.

Besides the definition of the data center modeled in the simulation, CloudSim users have also to define the behavior of simulated Cloud customers. This includes number of customers; how and when customers request virtual machines and from which data center; and how customers schedule applications to create virtual machines. The exact meaning of "Cloud customer" is not defined by CloudSim. It means that one data center can request VMs and submit applications to other data centers (or even to itself) if the simulated scenario requires it.

4. EMUSIM ARCHITECTURE AND OPERATION

This section describes EMUSIM, its components and interactions, and execution flows. As stated previously, EMUSIM was developed with the use of state-of-the-art tools for simulation, emulation, and execution of applications. The three components in use (AEF for emulation, CloudSim for simulation, and QAppDeployer for load generation) are implemented in Java and have APIs that allow them to be used by other programs. Nevertheless, the same methods presented in this section can be applied to take advantage of other tools to carry on those tasks, as long as such tools can be externally accessed via an API.

4.1. Architecture

Figure 3 depicts the internal organization of EMUSIM and the role of each component. EMUSIM itself coordinates execution of different open source tools to work together in order to achieve its goal of accurate emulation and simulation of Cloud applications. It is responsible for activating each of the system components that set up the emulation environment with different configurations, coordinate dispatching of varying number of tasks for execution, and perform the simulations with proper configuration files.

DOI: 10.1002/spe

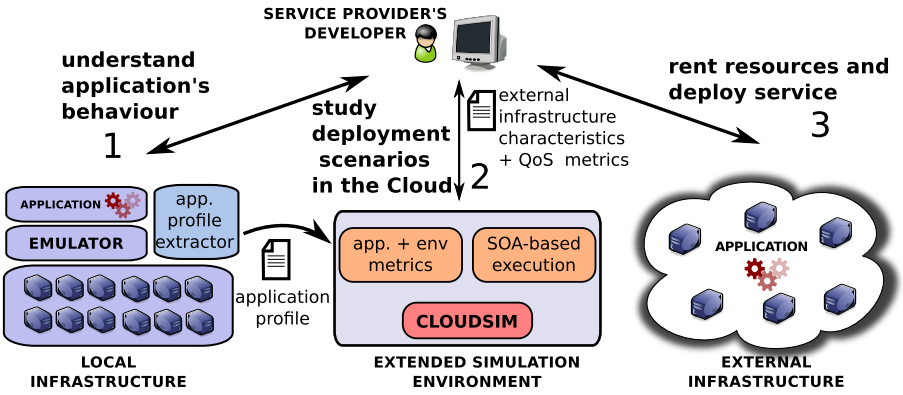

Figure 3. EMUSIM organization overview.

Basically, execution of EMUSIM is composed of two main activities. The first activity, *emulation*, is the stage where an application is submitted for execution with varying number of execution requests and varying number of available resources. In this step, actual instances of the application are executed in the environment that mimics the operation of a Cloud provider (therefore, we have the actual software executing in a model of the hardware, what is the definition of emulation proposed by Gustedt *et al.* [1])

For each request, the application is executed on the available hosts. The application itself is a Bag-of-Tasks (BoT) application, which means that tasks can spread over the available resources to speed up their execution time. Thus, each request for execution of the Cloud application translates to execution of one BoT application, and each application generates one or more tasks that are submitted to VMs. It allows the application to be evaluated regarding its scalability (i.e. how it behaves in the presence of different number of resources) or its divisibility (i.e. how it behaves with a fixed number of resources and different task partitioning). Resources in this case are virtual machines, which are deployed on a physical environment that is used by EMUSIM.

Virtualization enables isolated machines, behaving as actual servers, to be deployed on a physical infrastructure. This performance isolation is required for a proper extraction of application behavior information. Nevertheless, utilization of a virtualized environment is also required because it is widely used in Cloud providers. Therefore, utilization of virtualization allows us to better approximate the behavior of an application on its intended target environment.

In the second activity, *simulation*, profiling information extracted during emulation is used to model the application in the simulator. Unlike in the previous stage, what is executed in this stage is a *simulation model* of the software in a *simulation model* of the hardware, according to the definition of simulation given by Gustedt *et al.* [1]. The simulation allows evaluation of the application's behavior in the presence of different request arrival rate patterns for the application execution.

If an EMUSIM user wants to evaluate different simulated scenarios using the same application, the emulation stage of the experiment can be skipped: in this case, profiling information generated in a previous emulation execution can be reused, and only the input parameters of the simulation have to be reconfigured by users to reflect the new evaluation scenario. This enables quick evaluation of different simulated scenarios after a single round of (potentially slow) emulation.

To enable such activities to run automatically, EMUSIM requires the following input:

Description of the physical environment hosting the emulation. The emulator needs to know about the physical machines available to host the virtual machines used in the emulation. Because EMUSIM is based on AEF, this information is supplied in the form of an XML file;

Description of the emulated environment. This includes information such as minimum and maximum number of VMs necessary during the emulation. Moreover, VM image to be used is also required information. In EMUSIM, this is configured in a properties file;

DOI: 10.1002/spe

Algorithm 1: General operation of the emulation stage of EMUSIM.

```
   Data: environment: description of the emulated environment.
   Data: maxUsersPerVm: maximum number of simultaneous requests sent to a VM by the
         platform.
   /* environment initialization                                          */
 1 machinesList ← ∅ ;
 2 resultsList ← ∅ ;
 3 emulator.createEmulatedEnvironment(environment);
 4 platform.startService;
 5 foreach number of VMs tested do
 6 |   machinesList ← machinesList ∪ new VMs;
 7 |   platform.setMachines(machines);
 8 |   for users from 1 to maxUsersPerVm do
 9 |   |   simultaneously send to the platform one request per users;
10 |   |   foreach generated request do
11 |   |   |   summary ← platform.getExecutionSummary;
12 |   |   |   resultsList ← resultsList ∪ summary;
13 |   |   end
14 |   end
15 end
```

Application configuration. Because emulation requires execution of an actual application, it is commonly required by the application to receive some configuration. Such configuration can be available in the VM image or exposed to EMUSIM in a file that is read as part of experiment initialization;

Simulation configuration. It consists of a file that contains information of the simulation scenario where the application is evaluated: number of application users, application user request arrival pattern, QoS metrics; and also configuration of the simulated data center: number of hosts, its capacity, number of virtual machines, internal policies for provisioning and so on.

The above information, given in the form of four configuration files, is the only input required by EMUSIM users. The whole process of deployment of VMs, submission of parallel applications, acquisition and generation of application profile, execution of simulation, and generation of execution report is managed by EMUSIM itself. The execution report contains statistical information about expected QoS of the SaaS application in the presence of the simulated workload. SaaS providers, and/or PaaS services deployed on their behalf, then use this report to drive decision about number of VMs to be used to host the SaaS application.

4.2. Operation

In this section, we describe the operation of EMUSIM. As it was stated before, it is composed of two stages: emulation and simulation. Emulation operation is presented in Algorithm 1.

After initialization of the emulator (Line 3) and the platform (QAppDeployer—Line 4), EMUSIM starts all the required VMs progressively to compose the test environment (Line 5). This is done by dynamically adding VMs to the platform. The number of VMs tested starts from one and goes until the maximum number of VMs, as set in the environment description file. Users can control VM increments on each number of VMs tested. By default, it executes experiments with three numbers of VMs: one, the maximum number of VMs (as specified in the configuration file), and the average of these two values.

The number of VMs is increased by submission, via *ssh*, of a file containing the list of machines to be used. Then, the order policy for reading the file is sent via sockets to a listener in QAppDeployer (Lines 6 and 7).

DOI: 10.1002/spe

Algorithm 2: General operation of the simulation stage of EMUSIM.

Data: $round$: number of repetitions of the simulation.
Data: $applicationProfile$: description of application behavior, generated during the emulation.
`/* validation round */`
1 **foreach** *number of VMs tested* **do**
2 | runValidationRound($applicationProfile$,number of VMs);
3 **end**
`/* generalized workload */`
4 **foreach** $round$ **do**
5 | runSimulationRound($applicationProfile$);
6 **end**

After platform reconfiguration, one request is sent for execution. This request is translated to a BoT application that contains one task for each available VM. These tasks are sent to the available VMs and EMUSIM waits for their execution. Once the whole request execution finishes, a summary of the execution, containing number of available VMs (which defines task parallelism), number of simultaneous requests (concurrency), and execution time, is retrieved (Line 11). This information is later used to extract profiling information from the application.

Once all the requests are completed, the number of requests is increased by one (simulating one more user request). These requests are then simultaneously sent to the QAppDeployer (via threading to allow better concurrency than obtainable in sequential execution). When all the summaries are received, the number of requests is again increased by one, requests are simultaneously sent again, and summaries are retrieved. The process is repeated (Lines 8 to 14) until the maximum number of simultaneous user requests, as defined in the input file, is achieved.

Simultaneous submissions during emulation are carried out in order to determine how concurrency affects execution time of tasks. Similarly, the number of VMs is scaled to allow determination on how parallelism affects the application. During the simulation, EMUSIM does not assume that requests arrive simultaneously, and different submission patterns can be utilized.

The raw profiling information is then saved in a file, and automatically processed by EMUSIM, that extracts relevant information from them (see next section) to generate information relevant to the simulation. Execution of the simulation is depicted in Algorithm 2.

Simulation occurs in two steps. The first one is a validation round (Lines 1 to 3) to verify how accurately the application performance model can be represented in the simulation. To do so, the data center is configured with the same number of resources and runs a workload that reproduces the load generated during the emulation. The output of this stage can be compared with the output from the emulation, and it allows determination of the accuracy of the simulated model. Results of this stage allow EMUSIM users to decide whether the model of the application achieved the accuracy they expected or not. In case of the former, results of the next simulation step are accepted and can be used as an indicative of the performance of the application in the Cloud. In case of the latter, results can be rejected and the emulation step can be repeated, probably increasing the number of generated data points.

In the second simulation step, generalized workload (Lines 4 to 6), a data center with the configuration defined by the EMUSIM user is generated. This data center receives requests according to a workload also defined by EMUSIM users. Execution of the workload will cause variable load in the machines used for the application. By considering the load and arrival of requests, execution time of incoming requests are calculated by the simulator and applied in such request. Each request generates a number of tasks that are submitted for execution on the simulated VMs. Once all tasks complete, the corresponding request is considered completed and its execution time is calculated. Another situation that can emerge is that the platform can reject new requests when there are no available resources to handle them. EMUSIM also tracks the number of rejected requests, as it is also a QoS metric.

DOI: 10.1002/spe

The final report contains the total number of requests submitted, number of executed and rejected requests (both in absolute values and in rate), and average and standard deviation of response time. The whole process of running the generalized workload step can be automatically repeated at EMUSIM user discretion: users can define in the input properties file the number of times the simulation needs to be repeated. This is especially useful for simulation that contains randomized factors in any part of its scenario, such as a workload that is based on probability distributions.

4.3. Performance Model

The execution of the emulation stage generates data on the application performance with different parallelism levels, i.e. how long the application takes to complete considering its decomposition in different number of tasks that execute on independent machines, and also with different levels of concurrency, i.e. when a single application task shares resources with other tasks that belong to different requests.

The execution times obtained during the emulation stage are stored and processed by an EMUSIM Python script, which collects the average response time for tasks executed with the same parallelism and concurrency level. This is stored in a table that is indexed by concurrency and parallelism level.

The information in such a table is used in the simulation stage to estimate request run time in conditions that are different from the conditions found during the emulation. It means that in the simulation, requests can be divided in more parallel tasks than those achieved in the emulation or they can share resources with more tasks than during the emulation. Therefore, a form of extrapolation of execution time is required in the simulation stage.

Considering a bi-dimensional space composed of the concurrency level as one dimension and the parallelism as the second dimension, response time of a single request in the simulation stage is estimated as follows. If a response time for the given concurrency and parallelism level is defined (either because the case was executed during emulation or because it was calculated before during simulation), the value is retrieved and used as the runtime estimate for the request.

If an estimate for the runtime with the given concurrency and parallelism level has not been calculated yet, an estimation is generated according to Algorithm 3 (Lines 1 to 12). The estimation is based on a linear extrapolation of the runtime considering the two closest points defined during emulation, first extrapolating the concurrency level (if necessary) (Lines 13 to 22) and then the parallelism level (Lines 23 to 32). This is because the parallelism level is evaluated more sparsely than concurrency level during the emulation. If the estimation was executed before, the value is retrieved (Lines 15 and 25).

Once an estimate is calculated, it is stored in the table for future utilization in the simulation (Lines 20 and 30), when a request with the same conditions is submitted to the application deployer. The module to calculate the estimation can be replaced or extended according to the target computing environment.

5. PERFORMANCE EVALUATION

In this section, we present an evaluation of EMUSIM. We describe the hardware and software components, along with metrics and analysis of results.

5.1. Experiment Setup

The physical environment used in the experiment is a cluster composed of five nodes and one physical front-end. All the machines are dual-core AMD Opteron 2 GHz with 8GB of RAM and 250GB of storage running Xen 3.4.0. The cluster front end node runs Oracle Virtual Machine (OVM) server, as well as EMUSIM and its components. The virtual environment comprises 10 virtual machines, each with 1GB of RAM, 1 CPU core, 5GB of storage and Ubuntu Linux Operating System running on the cluster nodes (therefore 2 VMs per node). QAppDeployer runs on a VM with the same configuration as the described VMs, but it was deployed on the physical front end rather than on the physical cluster worker nodes. EMUSIM is installed and runs on a privileged

DOI: 10.1002/spe

Algorithm 3: Estimation of request runtime during the simulation.

Data: table: stores known runtimes for pairs of parallelism and concurrency.
Data: capacity: processing power of the simulated processors.
Data: p_0 : parallelism; number of tasks that compose the incoming request.
Data: c_0 : concurrency; number of requests running concurrently with the incoming one.

```
1  estimateRuntime(p0, c0)
2      p1 ← closest parallelism level to p0 obtained via emulation;
3      p2 ← second closest parallelism level to p0 obtained via emulation;
4      c1 ← closest concurrency level to c0 obtained via emulation;
5      c2 ← second closest concurrency level to c0 obtained via emulation;
6      if (c0 > c1) and (c0 > c2) then
7          extrapolateConcurrency(p1, c0, c1, c2);
8          extrapolateConcurrency(p2, c0, c1, c2);
9      end
10     length = extrapolateParallelism(t0, c0, p1, p2);
11     return length*capacity/c0;
12 end
13 extrapolateConcurrency(p, c, c1, c2)
14     if table.contains(p, c) then
15         return table.get(p, c);
16     end
17     r1 ← table.get(p, c1);
18     r2 ← table.get(p, c2);
19     r0 ← r1 + ((c−c1)*(r2−r1)) / (c2−c1);
20     table.put(p, c, r0);
21     return r0;
22 end
23 extrapolateParallelism(p, c, p1, p2)
24     if table.contains(p, c) then
25         return table.get(p, c);
26     end
27     r1 ← table.get(p1, c);
28     r2 ← table.get(p2, c);
29     r0 ← r1 + ((p−p1)*(r2−r1)) / (p2−p1);
30     table.put(p, c, r0);
31     return r0;
32 end
```

virtual machine (Xen's *Dom0*) on the physical front end, so that it has administrative access rights to trigger the VM deployment process on the cluster nodes.

The service application used in the experiment is an image rendering application based on the Persistence of Vision Ray tracer (POV-Ray) ray-tracing [22] application. Requests for such service trigger a process of rendering an image. The process is split across available VMs. As described in the EMUSIM operation, the emulation process is repeated with 1, 5, and 10 VMs in this experiment.

Once the emulation runs, profiling files are automatically generated and the simulation stage is triggered. Such a simulation consists in executing the same service application with both the validation workload (to show how close simulated and emulated application performance are from each other) and the workload model for Bag-of-Tasks (BoT) grid applications defined by Iosup *et al.* [23]. According to this workload model, inter-arrival time of a BoT application in peak time (between 8am and 5pm), and the number of requests received in a 30 minutes period in off-peak time (which is called daily cycle) follow Weibull distributions with parameters (4.25,7.86) and

DOI: 10.1002/spe

Table I. Execution times of different steps of an EMUSIM experiment.

	Step	Time (s)
Emulation	Initial VM deployment	1867
	Execution of emulation stage	2207 (Box), 10390 (Vase)
Simulation	Generation of application profile	0.02
	Execution of simulation validation round	0.4 (Box and Vase)
	Execution of simulation application	89.21 (Box) 126.61 (Vase)

(1.79,24.16), respectively. In the latter case, we assume that requests arrive in equal intervals inside the 30 minutes period.

Once the QAppDeployer receives a request to execute the application, such request is translated into a BoT job request. The number of tasks for such a new job is defined according to the workload model: the class of such a request is determined following a Weibull distribution with parameters (1.76,2.11). The class defines the number of tasks in a logarithmic scale: first class has 1 or 2 jobs, second class has 3-4 jobs, third class has 5-8 jobs, fourth class has 9-18 jobs, and so on until the ninth class that has 1000 tasks. The actual number of requests is generated uniformly from the range of allowed number of tasks from the chosen class. Execution time of each task in the simulation is obtained by analysis of the emulation results, which were validated in the validation round of EMUSIM's simulation stage.

The whole process has been run with two rendering scenarios: the first one, *Box*, comprises the approximation of a camera to an open box with objects inside. Time for rendering frames in this scenario increases during execution. In the second scenario, *Vase*, the rendering comprises rotation of a vase with mirrors around. Time for processing different frames in this scenario is constant.

Because requests might arrive to the application deployer in a higher rate than the rate on which requests finished, some concurrency is incurred on the requests. We allowed up to five tasks to run concurrently on each virtual machine. Every time a new request is received, the simulated application deployer looks for one VM for each task that composes the request. If there is room for all tasks of the request, it is executed; otherwise, the request is rejected.

The simulation rounds consist of a simulation of 24-hour length submission of jobs to the simulated data center according to the workload described above. Each simulation round was repeated 30 times, and the average of output metrics is reported. They are request acceptance and rejection rates, and request response time. The process was repeated by varying the number of virtual machines in the data center from 20 to 70, in steps of 10.

The time required to run each step of the experiment described in this section is shown in Table I. Deployment times are initially large because VM images have to be transferred to the hosts where they are deployed. Notice that AEF implements a cache mechanism in which, if a further deployment of the same image is required, the transfer process is skipped. In the table, *Execution of emulation stage* denotes the time to run the application for each number of VMs and for each number of concurrent requests. This time is application-dependent so the time taken to each rendering scenario is shown.

Table I also shows that the time taken to process the emulation output and generate the simulation-independent file is negligible, when compared with the rest of the experiment execution. Interpretation of the file generated in such a stage is performed during the simulation stage. Finally, execution of simulation stages (both validation and actual experiment) is also small compared to emulation times. Because the output of a single emulation can be used to multiple simulations, we expect the deployment and emulation times to be amortized along the time.

5.2. *Results and Analysis*

Figure 4 presents the accuracy of application models generated by EMUSIM. In the plots, circles represent the average application execution time, captured during emulation, with different number of tasks (parallelism level) and with different concurrent tasks running on the same machine

DOI: 10.1002/spe

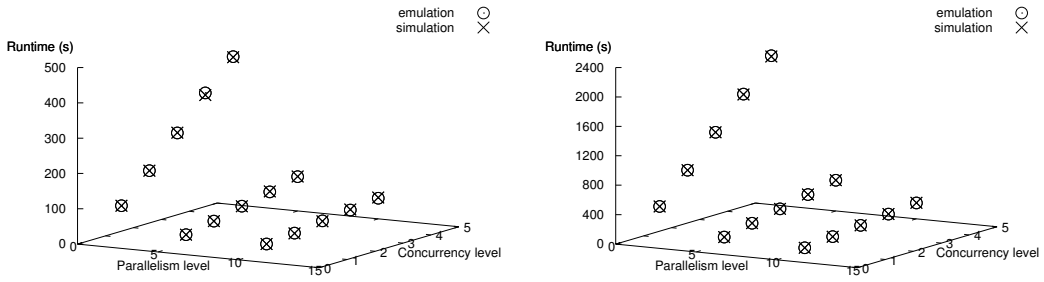

Figure 4. Accuracy of simulated applications (crosses) compared to emulated results (circles) for different rendering scenarios (a) Box (b) Vase.

(concurrency level). Crosses represent the average execution time of applications, in the same conditions regarding parallelism and concurrency, observed during validation rounds of simulation, for both rendering scenarios (Box and Vase). Coefficients of Variation of the simulated response times were smaller than 7%, and for most of cases below 3%.

The results show that execution times during simulation were very close to execution times during emulation and thus EMUSIM was able to accurately model our two real applications. Moreover, results show no trends regarding variation in parallelism and concurrency and increase in the deviation between emulation and simulation results.

Figure 5 presents rates of accepted and rejected requests, for each rendering scenario, when different numbers of VMs are deployed for the application. As expected, the more VMs are available in the data center, the lower the rejection rate. We observed that increase in number of VMs in the Vase scenario has a bigger effect in the acceptance rate than in the Box scenario. This is because the Vase application has a longer execution time. Therefore requests stay more time in the VMs being processed, thus occupying resources and contributing for further requests to be rejected. It means that the higher the execution time of a single request, the more VMs should be made available for processing it.

The results also show that the Vase application reaches more than 99% of accepted requests with 60 VMs, whereas the same is achieved for the Box scenario with 70 VMs, even though 60 VMs give already over 95% acceptance rate. Therefore, a provider of a rendering application on the Cloud expecting access patterns that resembles those of Grid BoT applications would have to deploy 70 static VMs to keep 99.9% acceptance rate. Rejections cannot be completely eliminated because there are always some bursts generated by the distribution that exceed system capacity. Nevertheless, each simulation generated in average 5156 requests, therefore an acceptance rate of 99.9% means that in average only five requests were rejected. This could be improved by increasing concurrency level (at the cost of higher response times in peak periods), with more VMs deployed (at the cost of smaller utilization), or with the utilization of some technique for dynamic provisioning of VMs, which would enable higher utilization of VMs at the same time response time is kept at acceptable levels.

Regarding average service time of requests, results for both rendering scenarios are depicted in Figure 6. Coefficient of variance for service times was between 0.2 and 1.6, with 10 out of 12 measurements points having the value above 1.0. The variance in service time is high because the run time is directly affected by the concurrency level: in periods when system load is low, concurrency level is low and applications can run in parallel without sharing resources with other processes. Nevertheless, we allowed up to five concurrency requests per VM, which delays the application by five times in peak time. Had we configured the system with a higher concurrency level, the expected variance would be even higher.

DOI: 10.1002/spe

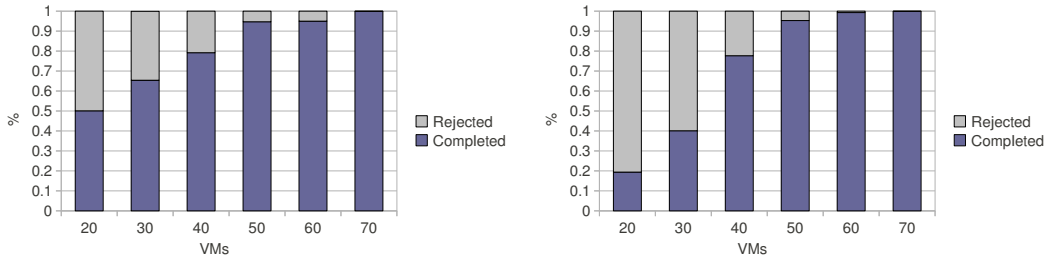

Figure 5. Acceptance and rejection rates of requests with different number of available VMs for different rendering scenarios (a) Box (b) Vase.

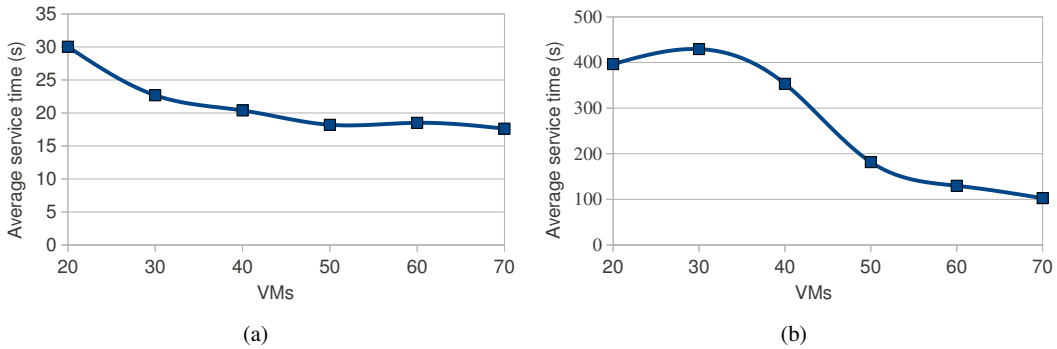

Figure 6. Average service time of requests with different number of available VMs for different rendering scenarios (a) Box (b) Vase.

Results for response time also show that reduction in the average response time for the Box scenario presented little variation for more than 50 machines. After this amount of resources, the extra resources enable more requests to be served, however they are accepted in peak times, when response time is high. The higher parallelism level achieved with the extra resources in low traffic times are compensated by the time extra requests take to execute in peak times, and then no improvement in response time can be seen, even though improvement in acceptance rate was observed.

The same trend was not observed for the Vase scenario, which has been observed as more sensitive to the number of VMs available in the data center. Moreover, we observed a slight increase in the response time when 30 VMs were used instead of 20. This is caused by the biggest execution time of the application in this scenario, which keeps resources in use for more time and makes the system operate at a higher concurrency level.

Therefore, if response time observed by customer is an issue, and a fixed number of resources have to be divided among applications (for example, because of budget limitations), providers should assign more resources to the Vase rendering scenario.

Moreover, because EMUSIM users can configure simulation, other deployment scenarios could also be modeled and applied for evaluation purposes. For example, dynamic provisioning techniques could be applied for changing the number of VMs available to applications accordingly to the expected load in different periods of time.

Finally, by enabling application performance evaluation to be executed in a small number of virtualized servers that possibly exist in an organization, EMUSIM enables savings in terms of investment for experimentation purposes. For example, each simulation round of the evaluation presented in this section requires 24-hour long executions. Therefore, reproducing our experiment for the same number of VMs (from 20 to 70) for both scenarios in Amazon EC2 would require

DOI: 10.1002/spe

12960 hours of VM usage for the execution nodes plus at least one extra node all the time for load generation. This represents a cost of U$ 1126.08 if the cheapest Linux small instances (single core, one EC2 compute unit, 1.7 GB of RAM, 160 GB of storage) are used. If the experiment would be repeated 30 times, the total cost would be at least U$ 33782.40. Therefore, EMUSIM is a cost-effective approach for evaluation of Cloud applications.

5.3. Validation in a Public Cloud

In order to evaluate accuracy of the extrapolation procedure of service time in EMUSIM, we evaluated the outcomes of the simulation stage of EMUSIM in a public Cloud. The evaluation has been performed as follows. The output of the emulation step for the Vase rendering scenario (described in Section 5.1), which contains emulation of up to 10 VMs and five concurrent tasks per VM, has been submitted to a new simulation experiment. Such new simulation consists of a simulated data center with 15 virtual machines that receive simultaneous submissions of the same numbers of requests performed during the emulation stage of EMUSIM.

The same setup is reproduced in Amazon AWS EC2. 16 VMs are deployed for the experiment purposes: one instance is the frontend, which generates the requests and submits to the QAppDeployer (which is hosted in the same machine). The other 15 VMs are workers that process the requests. All the VMs are Amazon EC2 High-CPU Medium Instances, which have 1.7 GB of RAM, and two CPU cores, each with 2.5 Compute Units (which Amazon defines as the capacity of one 1.7 GHz Opteron[†] 2006 machine) and cost of U$ 0.17 per instance per hour. The operating system of the VMs is Ubuntu Linux 11.10. We deployed this instance type because the nominal processing power of one core of this instance type is closer to the power of the machine used in the emulation stage. All the VMs were deployed in the same availability zone in the USA East Coast, to minimize network overhead.

Requests for execution were generated in the frontend. Likewise in the simulation, submissions go from 1 to 5 simultaneous requests, and each request is distributed to the 15 VMs. We collected service times obtained for each request. To compensate for the differences in CPU capacity between the machines used in the emulation and the Amazon machines, service times were normalized with the service time obtained with a single request (i.e., concurrency level equals to one).

Figure 7 presents results for this experiment. The figure shows that the difference between service times obtained during simulation and during the public Cloud experiment increases as the concurrency level increases. This is caused by the difference in performance between the machines used in the emulation and the machines used in the Cloud. Because the service time is based in execution of a CPU-intensive application and data transfer, the computation-communication rate of the application is different in both scenarios, and this causes a deviation between speed ups obtained in the local resources and in the Cloud, what results in estimation errors. We detail this effect and how it can be tackled in the next section. Nevertheless, considering the differences in performance between the machines (cluster machines performed 50% better than Cloud machines for one request in one VM), results are satisfactory. We also expect that, as IaaS Cloud technology matures, specification of actual capacity of Cloud resources will become more precise and enforced via Service Level Agreements (SLAs), what will make differences in performance more predictable, reducing the gap between results generated by analytical tools and simulation models and results obtained with actual execution of applications.

6. DISCUSSION

As previously discussed by Gustedt *et al.* [1], there are different methodologies for evaluation of distributed systems, and the definition of the most suitable approach depends on the objectives and restrictions of the envisioned experiment. Emulation allows evaluation of actual software, but it has

[†]http://aws.amazon.com/ec2/instance-types/

DOI: 10.1002/spe

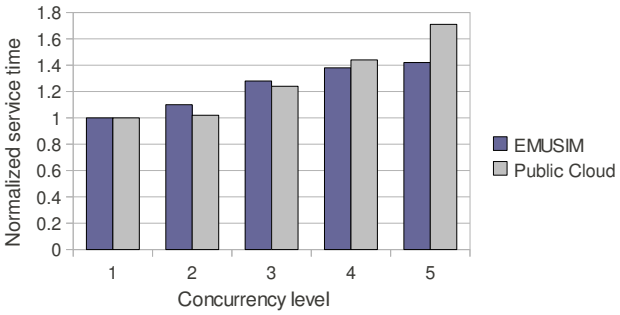

Figure 7. Comparison of normalized service times, with 15 VMs and different concurrency levels, provided by EMUSIM and through execution of the application in a public Cloud provider, for the Vase rendering scenario.

limitations regarding scalability due to either hardware constraints or difficulty in generating large and realistic workloads.

Simulation does not have these scalability limitations, but simulation results are as accurate as the model of the software submitted to the simulator. Therefore, if testers want to evaluate software in a simulator, they need to develop an accurate model of it. Our goal was to develop the process of extracting the model of software so it can be evaluated in a simulator. Another requirement we included in our design was that the process should be automated. In addition, our solution should work in a wide range of available software and hardware, and in a scale that even small companies and research laboratories could also support.

With the above guidelines in mind, and taking advantage of our previous experience with emulation tools [2], we decided to use emulation to support real application execution. Even though we used AEF for this purpose, any emulator that supports execution via scripts could be used for this purpose.

In our system design, output from the emulation stage and input to the simulation stage were loosely coupled: the emulation generated a performance file that was later translated into a simulation model of the application. We chose this design because it allows EMUSIM users to replace any part of the architecture without major changes in the core EMUSIM functionality. This is important because different simulators are used for different modeling purposes; therefore, EMUSIM users can choose the simulator that best suits their needs and only implement a converter to transform the emulation output to simulation input. Other actions (like correctly invoking the simulator) are performed via scripting.

To enable the use of our architecture in a wide range of software and hardware platforms, we opted for tools that rely on platform virtualization technology. The current AEF implementation, and therefore EMUSIM, supports Xen for this purpose. However, a port for supporting VMware can be easily developed in Java for AEF by extending interfaces designed to mediate the process of VM deployment. With a little extra development in EMUSIM, a platform management tool such as Eucalyptus could replace AEF. This, however, would require testers to develop solutions for executing applications in such an infrastructure. We believe that, as private Clouds are increasing in popularity, adding this support for tools such as Eucalyptus and another layer for management application is an interesting topic to be investigated in future developments of this research.

6.1. Opportunities for Enhancements

The results of experiments show that accuracy of the simulation increases as the difference in CPU power of the machines used during emulation and the Cloud machines decreases. This is because the computation-communication rate of the application changes: we observed a similar latency and bandwidth in both environments, so the time for data transfer for user requests was similar in both environments, but the processing times on each environment varied. We expect this effect to be less evident for applications with little data transfer, as most service time will be caused by application

DOI: 10.1002/spe

processing. Moreover, advances in virtualization technology, enabling a finer control on the amount of CPU allocated to a virtual machine, or even a more precise definition of a machine's computing power, enforced via SLAs, might also mitigate this effect.

The following approach could also be adopted in EMUSIM to tackle this limitation. A plugin could be added to the tool that can collect information about data transfer time for each request. The plugin would enable determination of the specific contribution of data transfer and computation on the total service time of requests. The service time extrapolation procedure would have to be modified to independently extrapolate each of these times to determine the contribution of each of them to the total simulated service time.

7. CONCLUSIONS AND FUTURE WORK

Even though Cloud is already a consolidated platform for service applications deployment, an efficient utilization of its resources depends on understanding the characteristics of the deployed application. This understanding can be achieved with actual deployment, which is risky and cost-ineffective, or can be achieved with support tools that allow deployment in a small scale and extrapolation of results to the larger Cloud scale.

In this paper, we presented EMUSIM, an architecture that combines emulation (for extracting profiling information of the actual application) and simulation (for evaluation of higher resource scale and variable request workloads) to evaluate the effect of different number of resources and patterns of requests for Cloud applications. Experiments showed that EMUSIM was able to accurately model our two applications as a simulation model and use it to supply information about their potential performance in a Cloud provider.

By using a small number of virtualized computers available in an organization, EMUSIM allows accurate representation of applications that can be timely evaluated in terms of performance in a simulated target higher-scale Cloud infrastructure. This is especially useful when the corresponding in-situ experiment has a complex input workload, what would require a special infrastructure and resources just to generate the input load to the system, or when the experiment is supposed to run for a long time, because simulation can speed up the experiment process.

Besides the savings in time and operational efforts for the evaluation, EMUSIM also reduces costs for running such an evaluation, because a local, small-scale infrastructure, rather than a pay-as-you-go public Cloud, is used for evaluation purposes.

In the current version, EMUSIM supports loosely coupled CPU-intensive applications. We plan to extend EMUSIM to support other types of applications such as Web servers, DBMS, and parallel applications. Finally, we also plan to extend our tool to support modeling and evaluation of applications composed of dependent services, which may run in the same data center or in different ones. This class of applications includes workflows, mash-ups, and multi-tier enterprise applications. We also plan to extend EMUSIM to support evaluation of performance of applications when resource allocation at infrastructure-level (such as number of virtual machines in a host) changes. Finally, we plan to develop plugins to allow extraction of more information during emulation (such as network utilization), which will contribute towards enhancement of the precision of evaluation studies.

ACKNOWLEDGEMENTS AND SOFTWARE AVAILABILITY

Authors would like to thank Sivaram Yoganathan, Dileban Karunamoorthy, and the anonymous reviewers for their insightful feedback. CloudSim is available for download at www.cloudbus.org/cloudsim. EMUSIM source code, containing all the subcomponents, is available at www.cloudbus.org/cloudsim/emusim.

REFERENCES

DOI: 10.1002/spe

1. Gustedt J, Jeannot E, Quinson M. Experimental methodologies for large-scale systems: a survey. *Parallel Processing Letters,* September 2009; **19**(3):399–418.
2. Calheiros RN, Buyya R, De Rose CAF. Building an automated and self-configurable emulation testbed for grid applications. *Software: Practice and Experience,* April 2010; **40**(5):405–429.
3. Calheiros RN, Ranjan R, Beloglazov A, De Rose CAF, Buyya R. CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience,* January 2011; **41**(1):23–50.
4. Wickremasinghe B, Calheiros RN, Buyya R. CloudAnalyst: A CloudSim-based visual modeller for analysing cloud computing environments and applications. *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA'10)*, 2010.
5. Quinson M. GRAS: A research & development framework for grid and P2P infrastructures. *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'06)*, 2006.
6. Casanova H, Legrand A, Quinson M. SimGrid: a generic framework for large-scale distributed experiments. *Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation (UKSim'08)*, 2008.
7. Citron D, Zlotnick A. Testing large-scale cloud management. *IBM Journal of Research and Development* November 2011; **55**(6):6–1:6–10.
8. Liu J, Mann S, Vorst NV, Hellman K. An open and scalable emulation infrastructure for large-scale real-time network simulations. *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM'07)*, 2007.
9. White B, Lepreau J, Stoller L, Ricci R, Guruprasad S, Newbold M, Hibler M, Barb C, Joglekar A. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review,* December 2002; **36**:255–270.
10. Zheng G, Wilmarth T, Jagadishprasad P, Kalé LV. Simulation-based performance prediction for large parallel machines. *International Journal of Parallel Programming,* June 2005; **33**(2):183–207.
11. Lacage M, Ferrari M, Hansen M, Turletti T, Dabbous W. NEPI: using independent simulators, emulators, and testbeds for easy experimentation. *ACM SIGOPS Operating Systems Review,* January 2010; **43**(4):60–65.
12. Ramakrishnan L, Gannon D, Plale B. WORKEM: Representing and emulating distributed scientific workflow execution state. *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, 2010.
13. Sobeih A, Hou J, Kung LC, Li N, Zhang H, Chen WP, Tyan HY, Lim H. J-Sim: a simulation and emulation environment for wireless sensor networks. *IEEE Wireless Communications,* August 2006; **13**(4):104–119.
14. Girod L, Stathopoulos T, Ramanathan N, Elson J, Estrin D, Osterweil E, Schoellhammer T. A system for simulation, emulation, and deployment of heterogeneous sensor networks. *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004; 201–213.
15. Netto MAS, Buyya R. Offer-based scheduling of deadline-constrained Bag-of-Tasks applications for utility computing systems. *Proceedings of the 18th International Heterogeneity in Computing Workshop (HCW'09)*, IEEE, 2009.
16. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.
17. Emeakaroha VC, Calheiros RN, Netto MAS, Brandic I, , De Rose CAF. DeSVi: An Architecture for Detecting SLA Violations in Cloud Computing Infrastructures. *Proceedings of the 2nd International ICST Conference on Cloud Computing (CloudComp'10)*, 2010.
18. Elmroth E, Tordsson J. A grid resource broker supporting advance reservations and benchmark-based resource selection. *Applied Parallel Computing. State of the Art in Scientific Computing*, *Lecture Notes in Computer Science*, vol. 3732. 2006; 1061–1070.
19. Abramson D, Buyya R, Giddy J. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems,* August 2002; **18**(8):1061–1074.
20. Krauter K, Buyya R, Maheswaran M. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience,* February 2002; **32**(2):135–164.
21. Abramson D, Giddy J, Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global grid? *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, 2000.
22. Glassner AS ((ed.)). *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.
23. Iosup A, Sonmez O, Anoep S, Epema D. The performance of bags-of-tasks in large-scale distributed systems. *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC'08)*, 2008.

DOI: 10.1002/spe